May 7–11, 2024
London, United Kingdom

Association for Computing Machinery

Advancing Computing as a Science & Profession

ICPE '24

Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering

Sponsored by:

ACM SIGMETRICS, ACM SIGSOFT, & SPEC

General Chairs:

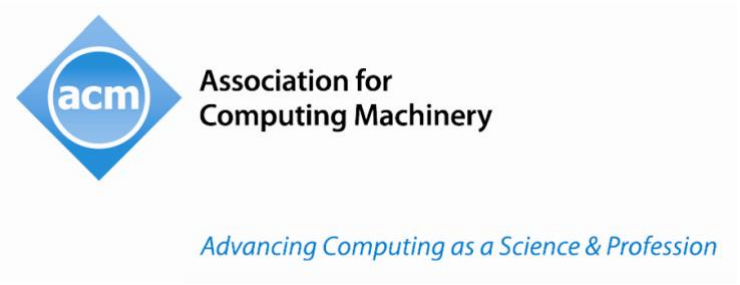
Simonetta Balsamo (Ca' Foscari University of Venice, Italy)
William Knottenbelt (Imperial College London, UK)

Program Chairs:

Cristina L. Abad (Escuela Superior Politecnica del Litoral, Ecuador)
Weiyi Shang (University of Waterloo, Canada)

Publications Chairs:

Mauro Iacono (Università degli Studi della Campania Luigi Vanvitelli, Italy)
Jianing Qiu (Imperial College London, UK)



The Association for Computing Machinery
1601 Broadway, 10th Floor
New York, NY 10019-7434

Copyright © 2024 by the Association for Computing Machinery, Inc. (ACM).
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: permissions@acm.org or Fax +1 (212) 869-0481.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through www.copyright.com.

ISBN: 979-8-4007-0444-4

Additional copies may be ordered prepaid from:

ACM Order Department
PO Box 30777
New York, NY 10087-0777, USA

Phone: 1-800-342-6626 (USA and Canada)
+1-212-626-0500 (Global)
Fax: +1-212-944-1318
E-mail: acmhelp@acm.org
Hours of Operation: 8:30 am – 4:30 pm ET

Printed in the USA

ICPE 2024 Chairs' Welcome

Years of planning have gone into preparing for ICPE 2024 in London, UK. For the first time in UK, the organization of ICPE has generated a great deal of excitement and expectation of productive interactions between the usual participants of ICPE conferences, the members of the various SPEC working groups, and a desire to increase the involvement of the local scientific community with ICPE.

It is our pleasure to welcome you to the 15th ACM/SPEC International Conference on Performance Engineering (ICPE), hosted at South Kensington, London, UK, from May 7-11, 2024. ICPE is the leading international forum for presenting and discussing novel ideas, innovations, trends and experiences in the field of performance engineering.

ICPE formed from merging the ACM Workshop on Software Performance (WOSP, since 1998) and the SPEC International Performance Engineering Workshop (SIPEW, since 2008). Despite the peculiar time we are all living in around the world, we are pleased to introduce an exciting program, which is the result of hard work by the authors, the program committee, and the conference organizers.

We received high quality submissions across the Research, Industry, Emerging Research, Artifact, Data Challenge, Poster and Demo. Each submission received at least two to three reviews from program committee members. Through a rigorous review process, including a discussion period, we selected the most meritorious papers to create a diverse and interesting conference program. For the first time in ICPE, we accept Journal-first submissions that encourage published journal papers to be presented in ICPE in order to further enrich our program.

In the research track, 18 out of 52 submissions were accepted as full papers, yielding a full paper acceptance ratio of 34.6%. 12 full papers received an ACM artifact badge after the subsequent review process in the artifact evaluation track.

In the industry track, 8 full papers have been accepted out of 13 submissions. One paper received an an ACM artifact badge in the artifact evaluation track.

In the emerging research track, we also accepted 7 out of 13 papers (53.8%), where 2 papers were acted as vision papers and 5 papers were accepted as work-in-progress papers.

We also selected candidates for the best paper awards, with the awards to be announced during the conference events.

In addition to the paper sessions, we are glad to welcome the keynote speakers. Manzoor Mohammed, from Capacitas, who plans to share his experience in software performance engineering in the cloud; Jane Hillston, from the University of Edinburgh, who plan to share her experience on performance in the era of large language models; and Giuliano Casale, from Imperial College London who plans to share his research on optimizing performance on Edge AI.

The technical program also includes a poster and demo track, 7 workshops and 3 tutorials.

We thank all the authors who submitted their innovative work to ICPE this year. In addition, we thank all the program committee members and external reviewers for volunteering their time for the benefit of the ICPE community and their hard work in providing quality reviews. Finally, we thank all ICPE participants as we rely on you to make this event interactive, engaging, and thought-provoking for everyone involved.

Simonetta Balasamo
ICPE'24 General Co-Chair
Ca'Foscari University of Venice, Italy

William Knottenbelt
ICPE'24 Genreal Co-Chair
Imperial College London, UK

Chairs continued.

Cristina L. Abad
ICPE'24 Program Co-Chair
Escuela Superior Politecnica del Litoral, Ecuador

Alexander Podelko
ICPE'24 Industry Chair
Amazon/AWS, USA

Lishan Yang
ICPE'24 Emerging Research Co-Chair
George Mason University, USA

Weiyi Shang
ICPE'24 Program Co-Chair
University of Waterloo, Canada

Vittoria de Nitto Personè
ICPE'24 Emerging Research Co-Chair
Tor Vergata University of Rome, Italy

Table of Contents

ICPE 2024: 15th ACM/SPEC International Conference on Performance Engineering Organization viii

ICPE 2024 Sponsors & Supporters xii

Keynote Talk 1

- **How the Cloud made Performance Appear on the Board Agenda** 1
  Manzoor Mohammed *(Capacitas Ltd)*

Session 1a: Benchmarking

- **ShuffleBench: A Benchmark for Large-Scale Data Shuffling Operations with Distributed Stream Processing Frameworks** 2
  Sören Henning *(JKU/Dynatrace Co-Innovation Lab, Johannes Kepler University Linz)*,
  Adriano Vogel *(JKU/Dynatrace Co-Innovation Lab, Johannes Kepler University Linz)*,
  Michael Leichtfried *(Dynatrace Research, Dynatrace LLC)*,
  Otmar Ertl *(Dynatrace Research, Dynatrace LLC)*,
  Rick Rabiser *(LIT CPS Lab, Johannes Kepler University Linz)*

Session 1b: Programming Langauge and Software

- **Vectorized Intrinsics Can Be Replaced with Pure Java Code without Impairing Steady-State Performance** 14
  Júnior Löff *(Faculty of Informatics, Università della Svizzera italiana (USI))*,
  Filippo Schiavio *(Faculty of Informatics, Università della Svizzera italiana (USI))*,
  Andrea Rosà *(Faculty of Informatics, Università della Svizzera italiana (USI))*,
  Matteo Basso *(Faculty of Informatics, Università della Svizzera italiana (USI))*,
  Walter Binder *(Faculty of Informatics, Università della Svizzera italiana (USI))*
- **Rethinking 'Complement' Recommendations at Scale with SIMD** 25
  Shrey Pandey *(Data Science, Myntra Designs Pvt. Ltd.)*,
  Saikat Kumar Das *(Data Science, Myntra Designs Pvt. Ltd.)*,
  Hrishikesh V. Ganu *(Independent)*, Satyajeet Singh *(Data Science, Myntra Designs Pvt. Ltd.)*
- **An Adaptive Logging System (ALS): Enhancing Software Logging with Reinforcement Learning Techniques** 37
  Amirmahdi Khosravi Tabrizi *(Brock University)*, Naser Ezzati-Jivan *(Brock University)*,
  Francois Tetreault *(Ciena)*
- **Time Series Forecasting of Runtime Software Metrics: An Empirical Study** 48
  Federico Di Menna *(University of L'Aquila)*, Luca Traini *(University of L'Aquila)*,
  Vittorio Cortellessa *(University of L'Aquila)*
- **An Empirical Analysis of Common OCI Runtimes' Performance Isolation Capabilities** 60
  Simon Volpert *(Institute of Information Resource Management, Ulm University)*,
  Sascha Winkelhofer *(Gini GmbH)*, Stefan Wesner *(University of Cologne)*,
  Jörg Domaschka *(BenchANT GmbH)*

Session 2a: Hardware

- **An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory** 71
  Lukas Alt *(RWTH Aachen University)*, Anara Kozhokanova *(IT Center, RWTH Aachen University)*,
  Thomas Ilsche *(CIDS, Information Services and High Performance Computing (ZIH), TU Dresden)*,
  Christian Terboven *(IT Center, RWTH Aachen University)*,
  Matthias S. Mueller *(IT Center, RWTH Aachen University)*
- **Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays** 83
  Stephen Nicholas Swatman *(University of Amsterdam & CERN)*, Ana-Lucia Varbanescu *(University of Twente)*,
  Andy D. Pimentel *(University of Amsterdam)*, Andreas Salzburger *(CERN)*, Attila Krasznahorkay *(CERN)*

- **Energy Efficiency Features of the Intel Alder Lake Architecture** 95
  Robert Schöne *(CIDS, Information Services and High Performance Computing (ZIH), TU Dresden)*,
  Markus Velten *(CIDS, Information Services and High Performance Computing (ZIH), TU Dresden)*,
  Daniel Hackenberg *(CIDS, Information Services and High Performance Computing (ZIH), TU Dresden)*,
  Thomas Ilsche *(CIDS, Information Services and High Performance Computing (ZIH), TU Dresden)*
- **Developing Index Structures in Persistent Memory Using Spot-on Optimizations with DRAM** 107
  Xingsheng Zhao *(University of Texas at Arlington)*, Prajwal Challa *(University of Texas at Arlington)*,
  Chen Zhong *(University of Texas at Arlington)*, Song Jiang *(University of Texas at Arlington)*

Keynote Talk 2

- **What does Performance Mean for Large Language Models?** 118
  Jane Hillston *(University of Edinburgh)*

Session 3a: Micro-services 1

- **InstantOps: A Joint Approach to System Failure Prediction and Root Cause Identification in Microserivces Cloud-Native Applications** 119
  Raphael Rouf *(York University)*, Mohammadreza Rasolroveicy *(IBM Canada Lab)*,
  Marin Litoiu *(York University)*, Seema Nagar *(IBM Research)*, Prateeti Mohapatra *(IBM Research)*,
  Pranjal Gupta *(IBM Research)*, Ian Watts *(IBM Canada Lab)*

Session 3b: Resource Management

- **Daedalus: Self-Adaptive Horizontal Autoscaling for Resource Efficiency of Distributed Stream Processing Systems** 130
  Benjamin J. J. Pfister *(Technische Universität Berlin)*, Dominik Scheinert *(Technische Universität Berlin)*,
  Morgan K. Geldenhuys *(Technische Universität Berlin)*, Odej Kao *(Technische Universität Berlin)*
- **Demeter: Resource-Efficient Distributed Stream Processing under Dynamic Loads with Multi-Configuration Optimization** 142
  Morgan K. Geldenhuys *(Technische Universität Berlin)*, Dominik Scheinert *(Technische Universität Berlin)*,
  Odej Kao *(Technische Universität Berlin)*, Lauritz Thamsen *(University of Glasgow)*
- **BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era** 154
  Zebin Ren *(Vrije Universiteit Amsterdam)*, Krijn Doekemeijer *(Vrije Universiteit Amsterdam)*,
  Nick Tehrany *(BlueOne Business Software LLC & Vrije Universiteit Amsterdam)*,
  Animesh Trivedi *(Vrije Universiteit Amsterdam)*
- **The Cost of Simplicity: Understanding Datacenter Scheduler Programming Abstractions** 166
  Aratz Manterola Lasa *(WarpStream Labs)*, Sacheendra Talluri *(Vrije Universiteit Amsterdam)*,
  Tiziano De Matteis *(Vrije Universiteit Amsterdam)*, Alexandru Iosup *(Vrije Universiteit Amsterdam)*

Session 4a: GPU and AI

- **Accelerating ML Workloads using GPU Tensor Cores: The Good, the Bad, and the Ugly** 178
  Bagus Hanindhito *(The University of Texas at Austin)*, Lizy K. John *(The University of Texas at Austin)*
- **MalleTrain: Deep Neural Networks Training on Unfillable Supercomputer Nodes** 190
  Xiaolong Ma *(University of Nevada, Reno)*, Feng Yan *(University of Houston)*,
  Lei Yang *(University of Nevada, Reno)*, Ian Foster *(Argonne National Laboratory & University of Chicago)*,
  Michael E. Papka *(Argonne National Laboratory & University of Illinois Chicago)*,
  Zhengchun Liu *(Argonne National Laboratory)*,
  Rajkumar Kettimuthu *(Argonne National Laboratory & University of Chicago)*
- **Leftovers for LLaMA** 201
  Ravi Kumar Singh *(TCS Research)*, Likhith Bandamudi *(TCS Research)*,
  Shruti Kunde *(TCS Research)*, Mayank Mishra *(TCS Research)*, Rekha Singhal *(TCS Research)*
- **Processing Natural Language on Embedded Devices: How Well Do Modern Models Perform?** 211
  Souvika Sarkar *(Auburn University)*, Mohammad Fakhruddin Babar *(Washington State University)*,
  Md Mahadi Hassan *(Auburn University)*, Monowar Hasan *(Washington State University)*,
  Shubhra Kanti Karmaker Santu *(Auburn University)*

Keynote Talk 3

- **Optimizing Edge AI: Performance Engineering in Resource-Constrained Environments** 223
  Giuliano Casale *(Department of Computing, Imperial College London)*

Session 5a: Analysis

- **TBASCEM - Tight Bounds with Arrival and Service Curve Estimation by Measurements** 224
  Christoph Funda *(Test System Development, ZF Mobility Solutions GmbH)*,
  Thomas Herpel *(System Qualification & Product Test Strategy, ZF Mobility Solutions GmbH)*,
  Reinhard German *(Computer Networks and Communication Systems, Friedrich-Alexander-Universität Erlangen-Nürnberg)*,
  Kai-Steffen Jens Hielscher *(Computer Networks and Communication Systems, Friedrich-Alexander-Universität Erlangen-Nürnberg)*

Session 5b: Edge

- **A Learning-Based Caching Mechanism for Edge Content Delivery** 236
  Hoda Torabi *(Department of Electrical Engineering and Computer Science, York University)*,
  Hamzeh Khazaei *(Department of Electrical Engineering and Computer Science, York University)*,
  Marin Litoiu *(Department of Electrical Engineering and Computer Science, York University)*
- **Function Offloading and Data Migration for Stateful Serverless Edge Computing** 247
  Matteo Nardelli *(Bank of Italy)*, Gabriele Russo Russo *(Tor Vergata University of Rome)*
- **No Clash on Cache: Observations from a Multi-tenant Ecommerce Platform** 258
  Anna Lira *(Federal University of Campina Grande)*, Ruan Alves *(Federal University of Campina Grande)*,
  Thiago Emmanuel Pereira *(Federal University of Campina Grande)*,
  Fabio Morais *(Federal University of Campina Grande)*, João Ramalho *(Federal University of Campina Grande)*,
  Mariana Mendes *(VTEX)*
- **MemSaver: Enabling an All-in-memory Switch Experience for Many Apps in a Smartphone** 267
  Prajwal Challa *(University of Texas at Arlington)*, Baohua Song *(Coredump Limited)*,
  Song Jiang *(University of Texas, Arlington)*

Session 6: Micro-services 2

- **Systemizing and Mitigating Topological Inconsistencies in Alibaba's Microservice Call-graph Datasets** 276
  Darby Huye *(Tufts University)*, Lan Liu *(Tufts University)*, Raja R. Sambasivan *(Tufts University)*
- **Disambiguating Performance Anomalies from Workload Changes in Cloud-Native Applications** 286
  Alexandru Baluta *(York University)*, Yar Rouf *(York University)*,
  Joydeep Mukherjee *(California Polytechnic State University)*,
  Zhen Ming Jiang *(York University)*, Marin Litoiu *(York University)*

Author Index 298

ICPE 2024: 15th ACM/SPEC International Conference on Performance Engineering Organization

General Chairs: Simonetta Balsamo (Ca' Foscari University of Venice, Italy)
William Knottenbelt (Imperial College London, UK)

Program Chairs: Cristina L. Abad (Escuela Superior Politecnica del Litoral, Ecuador)
Weiyi Shang (University of Waterloo, Canada)

Journal-First Chairs: Cristina L. Abad (Escuela Superior Politecnica del Litoral, Ecuador)
Weiyi Shang (University of Waterloo, Canada)

Industry Track Chair: Alexander Podelko (Amazon/AWS, USA)

Emerging Research Track Chairs: Vittoria de Nitto Personè (Tor Vergata University of Rome, Italy)
Lishan Yang (George Mason University, USA)

Artifact Evaluation Chairs: Robert Ricci (University of Utah, USA)
Dmitry Duplyakin (NREL, USA)

Workshop Chairs: Diego Costa (Concordia University, Canada)
Michele Tucci (University of L'Aquila, Italy)

Tutorial Chair: Heng Li (Polytechnique Montreal, Canada)

Poster and Demos Chairs: André Bauer (University of Chicago, USA)
Martin Straesser (University of Würzburg, Germany)

Data Challenge Chairs: Luca Traini (University of L'Aquila, Italy)
Christoph Laaber (Simula Research Laboratory, Oslo, Norway)

Award Chairs: Katinka Wolter (Freie Universitaet zu Berlin, Germany)
Marin Litoiu (York University, Canada)

Publicity & Social-Media Chair: Marco Paolieri (University of Southern California, USA)

Finance Chair: Tom Curtin (Imperial College London, UK)

Publications Chairs: Mauro Iacono (Università degli Studi della Campania Luigi Vanvitelli, Italy)
Jianing Qiu (Imperial College London, UK)

Web Chair: Giordano d'Aloisio (University of L'Aquila, Italy)

Program Committee: Antinisca Di Marco (University of L'Aquila)
Varsha Apte (Indian Institute of Technology – Bombay)
Yiming Tang (Rochester Institute of Technology)
Alexandru Iosup (VU)
Federica Sarro (University College London)
Murray Woodside (Carleton University)
Josef Spillner (Zurich University of Applied Sciences)
Philipp Leitner (Chalmers | University of Gothenburg)
Tingting Yu (University of Cincinnati)
Nikolas Herbst (University of Würzburg)
Cor-Paul Bezemer (University of Alberta)
Catia Trubiani (Gran Sasso Science Institute)
André van Hoorn (University of Hamburg)
Samuel Kounev (University of Wuerzburg)
Andre Bondi (Software Performance and Scalability Consulting LLC)
José Merseguer (Universidad de Zaragoza)
Valeria Cardellini (University of Roma "Tor Vergata")
Andrea Marin (Università Ca' Foscari Venezia)
Maria Carla Calzarossa (Universita di Pavia)
Raffaela Mirandola (Politecnico di Milano)
Steffen Becker (University of Stuttgart)
Marco Vieira (University of Coimbra)
Yintong Huo (Hong Kong University of Science and Technology)
Mauro Iacono (Università degli Studi della Campania "Luigi Vanvitelli")
Lishan Yang (George Mason University)
Alberto Avritzer (EsulabSolutions Inc.)
Wes Lloyd (University of Washington)
Junwen Yang (University of Chicago)
Petr Tuma (Charles University)
Patrick P. C. Lee (The Chinese University of Hong Kong)
Catalina M. Lladó (Universitat Illes Balears)
Sen He (The University of Arizona)
Vittorio Cortellessa (Università dell'Aquila)
Connie Smith (Performance Engineering Services)
Katinka Wolter (Freie Universitaet zu Berlin)
Jianmei Guo (East China Normal University)
Anne Koziolek (Karlsruhe Institute of Technology)
Wilhelm Hasselbring (Kiel University)
Evgenia Smirni (College of William and Mary)
Aris Leivadeas (École de technologie supérieure)
Shaohua Wang (Central University of Finance and Economics)
Daniele Di Pompeo (University of L'Aquila)

Program Committee (continued): Manoj Nambiar (Tata Consultancy Services)
Heng Li (Polytechnique Montréal)
André Bauer (University of Chicago)
Tse-Hsun (Peter) Chen (Concordia University)

Program Committee (Industry track): Muhammad Shoaib Bin Altaf (Oracle)
Vlastimil Babka (SUSE)
David Daly (MongoDB)
François Farquet (Oracle Labs)
Matt Flemming (Datastax)
Ajay Joshi (ARM)
Klaus-Dieter Lange (HPE)
Sai Sindhur Malleni (Red Hat)
Anoush Najarian (MathWorks)
Nishant Rawtani (HPE)
Daniel Seybold (benchANT)
Rekha Singhal (TCS)
Igor Trubin (Capital One)
Alexander Wert (Elastic)

Program Committee (Artifact) Aarushi Jain (University of Texas Arlington)
Aleksandra Kowalczuk (University of Warsaw)
Amit Samanta (University of Utah)
Elizabeth Ondula (University of Southern California)
Ghadeer Almusaddar (Binghamton University)
Hongyu Hè (ETH Zurich)
Jaiaid Mobin (Rochester Institute of Technology)
Juno Suárez (Portland State University)
Kevin Assogba (Rochester Institute of Technology)
Matthew Forshaw (Newcastle University)
Mazahir Hussain (Korea Institute of Science and Technology Information)
Muhammed Emin Ozturk (University of Utah)
Pegah Ahadian (Kent State University)
Rafael Herrera (University of Delaware)
Ryan Scherbarth (University of New Mexico)
Saheed Olayemi Bolarinwa (The Leibniz Supercomputing Centre)
Sören Henning (Johannes Kepler University Linz)
Stephen Nicholas Swatman (University of Amsterdam)
Swetha Varadarajan
Tamoghna Sarkar (University of Southern California)
Souptik Sen (Snowflake)
Urjoshi Sinha (Lawrence Berkeley National Lab)
Radhakrishnan Venkataramani (Snowflake)

Program Committee (Data Challenge):
Jinfu Chen (Wuhan University)
Zishuo Ding (University of Waterloo)
Martin Grambow (Technische Universität Berlin)
Sen He (University of Arizona)
Lizhi Liao (University of Waterloo)
Max Weber (Leipzig University)
Chenxi Zhang (Fudan University)

Additional Reviewers:
Andres Abad
Palak Bhandari
Prateek Bhatnagar
Andrea Bianchi
Vanessa Borst
Sophie Corallo
Robert Cordingly
Timo Dittus
Daniel Grillmeyer
Marius Hadry
Zhihan Jiang
Supriya Kamthania
Shruti Kunde
Yichen Li
Chuansheng Lu
Yannik Lubas
Maximilian Meissner
Diletta Olliaro
Kebin Peng
Ivo Rohwer
Larissa Schmid
Seyedehhaleh Seyeddizaji
Shuai Shao
Anushree Singh
Michael Stenger
Rajesh Tadakamadla
Chen Zou

ACM/SPEC ICPE 2024 Sponsors & Supporters

Sponsors:

Supporters:

How the Cloud made Performance Appear on the Board Agenda

Manzoor Mohammed
Capacitas Ltd, London, United Kingdom
manzoormohammed@capacitas.co.uk

ABSTRACT

Cloud spending is growing! Gartner predicts a 20% surge to $678.8 billion in 2024, making it a top expense after personnel for many organisations. In fact, 78% of US businesses and 54% in EMEA already leverage the cloud for diverse needs, from infrastructure and storage to development. However, a crucial question lingers: are we maximising our cloud investments? This year, boards are wanting to know the answer this question as investments continue to increase due to new technologies such as LLM. Thankfully, by delving deeper into cloud performance characteristics, we can unlock valuable insights. This keynote will explore how understanding performance empowers organisations to extract the full potential of their cloud, transforming cost and performance data into strategic advantage.

CCS Concepts

•Computer systems organization~Cloud computing

Author Keywords

cloud optimization, cloud performance, cloud cost

ACM Reference format:

Manzoor Mohammed. 2024. How the Cloud made Performance Appear on the Board Agenda. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 1 page. https://doi.org/10.1145/3629526.3645056

BIOGRAPHY

Manzoor, a cloud veteran with over 25 years of experience, co-founded Capacitas and serves as its Chief Innovation Officer. He has a proven track record of delivering millions in annual cloud spend reductions and optimising the performance of complex systems and has helped companies achieve their financial goals through strategic cloud management. Manzoor recognised the transformative power of cloud computing early on, understanding how it strengthens the link between performance and cost. This realisation led him to create Capacitas' unique methodology, which has empowered numerous clients, including easyJet, Skype, JAGGAER, Ancestry, Cegid, and BMC Software, to achieve significant cost reductions and performance improvements. Manzoor is a trusted advisor and thought leader in the cloud computing space, passionate about helping businesses leverage the cloud to achieve their full potential.

Acknowledgements

The keynote is based on our clients continued support and support of our the principal that delivering better performance means lower costs. A list can be found on our website.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author(s).
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright is held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645056

ShuffleBench: A Benchmark for Large-Scale Data Shuffling Operations with Distributed Stream Processing Frameworks

Sören Henning
JKU/Dynatrace Co-Innovation Lab,
Johannes Kepler University Linz
Linz, Austria
soeren.henning@jku.at

Adriano Vogel
JKU/Dynatrace Co-Innovation Lab,
Johannes Kepler University Linz
Linz, Austria
adriano.vogel@jku.at

Michael Leichtfried
Dynatrace Research,
Dynatrace LLC
Linz, Austria
michael.leichtfried@dynatrace.com

Otmar Ertl
Dynatrace Research,
Dynatrace LLC
Linz, Austria
otmar.ertl@dynatrace.com

Rick Rabiser
LIT CPS Lab,
Johannes Kepler University Linz
Linz, Austria
rick.rabiser@jku.at

ABSTRACT

Distributed stream processing frameworks help building scalable and reliable applications that perform transformations and aggregations on continuous data streams. This paper introduces *ShuffleBench*, a novel benchmark to evaluate the performance of modern stream processing frameworks. In contrast to other benchmarks, it focuses on use cases where stream processing frameworks are mainly employed for *shuffling* (i.e., re-distributing) data records to perform state-local aggregations, while the actual aggregation logic is considered as black-box software components. ShuffleBench is inspired by requirements for near real-time analytics of a large cloud observability platform and takes up benchmarking metrics and methods for latency, throughput, and scalability established in the performance engineering research community. Although inspired by a real-world observability use case, it is highly configurable to allow domain-independent evaluations. ShuffleBench comes as a ready-to-use open-source software utilizing existing Kubernetes tooling and providing implementations for four state-of-the-art frameworks. Therefore, we expect ShuffleBench to be a valuable contribution to both industrial practitioners building stream processing applications and researchers working on new stream processing approaches. We complement this paper with an experimental performance evaluation that employs ShuffleBench with various configurations on Flink, Hazelcast, Kafka Streams, and Spark in a cloud-native environment. Our results show that Flink achieves the highest throughput while Hazelcast processes data streams with the lowest latency.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Computer systems organization** → **Cloud computing**; • **Information systems** → **Stream management**.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645036

KEYWORDS

benchmarking, data shuffling, performance, stream processing

ACM Reference Format:
Sören Henning, Adriano Vogel, Michael Leichtfried, Otmar Ertl, and Rick Rabiser. 2024. ShuffleBench: A Benchmark for Large-Scale Data Shuffling Operations with Distributed Stream Processing Frameworks. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645036

1 INTRODUCTION

State-of-the-art distributed stream processing frameworks such as Spark [4, 39], Flink [6], Kafka Streams [31, 38], or Hazelcast with its Jet engine [10] have gained widespread adoption over the last years not only for building data analytics pipelines, but also for implementing core business logic in software-based organizations [9, 22]. Such frameworks support software engineers in building highly scalable, reliable, and efficient applications that process continuous data streams of massive volume. They provide high-level APIs and domain-specific languages to define the processing logic as directed acyclic processing graphs that filter, transform, aggregate, and merge data streams.

Over the last decade, several works have been published that evaluate the performance of distributed stream processing frameworks or propose new benchmarks, evaluation methods, and auxiliary tools [5, 13, 18, 21, 34, 37]. Available benchmarks usually contain one or a few task samples that fulfill domain-specific use cases, for example, for analyzing car traffic data [3, 34] or aggregating Industrial Internet of Things (IIoT) sensor measurements [13, 18]. To implement these task samples (also referred to as queries), the benchmarks define the domain-specific processing logic with the frameworks' high-level APIs.

In this paper, we study a different, more general use case, where the actual domain-specific processing logic is out of the scope of a stream processing framework, but stream processing frameworks are still used for their abstractions for cluster management, means to scale out the data processing, fault-tolerant state management, well-defined processing guarantees (e.g., exactly-once or at-least-once), and rich ecosystem of documentation, support, and associated tooling. We illustrate this use case inspired by requirements of a large

cloud observability platform, where potentially thousands or millions of stateful black-box software components have to receive and process selected data records. The literature is currently missing a well-defined evaluation method for this use case and previous work found that the performance of stream processing frameworks highly depends on the use case [16].

Therefore, we propose *ShuffleBench* as a new stream processing benchmark focusing on the use of stream processing frameworks for *shuffling* (i.e., re-partitioning) data streams to efficiently process data in high numbers of stateful components. ShuffleBench provides well-defined metrics, measurement methods, and a highly configurable task sample. Thus, it allows researchers and practitioners to evaluate stream processing frameworks with respect to performance attributes such as throughput, latency, and scalability and regarding a representative use case. Our research design combines requirements identified from a large cloud observability platform with established benchmarking metrics, methods, and techniques from the performance engineering research community.

Contributions. In summary, we make the following contributions with this paper to industry practitioners and the research community:

- We propose ShuffleBench, a new benchmark for distributed stream processing frameworks. It addresses a use case not covered by existing benchmarks and is highly configurable for different assorted characteristics.
- We provide open-source implementations[1] of ShuffleBench for different state-of-the-art stream processing frameworks as well as associated tooling to automate running benchmarks in Kubernetes-based cloud environments.
- We conduct an experimental evaluation covering throughput and latency that are highly relevant metrics for stream processing [37]. Our evaluation covers the open-source stream processing frameworks Flink, Hazelcast, Kafka Streams, and Spark Structured Streaming due to their industry acceptance and academic relevance [37]. We provide a replication package and the collected data of our experiments as supplemental material [17], such that other researchers and practitioners may repeat and extend our work.

Outline. The remainder of this paper is structured as follows: Section 2 introduces the fundamental concepts of distributed stream processing frameworks and related benchmarking studies. Section 3 describes the use of stream processing for large-scale data shuffling illustrated by industrial requirements. Afterward, Section 4 presents ShuffleBench, our proposal for a new stream processing benchmark. Section 5 employs ShuffleBench to evaluate four stream processing frameworks regarding their throughput and latency for different configurations. Section 6 concludes this work and discusses future work.

2 BACKGROUND AND RELATED WORK

In the following, we briefly introduce the fundamental concepts of modern stream processing frameworks and discuss related work on benchmarking stream processing frameworks.

[1] https://github.com/dynatrace-research/ShuffleBench

2.1 Distributed Stream Processing

Stream processing frameworks perform operations such as filterings, transformations, or aggregations in near-real time on continuous streams of data [19]. State-of-the-art frameworks are designed for high throughput and low-latency processing, while also scaling with massive amounts of data [9, 16]. To address these requirements, they run in a distributed fashion on commodity hardware. A key advantage of stream processing frameworks is that they provide dataflow models that abstract aspects such as cluster management, state management, and time semantics from their users [2, 31]. With such models, engineers describe the processing logic in directed acyclic dataflow graphs of processing operators. The frameworks allow the initiation of multiple instances across various compute nodes, containers, or with multiple threads, with each instance handling a distinct portion of the data. While the isolated processing of data records remains unaffected by the assignment of data portions to instances, processing that depends on previous data records, such as aggregations, requires state management. Similar to the MapReduce [8] programming model, keys are assigned to records before a stateful operation. This allows the stream processing frameworks routing all records with the same key to the same instance. Consequently, no state synchronization among instances is required, which allows operating stream processing applications at significantly lower costs compared to, for example, Functions-as-a-Service [29]. When a processing operator modifies the key of a record and a subsequent operator performs a stateful operation, the framework divides the dataflow graph into subgraphs that can be independently processed by different instances.

Popular stream processing frameworks include Apache Flink [6], Hazelcast with its Jet engine [10], Apache Kafka Streams [31, 38], and Apache Spark [39] with its Structured Streaming engine [4]. Although all these frameworks follow similar concepts, several differences in their design decisions, programming functionalities, and execution models can be noted. Whereas Flink and Spark follow a master-worker cluster architecture, Hazelcast and Kafka Streams instances form clusters and perform the necessary coordination internally. Moreover, Hazelcast and Kafka Streams can be embedded as library into applications, while Flink and Spark are the applications themselves. Spark is different from the other frameworks as it processes the data streams as a sequence of batches (called micro-batches), whereas the other frameworks process record by record as they arrive. Hazelcast's Jet engine is special by being built on top of the Hazelcast IMDG distributed, in-memory object store. It differs in its execution model, which is based on a concept similar to coroutines and cooperative threads to process data at very low latency [10]. Kafka Streams tightly integrates with the Apache Kafka messaging system [24]. It does not support other data sources and sinks than Kafka, but benefits from lightweight coordination based on Kafka's consumer group protocol.

Stream processing frameworks often read data from or write data to messaging systems. Such messaging systems serve both as a scalable middleware between different systems and services as well as the necessary infrastructure to ensure fault tolerance. For this purpose, industry-grade messaging systems such as Apache Kafka [24, 38] employ an immutable, sequentially appended log structure to store and replicate records across distributed nodes.

Table 1: Overview of related open-source stream processing benchmarks and associated benchmarking studies

	DSPBench [5]	OSPBench [34–36]	ESPBench [18]	Theodolite [13, 16]	**ShuffleBench (This work)**
Benchmark inspiration	From the literature	Car traffic	IIoT	IIoT	Cloud observability
Application metrics	Throughput, latency	Throughput, latency	Latency	Scalability	Throughput, latency
Frameworks	Spark, Storm	Flink, Kafka Streams, Spark, Spark Structured Streaming	Beam (Flink, Hazelcast, Spark)	Beam, Flink, Hazelcast, Kafka Streams	Flink, Hazelcast, Kafka Streams, Spark Structured Streaming
Deployment	Virtual machines	Containers, DC/OS	Virtual machines	Containers, Kubernetes	Containers, Kubernetes
Data size	1 to 60 KB	approx. 200 bytes	up to 280 bytes	up to 100 bytes	configurable
Processed record/second	up to 100k	up to 900k	up to 10k	up to one million	up to one million
Customizable state size	No	No	No	No	Yes

2.2 Benchmarking Stream Processing Frameworks

A considerable number of studies proposing new benchmarks or reporting on experimental evaluations with available benchmarks have been published over the last years [37]. Recent examples of open-source benchmarks include DSPBench [5], OSPBench [34], ESPBench [18], or the streaming benchmarks from the Theodolite framework [13]. Table 1 provided an overview of these benchmarks and a comparison with ShuffleBench. For a systematic and comprehensive review of the literature on stream processing benchmarking, we refer to our recent studies [16, 37].

The most important difference between other benchmarks and ShuffleBench is the addressed use case. Most benchmarks re-sample domain-specific analytics applications, for example, for IIoT data streams [13, 18], car traffic data [34], or online gaming [21]. Our paper in contrast investigates a more generic use case, which addresses a specific type of software architecture instead of a specific domain.

Despite the addressed use case, we found that benchmarks proposed by the research community are often not applicable for industry-grade performance evaluations. They are often not available as open-source software or not actively maintained, lack automation, and do not use deployments that are representative for production systems. For example, although it is nowadays common to run stream processing applications in containerized cloud-native environments, there is only one benchmark besides ShuffleBench specifically tailored to such deployments [13]. Likewise, results of performance evaluations conducted with these benchmarks are often not transferable to industry use cases, for example, because they evaluate frameworks only at small-scale deployments [5]. Other benchmarks or evaluations do not define metrics and measurement methods, which makes the research difficult to reproduce and extend.

Nevertheless, the literature provides valuable research on evaluation metrics and measurement methods [12, 21, 34]. We build upon such research to provide well-defined metrics and measurement methods with ShuffleBench.

3 LARGE-SCALE DATA SHUFFLING

We illustrate the use case of large-scale data shuffling with requirements for continuous dashboard queries and real-time alerting of a market-leading cloud observability platform. Using a powerful query language, it allows internal and external clients to define complex rules to aggregate and correlate different data sources such as metrics, events, logs, and traces. From a software architecture perspective, each registered query can be considered as a runtime software component, which continuously receives all data records that are affected by this query. We call these components *real-time consumers*. They have to manage state across multiple input records and might produce outputs. For example, a consumer that performs an anomaly detection by correlating logs and performance metrics might produce an output event when it detects an anomaly.

A core requirement for a corresponding query runtime is to efficiently route data to the respective consumers while also having cluster management abstractions, means to scale out the data processing, fault-tolerant state management, and well-defined processing guarantees. State-of-the-art stream processing frameworks fulfill these properties. They also provide programmatic APIs or dedicated SQL-like languages to define complex queries on data streams (see related benchmarks in Section 2). However, those only have limited relevance for our use case as we are facing a high amount of queries that have to be executed in parallel, although each query only requires a very small portion of the overall data volume. Therefore, it is not required to parallelize or even distribute the execution of a single query but only to route those data records to a consumer as needed. Moreover, considering that dedicated software components (e.g., anomaly detection models) can include the logic for operations such as joins or sliding windows, these features might not be required by the stream processing framework.

In addition to the observability use case described here, we expect other real-world systems to have similar requirements. For example, the stateful Function-as-a-Service [1] runtime Apache Flink Stateful Functions[2] is used in different contexts, but based on a very similar architecture as our benchmark (see the following section).

[2]https://nightlies.apache.org/flink/flink-statefun-docs-master/

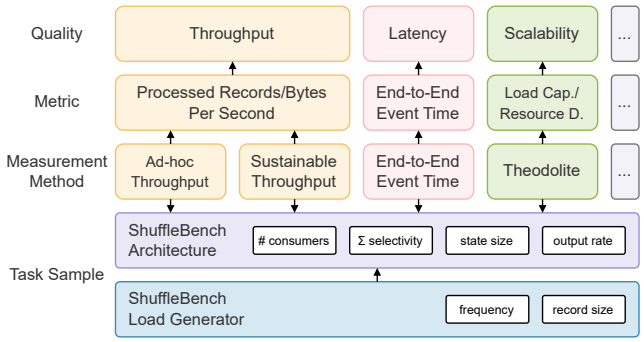


Figure 1: Overview of *ShuffleBench* benchmark components according to the Empirical Standard for benchmarking.

4 THE SHUFFLEBENCH BENCHMARK

We combine the architectural requirements described above with best practices of the performance engineering community and industrial consortia [11, 23, 28, 33].

4.1 Benchmark Design

We base our benchmark design on the ACM SIGSOFT Empirical Standard for benchmarking [11, 30].[3] It distinguishes between the following four components of a benchmark: the qualities to be evaluated, metrics to quantify these qualities, measurement methods for these metrics, and task samples to be evaluated with the measurement methods. As discussed in Section 2, the task samples of existing benchmarks have several shortcomings, making them impractical to study our described use case. On the other hand, several evaluation metrics, methods, and tools introduced with other benchmarks are applicable independently of the specific use case. According to the Empirical Standard, we, therefore, propose a new benchmark, which, on the one hand, introduces a new task sample but, on the other hand, takes up existing metrics and measurement methods that have demonstrated their effectiveness in the literature. This way, we address benchmark quality attributes such as relevance, reproducibility, fairness, verifiability, and usability as required by industrial consortia and the research community [33].

Figure 1 provides an overview of our benchmark's components. In the following, we first describe the task sample in terms of its dataflow architecture to be implemented by different stream processing frameworks and a corresponding load generator. Afterward, we describe the qualities, metrics, and measurement methods to assess the task sample implementations.

4.2 The ShuffleBench Task Sample

The ShuffleBench task sample consists of a dataflow architecture to be implemented by different stream processing frameworks and a load generator to stress the framework. Both are highly configurable to allow for evaluations in different scenarios.

[3] https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=Benchmarking

4.2.1 Benchmark Dataflow Architecture. A well-suited way to meet the described use case is a MapReduce-like architecture [8] on continuous data streams as it can be built with modern stream processing frameworks. Figure 2 depicts our benchmark architecture for a corresponding stream processing application. It represents the long-running application as a static dataflow graph, with data moving along the edges. It can be deployed with multiple instances, which execute the same processing logic but on different data subsets. The data processing starts by reading data records from a messaging system such as Kafka. Kafka topics are partitioned, allowing each instance of the stream processing application to subscribe to a dedicated set of partitions. In general, we cannot assume any specific partitioning on the data. After ingestion, the *matcher service* finds the relevant queries for each record. The matcher logic is wrapped in a *flatMap* operation of a stream processing framework that duplicates an incoming record for each relevant query while assigning a query-identifying key to each duplicate. In a subsequent operation, the data is re-partitioned among all instances such that all records with the same query key are forwarded to the same instance. This is done with an operation of the stream processing framework often called *groupBy*. In the next step, the actual black-box query logic is executed, which is stateful by aggregating multiple records. We abstract the query logic in *real-time consumers*, wrapped in an *aggregate* operation of the stream processing framework that manages the state. These real-time consumers adhere to a simple interface: they consume an incoming record and the previous state and output the updated state and, optionally, an event. Finally, these output events are written to another Kafka topic.

4.2.2 Benchmark Implementations. We implemented the proposed benchmark for the four stream processing frameworks Flink, Hazelcast, Kafka Streams, and Spark Structured Streaming. To provide a fair comparison, the implementation of the matcher service and the real-time consumers (which would be domain-specific in production) are shared among all frameworks. The matcher service is configured with a set of rules, which define their selectivity, i.e., the probability that this rule matches a record. The actual stateful aggregations logic is currently the same for each query: Every incoming record is updates to the state of the respective real-time consumer. The state size is configurable, but always includes the count of received records, a checksum, and the associated timestamps of the first and the last record (see Section 4.3.2). Real-time consumers emit an output event if the count is divisible by a configured value to simulate something like a generated alert.

4.2.3 Load Generator. To simulate incoming observability data, we provide a load generator that creates data records at a configurable frequency with random byte content of configurable size. We decided to use this way of generating data instead of replaying historical data to make performance evaluations more reproducible and ShuffleBench applicable to other domains [33]. The load generator can be deployed in a distributed fashion and writes the generated records to Kafka.

4.2.4 Benchmark Configuration Options. Our ShuffleBench implementations are highly configurable to evaluate frameworks for different use cases of large-scale data shuffling tasks. This includes the size of incoming records, the number of different real-time

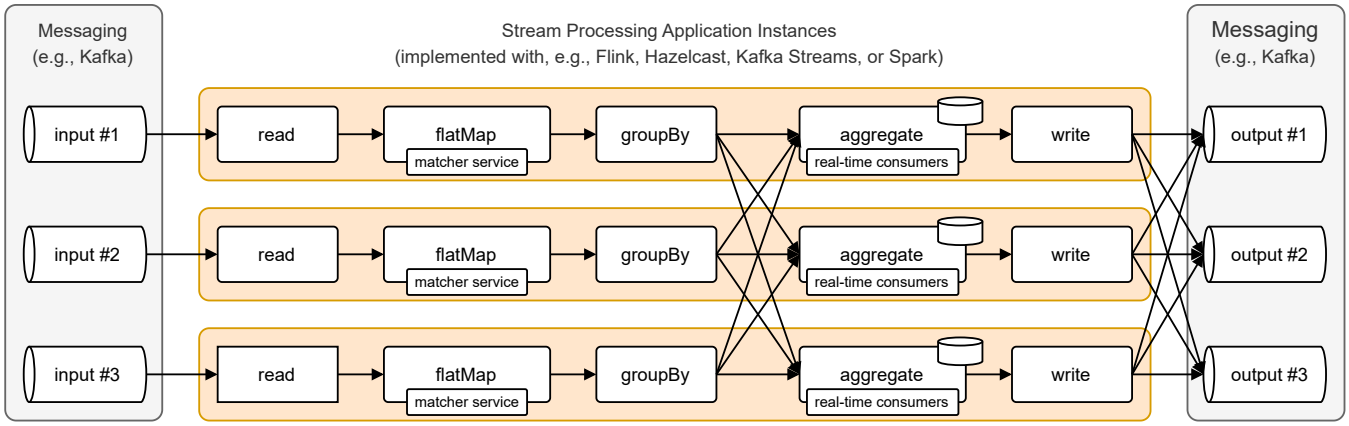


Figure 2: The *ShuffleBench* dataflow architecture at runtime for three stream processing application instances.

consumers, the total selectivity for all real-time consumers, the distribution of individual selectivities, the real-time consumer's state size, and their output frequency. Additionally, all stream processing frameworks have a wide range of configuration and deployment options that potentially impact throughput, latency, scalability, and fault tolerance. Many of them can easily be set with ShuffleBench to support experimental comparisons.

4.3 Qualities, Metrics and Measurement Methods

According to requirements from industry that motivated us to design ShuffleBench, we currently support benchmarking the qualities throughput, latency, and scalability with ShuffleBench. In the following, we describe our employed metrics and measurement methods for all qualities in detail. These descriptions, along with our executable benchmarking software (see Section 4.4) support reproducibility of benchmarking studies conducted with ShuffleBench, such as our evaluation in Section 5.

4.3.1 Throughput. We support measuring the throughput of stream processing frameworks in terms of the number of incoming records that can be processed per second. Note that the throughput in terms of processed bytes per second can be directly derived from that. Assuming validated functional correctness, all frameworks process each record exactly once in the absence of failures, which means that the output throughput is proportional to the input throughput.

Essentially, two measurement methods for throughput can be observed in the literature. We refer to them as *ad-hoc throughput* and *sustainable throughput*. As both have their pros and cons, we support both within ShuffleBench and evaluate them in Section 5.

Both measurement methods have in common that they only monitor the messaging system. This has the significant advantage that the throughput measurements do not influence the execution of the application. In fact, the measurement method is fully independent of the benchmarked framework and, thus, can also be used on arbitrary other task samples, including real-world applications.

Ad-hoc Throughput. Ad-hoc throughput measurements can be performed by generating and sending a constant number of records per second to the messaging system and measuring how much of these can be processed. It has to be ensured that the generated data volume is at least as high as the processing rate of the stream processing framework to not limit it. Several studies appear to have conducted throughput measurements using this approach [20, 26]. In ShuffleBench, ad-hoc throughput is measured by tracking the rate of committed offsets at the messaging system [24].

The key advantage of the ad-hoc throughput method is that it can be performed in a short time. However, the obtained throughput results might not fully reflect the real behavior of the stream processing framework when being subject to that load [12]. For example, optimizations such as batching might allow a system to ingest data at a higher rate when reading from a large backlog compared to when data is ingested as it is generated.

Sustainable Throughput. Sustainable throughput [21] is defined as the maximum load a system can sustain without violating performance goals. Such a performance goal can be a limit on the event latency [21, 34] or the maximum tolerable increase in the number of queued messages [13, 16, 34]. In ShuffleBench, sustainable throughput is measured by running multiple independent experiments, in which the generated load is increased from experiment to experiment and performance goals are evaluated [12]. Per default, we evaluate whether the number of queued messages increases substantially over time. This can again be obtained by tracking available message offsets and consumed message offsets at the messaging system [24]. However, we also allow for using custom (e.g., use case-specific) performance goals.

The sustainable throughput measurement method overcomes the limitations of the ad-hoc throughput method. It better reflects how a tested system would behave in a real-world deployment. On the downside, however, this method can only find a range in which the real achievable throughput lies. Moreover, it has significantly longer execution times compared to the ad-hoc method. Instead of evaluating a system with one constant load rate, experiments have to be executed for different load rates.

4.3.2 Latency. There are several different notions of latency in stream processing [7, 21]. In ShuffleBench, we quantify latency from

a user perspective by measuring how long it takes for an input event to generate an output event. Although most frameworks collect some kind of latency metrics, their measurement methods differ, preventing a fair comparison. Moreover, in contrast to throughput, we cannot measure latency purely in the messaging system since input and output events are not natively linked to each other.

In line with van Dongen and van den Poel [34] and Hesse et al. [18], we measure latency as the time difference between writing a record to the input messaging topic and the time a record is written to the output topic. These timestamps can be retrieved in a framework-independent way as messaging systems such as Kafka assign each record the current timestamp when appending it to the log. To correlate both timestamps, we extract the timestamp of the input record and append it to the record's payload. Depending on the framework, this is either done as part of the *read* operator (see Fig. 2) or in an intermediate stateless *map* operation. In the stateful aggregation step, we use this timestamp and include it in the state. Since the whole state is emitted as part of an output event, the final data written to Kafka contains both the time the input event has been written to Kafka and the time the output event has been written to Kafka. ShuffleBench comes with a *latency exporter* that reads all output events and computes a frequency distribution of their latency. The latency exporter can be scaled to multiple instances to also cope with large output data volume.

4.3.3 Scalability. We adopt the *Theodolite* scalability benchmarking method for stream processing systems in cloud-native environments [12, 14]. It provides two metrics for quantifying scalability: The *resource demand* metric describes how the number of required computing resources evolves with increasing load, whereas the *load capacity* describes how the maximum processable load evolves with increasing computing resources. The Theodolite measurement method extends the sustainable throughput method to a two-dimensional search space. It encompasses search strategies two reduce the number of independent experiments to be executed.

4.4 Open-Source Benchmark Availability

We provide ShuffleBench as free and open-source research software to the community. ShuffleBench utilizes and extends the benchmarking framework Theodolite [14, 15] to automate the benchmark execution in Kubernetes-based cloud environments. This includes the declarative definition of benchmark experiments, automated setup and teardown of all involved software components (i.e., stream processing frameworks, load generator, and the messaging system) as well as the collection of measurement data.

Our benchmark implementations for different stream processing frameworks as well as the load generator and the latency exporter are available as source code, Java archive files, container images, Kubernetes manifests, and Theodolite benchmark manifests. This allows the community to engage at different levels.

5 EXPERIMENTAL EVALUATION

We employ ShuffleBench to experimentally evaluate the performance of Flink (v. 1.17), Hazelcast (v. 5.3), Kafka Streams (v. 3.5), and Spark Structured Streaming (v. 3.4, in the following simply referred to as Spark). This evaluation serves two purposes: First, it compares how different stream processing frameworks compete regarding large-scale data shuffling use cases to assist in selecting the right technology. Second, it demonstrates how ShuffleBench allows researchers and practitioners to conduct their own experiments with different configurations of the benchmark.

After a brief description of our experimental setup (Section 5.1), we conduct a set of experiments. Section 5.2 starts by a baseline evaluation of throughput using both measurement methods presented in Section 4.3.1. Likewise, Section 5.3 reports on our baseline latency evaluation. Afterward, Section 5.4 repeats these experiments with a modified deployment, Section 5.5 with different generated record sizes, Section 5.6 with a different number of real-time consumers, and Section 5.7 with different total selectivities of the matcher service. We discuss the results in the context of related work in Section 5.8 and threats to validity in Section 5.9. We provide a replication package and the collected data of our experiments as supplemental material [17], such that other researchers and practitioners may repeat and extend our work.

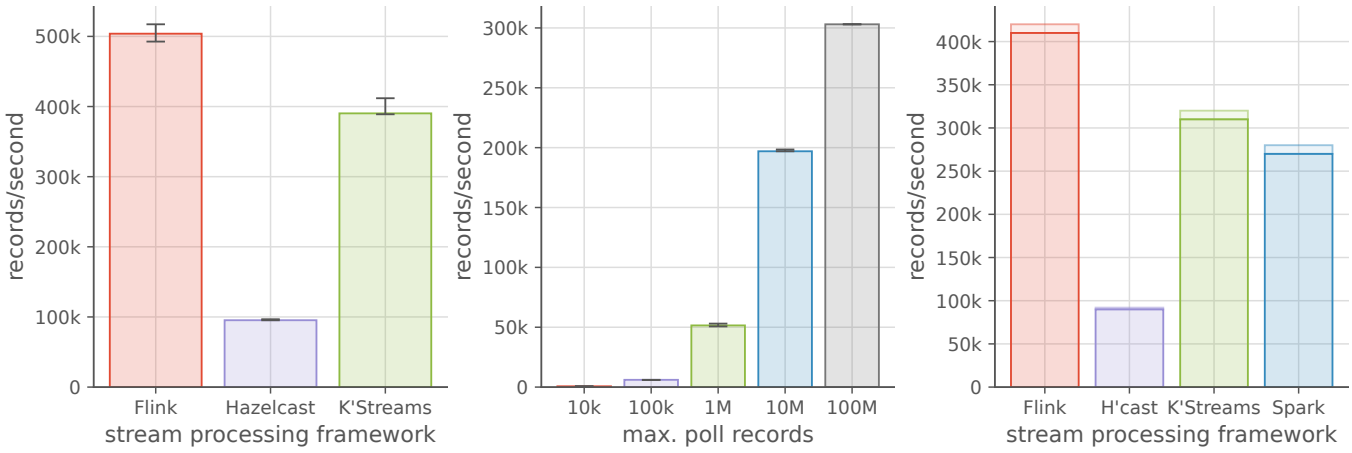
5.1 Experimental Setup

We conduct our experimental evaluation in a Kubernetes cluster managed by the Elastic Kubernetes Service of Amazon Web Services. The cluster consists of 10 nodes provisioned in the *us-east-1* region: 3 *m6i.xlarge* nodes run the stream processing framework, 3 *m6i.2xlarge* nodes run one Kafka broker each, and 4 *m6i.xlarge* nodes run the load generator instances plus additional benchmarking infrastructure.[4] Unless stated differently, we use the following configurations: We deploy the stream processing application with 9 application instances (3 per cluster node). Each instance is assigned 4 GB of memory and a single virtual CPU core, resulting in a total parallelism of 9. Except for a few adjustments for better comparability, we test all frameworks with their default configurations. We set up one million real-time consumers that all have the same selectivity, which sum up to 20 %, meaning that each record is forwarded to 0.2 consumers on average. Each consumer emits an output event for every tenth record. Generated records have a size of 1024 bytes. To increase statistical rigor, we run each experiment for 15 minutes and repeat it three times. We consider this sufficient as our results in the following sections show no large deviations across repetitions.

5.2 Baseline Evaluation of Throughput

In Section 4.3.1, we discuss how the *ad-hoc throughput* method provides throughput results faster than the more realistic *sustainable throughput* method. In this evaluation, we compare the results of both methods based on experiments. For the ad-hoc throughput experiments, we generate 1 million records per second and monitor how many are processed per second by the frameworks. For the sustainable throughput experiments, we generate records with different frequencies and determine the maximum frequency at which the number of queued records in the Kafka input topic does not substantially increase over time (the performance goal, see our previous work for a detailed explanation of this method [16]). In the following, we first discuss the results of Flink, Kafka Streams,

[4] As part of our replication package [17], we also provide the exact setup using established Infrastructure-as-Code tooling.



(a) Ad-hoc throughput **(b) Ad-hoc throughput (Spark)** **(c) Sustainable throughput**

Figure 3: Baseline throughput results obtained with the ad-hoc measurement method of Flink, Hazelcast, and Kafka Streams (a); of Spark for different limitations on the maximum number of pulled records per batch (b); and with the sustainable measurement method of Flink, Hazelcast, Kafka Streams, and Spark (c).

and Hazelcast as they allow for a straightforward interpretation, followed by a more detailed discussion of the results for Spark.

5.2.1 Results of Flink, Hazelcast, and Kafka Streams. Figure 3a shows the results of the ad-hoc throughput measurement method for Flink, Hazelcast, and Kafka Streams. We can observe a clear ranking in which Flink achieves the highest throughput, followed by Kafka Streams, and a considerably lower throughput of Hazelcast.

We contrast these results by the results of the sustainable throughput method as shown in Fig. 3c. Considering the results obtained by the sustainable throughput method as representative of a real deployment, we can see that the ad-hoc method overestimates the throughput of Flink and Kafka Streams by up to 20%. For Hazelcast, we can see no difference. Despite their overestimation, ad-hoc measurements are still useful since the ranking of frameworks is the same with both methods and ad-hoc measurements can be performed significantly faster. For these reasons, we apply the ad-hoc method for the throughput measurements in the following evaluation.

5.2.2 Results of Spark. Unless further adjusted, ad-hoc throughput measurements are not meaningful in Spark, because Spark ingests all available data in a batch, processes this batch, and then ingests the next batch of all available data. Data that is generated at a larger volume than can be processed leads to ever-increasing batch sizes and, thus, to ever-increasing batch processing time. However, Spark allows constraining the maximum number of records pulled per batch. We experiment with different limits and show how they impact the achieved throughput in Fig. 3b. In short, the results demonstrate the intuition that pulling more records increases the throughput. However, this comes at the cost of high latency. For instance, pulling in batches of 10 million records achieves a throughput of approximately 200 000 records per second, but then data is retrieved only every 50 seconds causing high latency (see Section 5.3).

With the sustainable throughput method, we determine the maximum throughput without a persistent increase in queued messages. As shown in Fig. 3c, Spark achieves a throughput of 270 000 to 280 000 records per second. However, this throughput comes again at the cost of large batches with processing times – and thus latencies (see Section 5.3) – of several seconds. Smaller data volumes lead to reduced batch processing times and, thus, lower latency.

In summary, we observe that Spark's throughput can be increased at the cost of increased latency and latency can be decreased at the cost of reduced throughput. This means both metrics should always be considered in relation to each other and providing a single result value is problematic. Nevertheless, we observe that with our benchmark, Spark achieves a throughput similar to the other frameworks only if a latency of a few minutes can be tolerated. As we consider this as too long for most stream processing use cases [21, 32] and to reduce the space of experiment configurations, we limit the maximum number of records pulled per batch to 1 000 000 in the following throughput experiments.

5.3 Baseline Evaluation of Latency

For our baseline latency evaluations, we generate a constant load of 90 000 records per second as our baseline throughput experiments in Section 5.2 showed that this is the maximum rate at which all frameworks are able to process data (see Fig. 3c).

Figure 4 shows a quantile function of the observed latency of Flink, Kafka Streams, and Hazelcast. We exclude Spark in the figure as it has a latency of over 10 seconds at every percentile. It can be seen that Hazelcast processes data with a very low latency of 8 milliseconds at the 95th percentile. The p95 latency of Flink (88 milliseconds) and Kafka Streams (183 milliseconds) are considerably higher but still by an order of magnitude lower than Spark's.

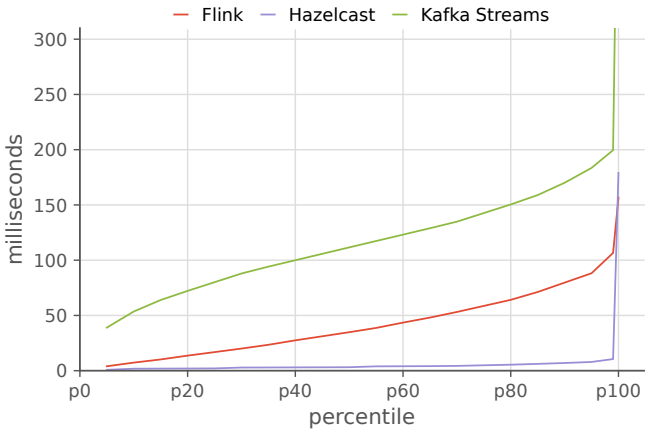


Figure 4: Quantile function of the median observed latency of Flink, Hazelcast and Kafka Streams.

5.4 Evaluation of Deployment Impact

In our baseline experiments, we deploy 9 instances of the stream processing framework, each limited to one virtual CPU core. Thus, all parallelization is happening on the level of Kubernetes Pods. We compare this deployment against deploying 3 instances with each being limited to 3 cores. This introduces a second level of parallelization (i.e., within a single instance) while maintaining an overall parallelism of 9. We scale the assigned memory per instance proportionally to 12 GB.

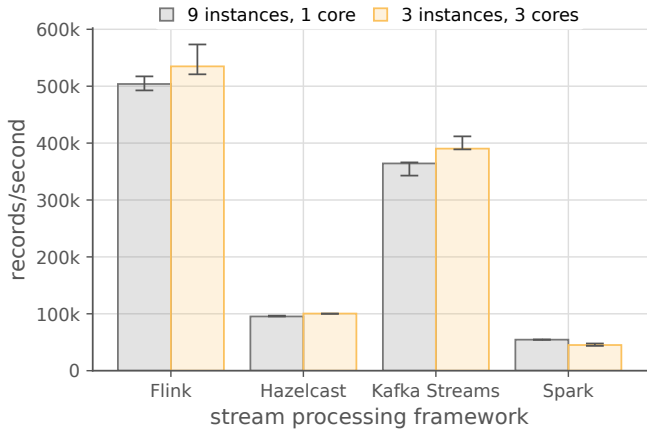
Flink, Hazelcast, and Kafka Streams benefit to a small extent from higher per-instance parallelization as shown in Fig. 5a. However, Spark's throughput decreases slightly. Figure 5b shows that the 3-node deployment has virtually no influence on processing latency for all frameworks.

5.5 Evaluation of Record Size Impact

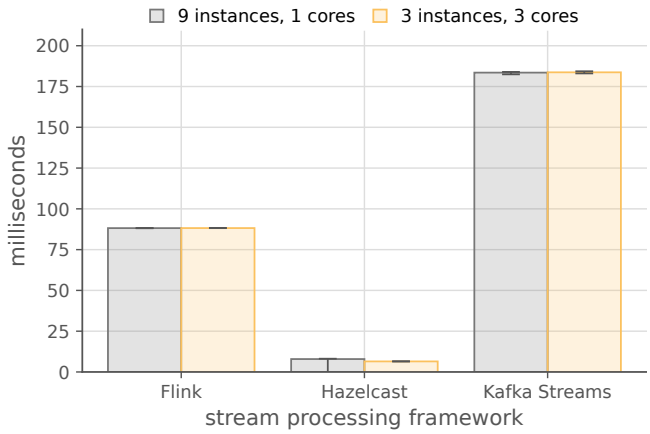
In our baseline experiments, we generate data records of 1024-byte size. While we consider this as a realistic value for our studied industrial use case, there are certainly use cases that process records of other sizes. As related benchmarking studies often used considerably smaller records [13, 18, 34], we evaluate the performance with record sizes of 128 bytes, 256 bytes, and 512 bytes.

Figure 6a shows that all frameworks can process more records as record sizes become smaller. It is interesting to see that in particular Hazelcast benefits from smaller record sizes. While with 1024-byte records, Flink achieves 5.3× higher and Kafka Streams 3.8× higher throughput than Hazelcast, Hazelcast achieves 68% of the throughput of Flink and approximately the same throughput as Kafka Streams with 128-byte records. It is also worth noting that records smaller than 256 bytes have only a minimal impact on Kafka Streams' throughput.

Fig. 6b shows that the record size only slightly affects the processing latency. We can even see a small increase in latency with smaller records, which may seem counterintuitive. We expect this to be because data from Kafka is pulled in batches (limited by its size in bytes). Smaller records imply more records per batch and, thus, that data is potentially longer queued in Kafka before being pulled.



(a) Ad-hoc throughput



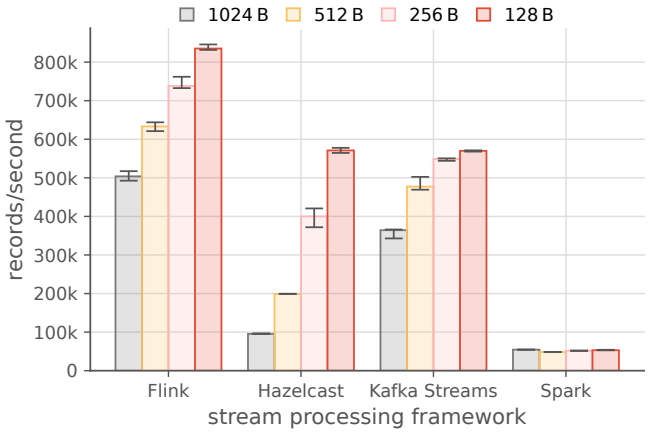
(b) Latency at the 95th percentile

Figure 5: Experimental results comparing a deployment with 9 instances and one core per instance with a deployment of 3 instances with 3 cores per deployment.

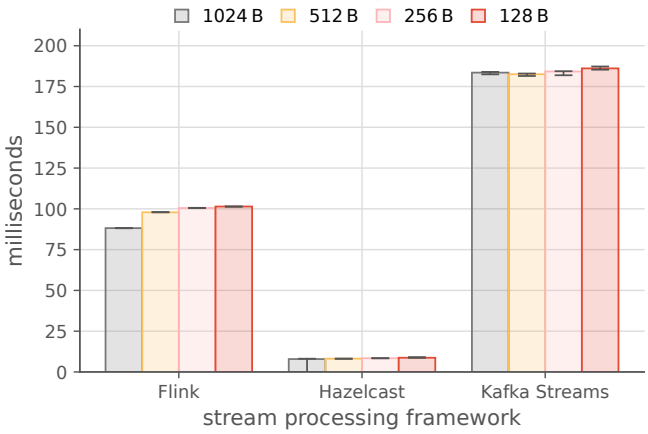
5.6 Evaluation of Consumer Count Impact

ShuffleBench allows for configuring the number of real-time consumers. The number of consumers impacts the runtime of the matcher service as well as the number of state entries and, thus, the overall state size. Our baseline evaluation used 1 million consumers, which we now compare to 100 000 consumers. Evaluations with significantly larger numbers would require more memory and are therefore not covered in the evaluation.

Figure 7a shows that with fewer real-time consumers, the throughput of Flink and Kafka Streams significantly increases, whereas we can see no change for Hazelcast. This indicates that the bottleneck of Hazelcast is unrelated to the number of consumers. With all frameworks, there is no change in the processing latency when using fewer consumers as shown in Fig. 7b.

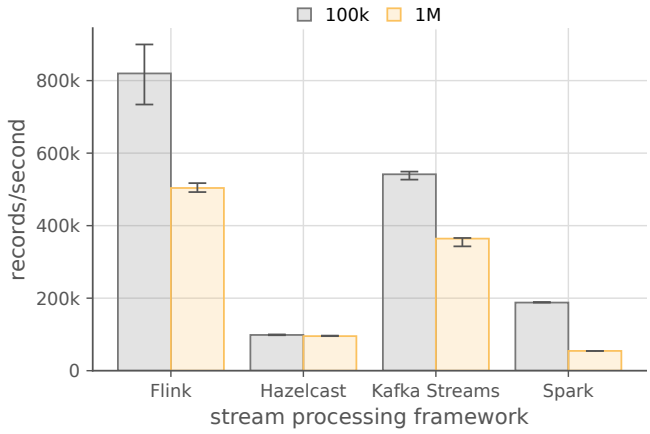


(a) Ad-hoc throughput

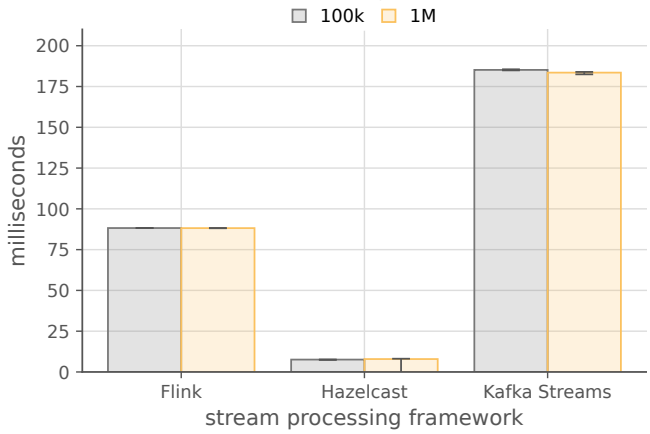


(b) Latency at the 95th percentile

Figure 6: Experimental results comparing the impact of different record sizes.



(a) Ad-hoc throughput



(b) Latency at the 95th percentile

Figure 7: Experimental results comparing 100 000 with one million real-time consumers.

5.7 Evaluation of Selectivity Impact

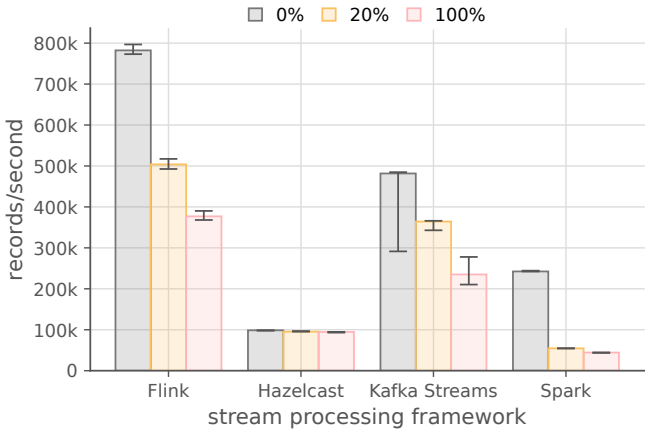
The summed-up selectivity of all real-time consumes describes how much of the overall input data volume is forwarded to the *groupBy* operation of ShuffleBench's dataflow architecture. Hence, it also determines the data volume that is re-distributed to a potentially different application instance. Besides a total selectivity of 20% in our baseline evaluation, we also compare a 0% and a 100% total selectivity.

Figure 8a shows the throughput results for all frameworks. All frameworks except Hazelcast achieve a higher throughput with smaller selectivity. Hazelcast's results indicate that its bottleneck is neither due to the shuffling nor the stateful aggregation. Noticeable is that Spark's results throughput significantly reduces from 0% selectivity to 20% selectivity, but only slightly from 20% selectivity to 100% selectivity. Figure 8b shows the 95th percentile processing latency of Flink, Hazelcast, and Kafka Streams for 20% selectivity and 100% selectivity. There can be no latency results for 0% selectivity, as no data is shuffled and, thus, no events are output. Hazelcast shows a very high latency with 100% selectivity as, in this case, it can not process the generated data volume anymore, which means that records are queuing up in Kafka. Surprisingly, with higher selectivity, the latency of Flink and Kafka Streams decreases. We expect this again to be due to buffering effects in the shuffling and in the output step.

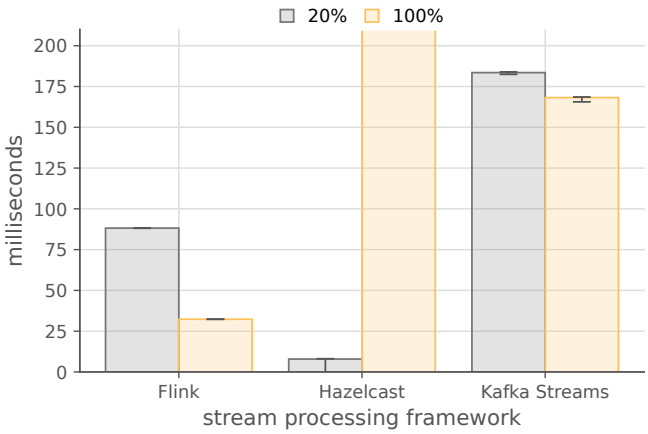
5.8 Discussion in the Context of Related Work

Across all our experiments, Flink processes data with the highest throughput, followed by Kafka Streams. Only for small records of 128-byte size, Hazelcast achieves a similar throughput to Kafka Streams. On the other hand, Hazelcast processes data with about ten times lower latency compared to Flink, which in turn requires approximately half the time of Kafka Streams. In our experiments with Spark, a strong correlation between throughput and latency can be noted. Engineers and operators can thus choose the right balance depending on the use case. To process data of a volume similar to that processed by other frameworks, however, latency in the order of several seconds up to minutes has to be tolerated.

Flink's superior performance for many use cases is also reported in the literature [16, 21, 34]. The throughput achieved with Spark was not as high (compared to other frameworks) as reported in



(a) Ad-hoc throughput



(b) Latency at the 95th percentile

Figure 8: Experimental results comparing a total selectivity across all real-time consumers of 0%, 20%, and 100%.

some related studies [20, 34]. We believe that this is due to specific technical aspects of our benchmark and experimental design of avoiding configuration tuning (see Section 5.9), which can be a requirement for Spark's execution in clusters [34]. In the future, we intend to further investigate Spark's configurations and their impact on performance. Likewise, Hazelcast's low throughput is surprising and could not be observed in related work [16]. Whether this is only due to our larger record sizes or there are further particularities in our benchmark or experimental setup has to be further evaluated.

Hazelcast's low latency processing was also observed in the related literature [10, 18] and is underpinned by its design [10]. Also, the strong correlation between throughput and latency in Spark is reported in the literature [34, 37]. Recent, experimental approaches in Spark for continuous processing[5] to reduce latency could be subject of an extended evaluation.

[5]https://spark.apache.org/docs/3.5.0/structured-streaming-programming-guide.html#continuous-processing

5.9 Threats to Validity

Despite careful research design, there are threats and limitations to the validity of our experimental evaluation, which we report below.

Threats to Internal Validity. We run all evaluations in a containerized environment on a public cloud platform to have a representative deployment. This means, however, that there are potentially many factors influencing the performance, which are out of our control [25, 28]. We address these limitations to some extent by running individual experiments for a longer time, repeating them, and assessing their variability. However, to reduce the benchmarking setup's complexity, we do not re-create the Kubernetes cluster between two experiments, which means that virtual machines are not re-provisioned before each experiment. We do not systematically run experiments at different times of the day or the week to keep the time required for these experiments manageable. Repeating certain experiments at different times, however, did not show noticeable deviations. To further mitigate these limitations, we provide our benchmark as open-source software, allowing for independent replication of our study.

Threats to External Validity. It is important to note that our results report on the performance of different stream processing frameworks for large-scale data shuffling use cases. Previous research has shown that benchmark results of one use case are not necessarily transferable to other use cases [16, 27, 37]. In Sections 5.4 to 5.7 we conduct throughput experiments with the less representative ad-hoc measurement method to reduce the overall time required for these experiments. Section 5.2 quantifies how much this method overestimates the realistic throughput. We intentionally evaluate all frameworks primarily using their default configurations. This approach helps to avoid bias resulting from different degrees of experience with the frameworks. However, we can only draw limited conclusions about potential performance improvements that can be achieved through fine-tuning for specific scenarios. For our experiments, we focused on a single kind of deployment (containers in Kubernetes) on a single cloud platform. As we measure performance on a high level (macro-benchmarking) using cloud-native abstraction layers and by setting resource limits for the containers, we expect no significantly different results in other execution environments.

6 CONCLUSIONS AND FUTURE WORK

This paper introduces ShuffleBench, our proposal for a new stream processing benchmark for large-scale data shuffling operations. Besides addressing a different use case than other stream processing benchmarks, ShuffleBench also overcomes several limitations of other benchmarks. Our benchmark design is based on requirements identified from a large cloud observability platform and established benchmarking metrics, methods, and techniques from the performance engineering research community. With ShuffleBench, we aim to support and foster research on stream processing by providing a standardized method that researchers and practitioners can use to compare their implementations, algorithms, and configurations. So far, we employed ShuffleBench to evaluate throughput and latency of Flink, Hazelcast, Kafka Streams, and Spark. Our

evaluation provides the most recent benchmark results to the research community and serves as a starting point for researchers and practitioners to conduct further evaluations with ShuffleBench.

Besides growing a community around ShuffleBench, we plan to support additional qualities such as reliability. In particular, an empirical investigation of the interconnection of throughput, latency, and fault-tolerance is highly demanded. We are also currently in the process of supporting and evaluating non-uniformly distributed record sizes, selectivities, state sizes, and output rates to address additional industrial requirements.

ACKNOWLEDGMENTS

We would like to thank the Johannes Kepler University Linz and Dynatrace for co-funding this research.

REFERENCES

[1] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful Functions as a Service in Action. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 1890–1893. https://doi.org/10.14778/3352063.3352092

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[3] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 480–491.

[4] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, 601–613. https://doi.org/10.1145/3183713.3190664

[5] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Cláudio F. R. Geyer, and Luiz Gustavo L. Fernandes. 2020. DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems. *IEEE Access* 8 (2020), 222900–222917. https://doi.org/10.1109/ACCESS.2020.3043948

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[7] Badrish Chandramouli, Jonathan Goldstein, Roger Barga, Mirek Riedewald, and Ivo Santos. 2011. Accurate latency estimation in a distributed event processing system. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 255–266. https://doi.org/10.1109/ICDE.2011.5767926

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[9] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2024. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (2024), 507–541. https://doi.org/10.1007/s00778-023-00819-8

[10] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz, Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Hazelcast Jet: Low-Latency Stream Processing at the 99.99th Percentile. *Proceedings of the VLDB Endowment* 14, 12 (July 2021), 3110–3121. https://doi.org/10.14778/3476311.3476387

[11] Wilhelm Hasselbring. 2021. Benchmarking as Empirical Standard in Software Engineering Research. In *Evaluation and Assessment in Software Engineering (EASE '21)*. ACM, 457–462. https://doi.org/10.1145/3463274.3463361

[12] Sören Henning and Wilhelm Hasselbring. 2021. How to Measure Scalability of Distributed Stream Processing Engines?. In *Companion of the ACM/SPEC International Conference on Performance Engineering*. ACM. https://doi.org/10.1145/3447545.3451190

[13] Sören Henning and Wilhelm Hasselbring. 2021. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research* 25 (2021), 100209. https://doi.org/10.1016/j.bdr.2021.100209

[14] Sören Henning and Wilhelm Hasselbring. 2022. A Configurable Method for Benchmarking Scalability of Cloud-Native Applications. *Empirical Software Engineering* 27, 6 (Aug. 2022). https://doi.org/10.1007/s10664-022-10162-1

[15] Sören Henning and Wilhelm Hasselbring. 2022. Demo Paper: Benchmarking Scalability of Cloud-Native Applications with Theodolite. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 275–276. https://doi.org/10.1109/IC2E55432.2022.00037

[16] Sören Henning and Wilhelm Hasselbring. 2024. Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. *Journal of Systems and Software* 208 (2024), 111879. https://doi.org/10.1016/j.jss.2023.111879

[17] Sören Henning, Adriano Vogel, Michael Leichtfried, Otmar Ertl, and Rick Rabiser. 2024. *Replication Package for: ShuffleBench: A Benchmark for Large-Scale Data Shuffling Operations with Distributed Stream Processing Frameworks*. https://doi.org/10.5281/zenodo.10605615

[18] Guenter Hesse, Christoph Matthies, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. 2021. ESPBench: The Enterprise Stream Processing Benchmark. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '21)*. ACM, 201–212. https://doi.org/10.1145/3427921.3450242

[19] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *Comput. Surveys* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

[20] Ziya Karakaya, Ali Yazici, and Mohammed Alayyoub. 2017. A Comparison of Stream Processing Frameworks. In *2017 International Conference on Computer and Applications (ICCA)*. IEEE, 1–12. https://doi.org/10.1109/COMAPP.2017.8079733

[21] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[22] Asterios Katsifodimos and Marios Fragkoulis. 2019. Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology*. OpenProceedings.org, 682–685. https://doi.org/10.5441/002/edbt.2019.86

[23] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. 2020. *Systems Benchmarking*. Springer.

[24] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the International Workshop on Networking Meets Databases*.

[25] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology* 16, 3, Article 15 (April 2016), 23 pages. https://doi.org/10.1145/2885497

[26] H. Nasiri, S. Nasehi, and M. Goudarzi. 2019. Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities. *Journal of Big Data* 6, 52 (2019). https://doi.org/10.1186/s40537-019-0215-2

[27] Alessio Pagliari, Fabrice Huet, and Guillaume Urvoy-Keller. 2020. NAMB: A Quick and Flexible Stream Processing Application Prototype Generator. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 61–70. https://doi.org/10.1109/CCGrid49817.2020.00-87

[28] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, José Nelson Amaral, Petr Tůma, and Alexandru Iosup. 2021. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1528–1543. https://doi.org/10.1109/TSE.2019.2927908

[29] Tobias Pfandzelter, Sören Henning, Trever Schirmer, Wilhelm Hasselbring, and David Bermbach. 2022. Streaming vs. Functions: A Cost Perspective on Cloud Event Processing. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 67–78. https://doi.org/10.1109/IC2E55432.2022.00015

[30] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, Breno Bernard Nicolau de França, Carlo Alberto Furia, Greg Gay, Nicolas Gold, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara Kitchenham, Valentina Lenarduzzi, Jorge Martínez, Jorge Melegati, Daniel Mendez, Tim Menzies, Jefferson Molleri, Dietmar Pfahl, Romain Robbes, Daniel Russo, Nyyti Saarimäki, Federica Sarro, Davide Taibi, Janet Siegmund, Diomidis Spinellis, Miroslaw Staron, Klaas Stol, Margaret-Anne Storey, Davide Taibi, Damian Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, Xiaofeng Wang, and Sira Vegas. 2021. Empirical Standards for Software Engineering Research. https://doi.org/10.48550/arXiv.2010.03525 Version 0.2.0.

[31] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics (BIRTE '18)*. ACM, Article 1, 10 pages. https://doi.org/10.1145/3242153.3242155

[32] Radu Tudoran, Alexandru Costan, Olivier Nano, Ivo Santos, Hakan Soncu, and Gabriel Antoniu. 2016. JetStream: Enabling high throughput live event streaming on multi-site clouds. *Future Generation Computer Systems* 54 (2016), 274–291. https://doi.org/10.1016/j.future.2015.01.016

[33] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM,

333–336. https://doi.org/10.1145/2668930.2688819

[34] Giselle van Dongen and Dirk van den Poel. 2020. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1845–1858. https://doi.org/10.1109/TPDS.2020.2978480

[35] Giselle van Dongen and Dirk van den Poel. 2021. Influencing Factors in the Scalability of Distributed Stream Processing Jobs. *IEEE Access* 9 (2021), 109413–109431. https://doi.org/10.1109/ACCESS.2021.3102645

[36] Giselle van Dongen and Dirk van den Poel. 2021. A Performance Analysis of Fault Recovery in Stream Processing Frameworks. *IEEE Access* 9 (2021), 93745–93763. https://doi.org/10.1109/ACCESS.2021.3093208

[37] Adriano Vogel, Sören Henning, Otmar Ertl, and Rick Rabiser. 2023. A systematic mapping of performance in distributed stream processing systems. In *Euromicro Conference on Software Engineering and Advanced Applications*. IEEE. https://doi.org/10.1109/SEAA60479.2023.00052

[38] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. 2021. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. ACM, 2602–2613. https://doi.org/10.1145/3448016.3457556

[39] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

Vectorized Intrinsics Can Be Replaced with Pure Java Code without Impairing Steady-State Performance

Júnior Löff
Faculty of Informatics
Università della Svizzera italiana (USI)
Lugano, Switzerland
loeffj@usi.ch

Filippo Schiavio
Faculty of Informatics
Università della Svizzera italiana (USI)
Lugano, Switzerland
filippo.schiavio@usi.ch

Andrea Rosà
Faculty of Informatics
Università della Svizzera italiana (USI)
Lugano, Switzerland
andrea.rosa@usi.ch

Matteo Basso
Faculty of Informatics
Università della Svizzera italiana (USI)
Lugano, Switzerland
matteo.basso@usi.ch

Walter Binder
Faculty of Informatics
Università della Svizzera italiana (USI)
Lugano, Switzerland
walter.binder@usi.ch

ABSTRACT

Several methods of the Java Class Library (JCL) rely on *vectorized intrinsics*. While these intrinsics undoubtedly lead to better performance, implementing them is extremely challenging, tedious, error-prone, and significantly increases the effort in understanding and maintaining the code. Moreover, their implementation is platform-dependent. An unexplored, easier-to-implement alternative is to replace vectorized intrinsics with portable Java code using the Java Vector API. However, this is attractive only if the Java code achieves similar steady-state performance as the intrinsics.

This paper shows that this is the case. We focus on the `hashCode` and `equals` computations for byte arrays. We replace the platform-dependent vectorized intrinsics with pure-Java code employing the Java Vector API, resulting in similar steady-state performance. We show that our Java implementations are easy to fine-tune by exploiting characteristics of the input (i.e., the array length), while such tuning would be much more difficult and cumbersome in a vectorized intrinsic. Additionally, we propose a new vectorized `hashCode` computation for long arrays, for which a corresponding intrinsic is currently missing. We evaluate the performance of the tuned implementations on four popular benchmark suites, showing that the performance are in line with those of the original OpenJDK 21 with intrinsics.

Finally, we describe a general approach to integrate code using the Java Vector API into the core classes of the JCL, which is challenging because premature use of the Java Vector API would crash the JVM during its fragile initialization phase. Our approach can be adopted by developers to modify JCL classes without any changes to the native codebase.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; **Evaluation**; **Performance**; • **Software and its engineering** → *Just-in-time compilers*; *Dynamic compilers*; *Runtime environments*; • **Computing methodologies** → **Parallel programming languages**; • **Computer systems organization** → **Single instruction, multiple data**.

KEYWORDS

Compiler Intrinsics, SIMD, Vector Instructions, Java Vector API, Portability, Java Virtual Machine.

ACM Reference Format:
Júnior Löff, Filippo Schiavio, Andrea Rosà, Matteo Basso, and Walter Binder. 2024. Vectorized Intrinsics Can Be Replaced with Pure Java Code without Impairing Steady-State Performance. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645051

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645051

1 INTRODUCTION

In the context of the Java Virtual Machine (JVM), *intrinsics* [2, 21] are a complex runtime machinery introduced for implementing efficient low-level operations.[1] Intrinsics are often implemented in the form of template-generated, assembly-like code or as dedicated nodes in the intermediate representation (IR) used by the just-in-time (JIT) compiler. Several methods of the Java Class Library (JCL) are implemented as intrinsics to improve performance [19]: instead of relying on the JIT compiler to optimize a pure-Java implementation, the JIT compiler directly emits machine code as specified by the intrinsic. In this way, the compilation cost is reduced since the compiler does not need to perform optimization passes. Moreover, by leveraging intrinsics, JVM developers can express low-level computations which are not directly expressible in the Java language. With the increasing support of *vector instructions* [1] (i.e., special machine instructions leveraging SIMD registers to apply an operation to multiple data elements in parallel) in common processors, intrinsics can leverage such instructions to improve performance even on ordinary hardware. We refer to such intrinsics as *vectorized intrinsics*.

[1] Intrinsics are used for multiple reasons, such as for making low-level resources accessible or for improving performance. This paper focuses solely on the latter type of intrinsics, in particular on those making use of vector instructions.

While intrinsics undoubtedly lead to better performance, implementing them is extremely challenging [8, 50]. In addition to the very specific knowledge needed to write highly optimized, template-generated assembly code for a target architecture, implementing an intrinsic is hard, tedious, error-prone, requires a major development effort, and makes debugging, tuning, and testing very time-consuming. Moreover, as intrinsics are platform-dependent, the above effort should be repeated for every architecture that one wishes to support. This issue is aggravated in vectorized intrinsics, since even processors of the same architecture may support different vector extensions. As an example, x86 processors may support Streaming SIMD Extensions [24] (SSE), which have been released in four different versions (SSE1, SSE2, SSE3, SSE4) and Advanced Vector Extensions [25] (AVX), released in three different versions (AVX, AVX2, AVX512). To fully leverage vector instructions, developers need to implement different versions of the vectorized intrinsics, each making use of the vector instructions supported by the underlying processor. Due to this complexity and effort required, in practice, developers implement vectorized intrinsics only for very common architectures (e.g., x86) and only for selected methods of the JCL (e.g., commonly and frequently used primitives on byte arrays such as `hashCode` and `equals`), where the complexity and cost of implementing an intrinsic is justified by a significant performance gain.

These issues—huge effort in development, debugging, and testing, limited portability, substantial platform dependence of vector instructions—can be mitigated by replacing vectorized intrinsics with equivalent platform-independent Java code making use of the *Java Vector API* (JVA) [35], which allows developers to express explicit vector operations from Java code using an object-oriented API, without resorting to any native code. Using the API, appropriate vector instructions are emitted by the JIT compiler depending on the vector instructions supported by the architecture; hence, the same Java code can be executed on multiple platforms, possibly supporting different vector extensions.

The goal of this paper is to show that in addition to reducing the effort in understanding, extending, and maintaining the code, replacing vectorized intrinsics with equivalent Java code using the JVA results in similar steady-state performance, making the approach attractive even in production-level JVMs. We present a brief background on intrinsics in Section 2. As practical use cases, we apply this approach to the `hashCode` (Section 3) and `equals` (Section 5) computations for byte arrays. Moreover, as the code is now easier to extend, debug, and test, we show that it is also easy to fine-tune the code, exploiting characteristics of the input (i.e., the array length); fine-tuning would be very difficult to implement in an intrinsic. Furthermore, we propose a new vectorized `hashCode` computation for long arrays, for which a corresponding intrinsic is currently missing (Section 4). We evaluate the performance of the tuned implementations of `hashCode` and `equals` for byte arrays (but not the new vectorized methods) using four popular benchmark suites, showing that they provide similar performance than the vectorized intrinsics (Section 6). Our implementation is portable and can be run on any JVM and any architecture supported by the JVA, whereas the vectorized intrinsics in OpenJDK 21 [14] (the latest version of one of the most used JVMs worldwide) only work on selected architectures.

The JVA cannot be used during the fragile JVM initialization phase, because it would lead to the premature initialization of classes when the JVM is not yet ready to execute arbitrary Java code. This is a major obstacle, because many methods of the JCL (including `hashCode` and `equals`, on which we focus) are already exercised in the early phases of JVM initialization. This paper proposes a general approach to solve this issue. We describe our approach to modify the Java source code of the JCL, such that our Java implementation making use of the JVA can substitute the use of intrinsics in core classes of the JCL (Section 7). Our approach does not require any change in the native codebase.

To summarize, we show a new, practical way to implement primitives in the JCL using the JVA, offering an attractive, more portable, and easier-to-maintain alternative to the use of vectorized intrinsics. Our approach does not require any knowledge of the OpenJDK native codebase and can be reused by researchers and practitioners to modify JCL classes. In particular, our work makes the following contributions:

- We propose and evaluate fine-tuned JVA implementations of `hashCode` and `equals` for byte arrays (Sections 3 and 5, respectively).
- We propose and evaluate a new vectorized `hashCode` JVA implementation for long arrays (Section 4).
- We assess the performance of our new JVA implementations of `hashCode` and `equals` for byte arrays on four popular benchmark suites for Java (Section 6).
- We propose an approach to modify the Java source code of the core classes of the JCL (Section 7).

We complement the paper with an overview of related work (Section 8), a discussion on the limitations of our approach (Section 9), and our concluding remarks (Section 10).

2 BACKGROUND

Intrinsic functions (henceforth also called *intrinsics* for short and also known as *built-in functions*) are functions whose implementations are specially handled by the managed language runtime system. In particular, the managed language runtime system provides 1) a default implementation of the intrinsic function, written in the managed language, and 2) operating system- and architecture-specific semantically equivalent optimized implementations of the intrinsic function. If an optimized implementation is available for the underlying operating system and architecture, at runtime, the interpreter and/or the compiler replaces the default implementation with that optimized implementation. Optimized implementations often leverage operating system and hardware features that are not available in the managed language constructs and APIs, such as vector operations.

Since implementing intrinsics is challenging [8, 50], JVMs implement intrinsics only for some particular architectures and only for some frequently used methods provided by the JCL. Intrinsic implementations may vary not only depending on the operating system and the underlying architecture but also based on the JVM version, JVM vendor, and JIT compiler the JVM uses. For this reason, the platform-dependency of intrinsics aggravates the unpredictable performance of Java code on different architectures.

In this paper, we conduct our experiments on OpenJDK, a widely used open-source JVM implementation. OpenJDK specifies two types of intrinsics [21]: 1) *library intrinsics* that "may be replaced with hand-crafted assembly code, with hand-crafted compiler IR, or with a combination of the two" [22], and 2) *bytecode intrinsics* that are "not replaced by special code, but they are treated in some other special way by the compiler" [20]. In our work, we focus on intrinsics replaced by the JIT compiler and implemented as hand-crafted assembly code that leverages vector hardware instructions. We note that implementing this kind of instrinsics in OpenJDK not only requires knowledge of the low-level assembly programming model, but also knowledge of the metaprogramming techniques used to implement their code generation.

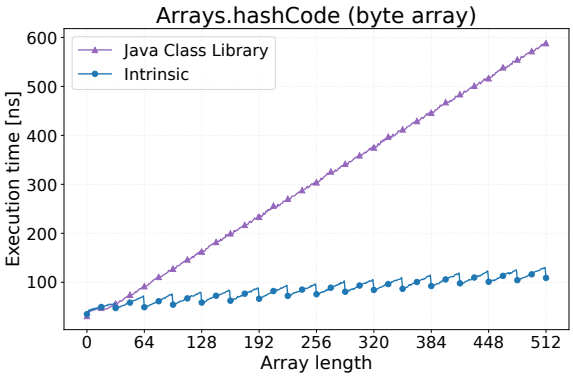


Figure 1: Performance comparison between the OpenJDK 21 `hashCode` intrinsic and the byte-to-byte loop in pure-Java code implemented in the JCL up to OpenJDK 20.

3 VECTORIZED HASHCODE FOR BYTE ARRAYS

In this section, we present the vectorized implementation and evaluation of the `hashCode` method. In Java, every object and array supports method `hashCode`, which returns an integer associated with the object or array. In this section, we focus on the `hashCode` computation for byte arrays, which is used also to compute the `hashCode` for Java Strings (as they are internally implemented as byte arrays). An efficient `hashCode` calculation for Java String is very important in multiple scenarios, such as pattern-matching algorithms [30, 31, 46, 47], data-processing systems [10, 23, 49], and data compression [18]. `Arrays.hashCode` is implemented as a variant of the *polynomial rolling hash* function [28], computed as reported in Equation 1.

$$hash(s) = \begin{cases} 0 & \text{if } N = 0 \\ \sum_{i=0}^{N-1} s[i] \cdot p^{N-i-1} \mod m & \text{otherwise} \end{cases} \quad (1)$$

where s is an array of length N, $s[i]$ is the i-th element of s, while p and m are positive integers. In OpenJDK 21, $p = 31$, $m = 2^{32}$, and the computation is performed by a vectorized intrinsic requiring AVX2 vector instructions (which use 256-bit vector registers). We note that the mod 2^{32} operation is obtained implicitly due to overflows. The intrinsic has recently been introduced in OpenJDK 21 [13]. We highlight that our experiments show that the previous implementation of `hashCode` (i.e., as simple scalar loop) is not automatically optimized with the superword auto-vectorization optimization performed by the JIT compiler: as shown in Figure 1, the intrinsic largely outperforms the pure-Java implementation.

3.1 HashCode Implementation for Byte Arrays

The implementation of the vectorized intrinsic for `Arrays.hashCode` in OpenJDK 21 can be found in the link [17]. Due to its complexity, for the sake of clarity, we introduce the *recast* version of the intrinsic (i.e., an equivalent version using pure Java code relying on the JVA) and discuss this version. We note that all our versions presented in this paper have been thoroughly tested with generated test cases. The pseudo-code of the recast is shown in Figure 2. Variables with a bar on top denote vectors. Since the intrinsic relies on AVX2 instructions, the implementation (using four accumulators) efficiently processes 32 bytes in each iteration. The AVX2 instructions can only operate on 8 input bytes at a time, since each byte is converted to a 32-bit integer (zero extension), resulting in a fourfold size increase of the data.

The implementation initializes the accumulators (each containing 8 integers) with zeroes (lines 5–8). Then, it loads the coefficient representing the constant 31^{32} (line 9). This constant depends on the number of bytes processed per iteration (32 in our example) and is loaded from the `POW31` array. Both our implementation and the intrinsic use pre-computed powers of 31. At position i, the `POW31` array stores the value 31^i.

The main computation unfolds through the sequence of loading data (lines 12–15) and accumulating the computed intermediate results (lines 16–19). The accumulators are updated with the product of the prior accumulation and the coefficient, adding then the latest loaded data. Subsequently, the index advances by 32, reinstating the computation if at least 32 bytes remain. After the loop, each vector is multiplied with a reversed list storing the powers of 31 (lines 21–24). Ultimately, an aggregation is applied to all accumulators through an addition (line 25), yielding the hash value.

Subsequently to the execution of the unrolled vectorized loop, the code processes eventual residual bytes (up to 31). If such bytes exist (i.e., the array length is not a multiple of 32), they are aggregated using a scalar loop that processes 2 bytes in each iteration. This can be seen in lines 29–31. As in the case of the vectorized unrolled loop, the intrinsic uses this strategy to reduce the dependencies between iterations. After the loop, a single remaining byte may still need to be processed. If so, it is subsequently added to the hash value (line 33).

During the implementation of the recast version, we observed two improvements that could be applied to use more vector instructions. We describe the improvements in the following text and implement them in a *tuned* version of `hashCode`.

The first improvement regards the computation of hashcode for arrays of < 32 bytes. In this case, the intrinsic processes all the input data with scalar operations (processing 2 bytes in each loop iteration). In our tuned version, we use scalar computation only for arrays of < 8 bytes, while for lengths from 8 to 31, we use vectorized instructions relying on a single accumulator.

```
int hashCode_recast(byte[] s) {
 int len = s.length;
 int h = 0;
 if (len >= 32) {
  acc0 = zero(8_INT);
  acc1 = zero(8_INT);
  acc2 = zero(8_INT);
  acc3 = zero(8_INT);
  m = load(8_INT, POW31[32]);
  int bound = len & ~31; // (length/32)*32
  for (int i = 0; i < bound; i += 32) {
   data0 = fromArray(8_BYTE, s, i);
   data1 = fromArray(8_BYTE, s, i+8);
   data2 = fromArray(8_BYTE, s, i+16);
   data3 = fromArray(8_BYTE, s, i+24);
   acc0 = acc0.mul(m).add(data0.convert(...));
   acc1 = acc1.mul(m).add(data1.convert(...));
   acc2 = acc2.mul(m).add(data2.convert(...));
   acc3 = acc3.mul(m).add(data3.convert(...));
  }
  acc0 = acc0.mul(POW31[31..24]);
  acc1 = acc1.mul(POW31[23..16]);
  acc2 = acc2.mul(POW31[15..8]);
  acc3 = acc3.mul(POW31[7..0]);
  h = acc0.add(acc1).add(acc2)
             .add(acc3).reduce(ADD);
 }
 int i = 1 + (len & ~31);
 for (; i < len ; i += 2) {
  h = 31 * 31 * h + 31 * (s[i-1] & 0xff) +
      (s[i] & 0xff);
 }
 if (i == len) {
  h = 31 * h + (s[i-1] & 0xff);
 }
 return h;
}
```

Figure 2: Pseudo-code of the recast version of `hashCode` for byte arrays using the JVA.

The second improvement applies for lengths > 32 that are not multiples of 32. The intrinsic processes the residual bytes (i.e., length mod 32) in a scalar loop. Instead, we use up to four additional vector instructions to process 8 bytes at a time. At the end, our tuned implementation loads the last 8 bytes of the array and applies a vector mask, setting to zero the vector positions that contain previously processed bytes to avoid processing them twice. The remaining (not masked) bytes are finally processed in the same way as in the loop using only vector instructions.

3.2 Evaluation Methodology and Experimental Setup

We designed a micro-benchmark to compare the execution time of the different `hashCode` implementations for different array lengths. Our goal is to understand whether our pure-Java implementations (recast and tuned) achieve similar steady-state performance as the intrinsic. In this section, we describe our evaluation methodology. We will use the same methodology in Sections 4.2 and 5.2. All the experiments in this paper are executed on a machine equipped with an Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz, featuring 16 physical cores supporting the AVX512 instruction set (supporting 512-bit vector registers). Hyper-threading and turbo boost are disabled. The machine has 256GB of RAM @ 3200MHz. The kernel is Linux 5.15.0-25-generic, and the OS is Ubuntu 22.04 LTS. We use OpenJDK build 21.0.1+12-29.

Our figures report the execution time for each array length from 0 to 512. In each run of the micro-benchmark (within one JVM process), we perform 40k series of measurements, each series with 513 measurements (on byte arrays of lengths 0–512). In each series, we execute the 513 measurements in a randomized order, to ensure that our dynamically compiled Java implementations do not gain any unfair advantage due to predictable execution paths. Among the 40k series of measurements, we consider the first 20k as warm-up, ignoring them. We run the micro-benchmark 5 times in different JVM processes. Overall, each data point (i.e., execution time) in the shown figures is the arithmetic mean of 100k measurements (the 20k steady-state measurements for each of the 5 runs). We note that there is no object allocation during these measurements, and we use the Epsilon no-op garbage collector [4] to prevent any interference of the measurements by the garbage collector. In all figures where the x-axis represents different array lengths, the y-axis shows the execution time in nanoseconds.

3.3 Evaluation of HashCode for Byte Arrays

Figure 3 presents a comparison of the steady-state performance of different `hashCode` implementations: the `hashCode` *intrinsic* implemented in OpenJDK 21 and the *recast* and *tuned* versions using the JVA. As the intrinsic only uses AVX2 instructions, the recast and tuned versions only use 256-bit vector instructions as well. The figures also show a tuned version that exploits AVX512 (i.e., 512-bit vector instructions), which was easily written with the JVA. For a fair comparison with the intrinsic, our focus remains exclusively on the 256-bit implementation. We show the AVX512 version only to highlight the potential for further performance enhancement on processors that support 512-bit vector instructions.

As shown in the figure, the curves of each version exhibit different characteristics. The performance trend of the intrinsic and the recast is characterized by its main vectorized loop using four accumulators and the residual computation employing a scalar loop. This can be seen in the figure, where the shortest execution times are measured for array lengths that are multiples of 32, for which the `hashCode` computation involves only vector instructions. Then, the subsequent 31 data points in the curve follow a pattern of increasing overhead due to the scalar loop.

The performance curve of our tuned implementation is also shaped by its main vectorized loop featuring four accumulators (for lengths ≥ 32). However, a notable difference lies in the final phase of the `hashCode` computation. Here, we substitute the scalar `for` loop iterations with vector instructions that process up to 8 bytes at a time. Since often there are less than 8 residual bytes, we apply a mask to disable unused lanes of the vector instruction. This approach leads to a discernible pattern marked by four steps within each 32-byte range. Each step in the range corresponds to the final vector instructions processing 8 bytes using a mask, accompanied by extra vector instructions (at most three) every 8 bytes. While

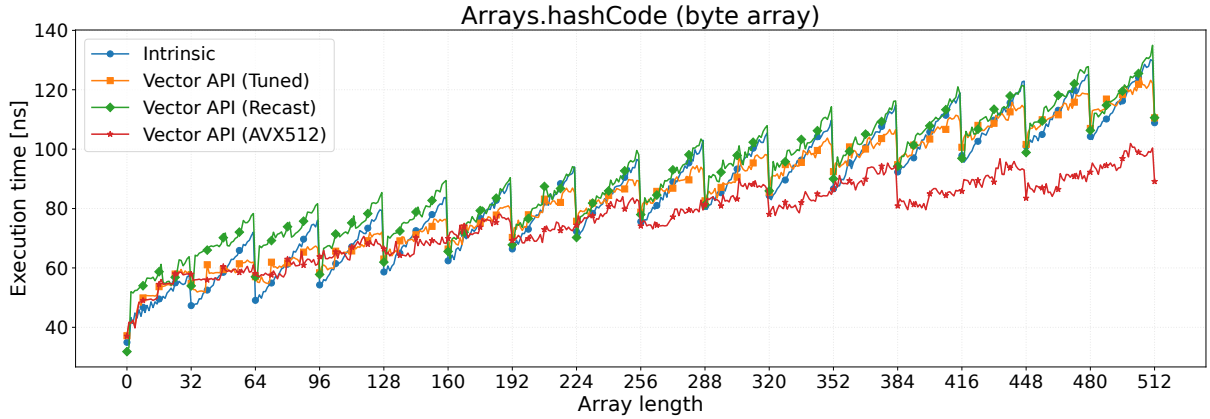


Figure 3: Performance comparison between the OpenJDK 21 `hashCode` intrinsic and our pure-Java recast and tuned implementations.

our optimization performs slightly worse than the intrinsic when there are only few residual bytes, it shows speedups when more residual bytes need to be processed.

In our experiments, we observe a different trend between the intrinsic and the recast version up to an array length of 160; beyond this threshold, the two implementations show comparable performance. This is because for longer arrays, the execution time is dominated by the loop using four accumulators, which is implemented in the same way in all versions. Compared with the intrinsic, our experimental results on this micro-benchmark show an overall speedup (geometric mean of the speedup factors for all measured lengths) of 0.94× for the recast version and of 1.10× for the tuned version.

4 VECTORIZED HASHCODE FOR LONG ARRAYS

In this section, we present and evaluate our implementation of the vectorized `hashCode` computation for long arrays. In OpenJDK 21, this computation has not (yet) been intrinsified. We describe a pure-Java vectorized implementation to show that using the JVA we can easily vectorize additional methods in the JCL that do not benefit from auto-vectorization.

Figure 4 illustrates the `hashCode` computation for long arrays. The computation makes use of the same polynomial rolling hash function we introduced in the previous section (Equation 1), with the difference that a pre-processing step is performed, consisting of an `XOR` operation on the most- and least significant part of each `long` element, yielding an integer (line 4).

4.1 HashCode Implementation for Long Arrays

We describe a new pure-Java implementation that exploits 256-bit vectors to efficiently vectorize the `hashCode` computation for long arrays. Figure 5 shows the core part of the proposed implementation.[2] As mentioned before, the interesting aspect is reading the data in a way to fully exploit the potential of vector instructions.

[2]For more details on vector operations in our pseudo-code, such as `withLane`, refer to the documentation of class `Vector` and its subclasses [26].

```
1  public static int hashCode(long[] ls) {
2    int h = 1;
3    for (long l : ls) {
4      int hash = (int)(l ^ (l >>> 32));
5      h = 31 * h + hash;
6    }
7    return h;
8  }
```

Figure 4: Pseudo-code of `hashCode` for long arrays in OpenJDK 21.

The main loop is implemented in lines 6–17. In lines 7–10, we load 4 array elements into a long vector and reinterpret it as an integer vectors of 8 elements. We perform this operation 4 times, reading 16 long elements (reinterpreted as 32 integers) in total. In lines 11–14, we 1) perform the `XOR` operation between each pair of integers originated from the same long via vector shifts, and 2) merge the results from the 4 *part* vectors into 2 *data* vectors, interleaving the results in even and odd positions. Finally, in lines 15–16, we perform the `hashCode` computation.

4.2 Evaluation of HashCode for Long Arrays

Figure 6 compares the steady-state performance of the default `hashCode` implementation in OpenJDK 21 with ours. The methodology and experimental setup are the same ones described in Section 3.2. As can be seen from the figure, our vectorized implementation significantly improves performance w.r.t. the default implementation. The performance improvement is more evident as the array length increases. Overall, experimental results on this micro-benchmark show that our JVA implementation outperforms the JCL implementation by a speedup factor (geometric mean of the speedup factors for all measured lengths) of 1.92×.

5 VECTORIZED EQUALS FOR BYTE ARRAYS

In this section, we present a vectorized implementation and evaluation of the `equals` method, which determines the equality of two arrays. It is extensively employed in applications and libraries, for

```
1  int hashCode_long(long[] ls) {
2   int len = ls.length;
3   ... // omitted code for len < 16
4   acc0 = zero(8_INT);
5   acc1 = zero(8_INT).withLane(7, 1);
6   for (int i = 0; i <= len-16; i += 16) {
7      part0 = fromArray(4_LONG,ls,i).asInt();
8      part1 = fromArray(4_LONG,ls,i+4).asInt();
9      part2 = fromArray(4_LONG,ls,i+8).asInt();
10     part3 = fromArray(4_LONG,ls,i+12).asInt();
11     data0 = part0.shiftLeft().XOR(part0)
12     .blend(part1.shiftRight().XOR(part1), mask);
13     data1 = part2.shiftLeft().XOR(part2)
14     .blend(part3.shiftRight().XOR(part3), mask);
15     acc0 = acc0.mul(POW31[16]).add(data0);
16     acc1 = acc1.mul(POW31[16]).add(data1);
17  }
18  ... // omitted code for residual (len-i) longs
19 }
```

Figure 5: Pseudo-code of the vectorized implementation of `hashCode` for long arrays using the JVA.

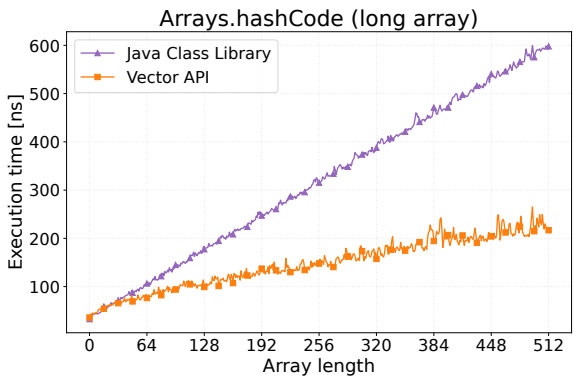


Figure 6: Performance comparison between the OpenJDK 21 `hashCode` implementation and our version using the JVA.

example, string equality is internally implemented as arrays equality. This operation is implemented as a simple loop comparing one byte in each iteration. The loop could in principle be automatically vectorized with the superword auto-vectorization of the JIT compiler. However, we highlight that our experiments show that such an optimization is not applied on `equals`: as shown in Figure 7, the intrinsic largely outperforms the pure-Java implementation. In this section, we show that an implementation based on the JVA shows performance in line with the intrinsic.

5.1 Equals Implementation for Byte Arrays

We focus on the `equals` implementation for byte arrays, where equality is determined by performing byte-to-byte comparisons. The `equals` implementation starts testing trivial cases, such as the arrays' references and their lengths. Subsequently, the implementation resorts to a byte-to-byte comparison to check whether the elements are equal, which is implemented as an intrinsic [16].

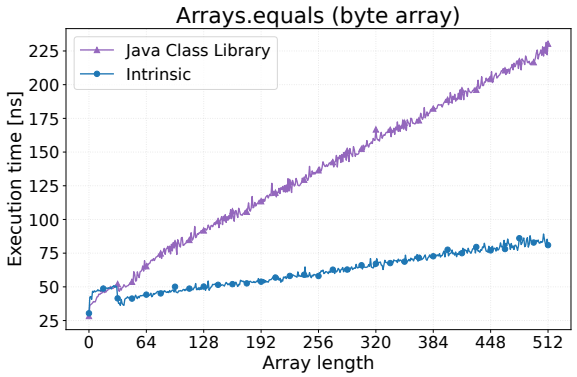


Figure 7: Performance comparison between the OpenJDK 21 `equals` intrinsic and the byte-to-byte loop in pure-Java code implemented in the Java Class Library.

```
1  int equals_recast(byte[] s1, byte[] s2) {
2   ... // check if s1 and s2 have same length
3   int len = s1.length;
4   if (len >= 64) {
5      int pos = 0;
6      do {
7        if (!(fromArray(64_BYTE, s1, pos)
8         .cmpEQ(fromArray(64_BYTE, s2, pos))
9         .allTrue()) return false;
10       pos += 64;
11     } while (pos < len-64);
12     return fromArray(64_BYTE, s1, len-64)
13       .cmpEQ(fromArray(64_BYTE, s2, len-64))
14       .allTrue();
15  } else if (len >= 32) {
16     return fromArray(32_BYTE, s1, 0)
17       .cmpEQ(fromArray(32_BYTE, s2, 0))
18       .allTrue() &&
19       fromArray(32_BYTE, s1, len-32)
20       .cmpEQ(fromArray(32_BYTE, s2, len-32))
21       .allTrue();
22  } else {
23     for (int j = 0; j < len; j++) {
24       if (s1[j] != s2[j]) {
25         return false;
26       }
27     }
28     return true;
29  }
30 }
```

Figure 8: Pseudo-code of the recast version of `equals` for byte arrays using the JVA.

Based on the original implementation of the intrinsic, we implement a recast version using the JVA (see Figure 8). As the intrinsic employs 512-bit vector instructions if available, our recast version also uses them. For arrays with lengths from 64, the implementation iterates over the arrays, comparing them in batches of 64 bytes at a time (lines 6–11). Subsequently, a final vector instruction is utilized to compare all residual bytes (lines 12–14). This vectorized comparison loads the final vector starting at position `len-64` (line 12), possibly comparing some array elements for a second time.

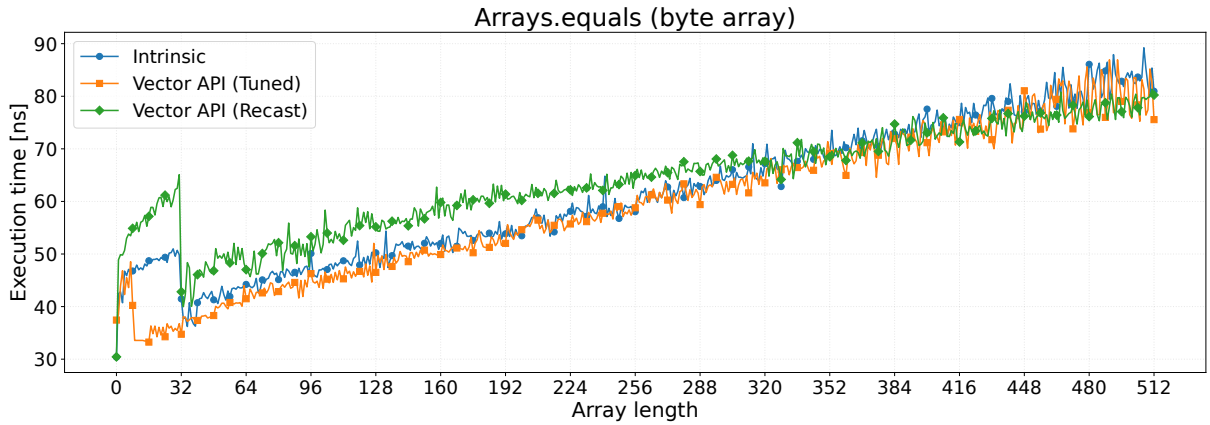


Figure 9: Performance comparison between the OpenJDK 21 `equals` intrinsic and our pure-Java recast and tuned implementations.

For arrays with lengths from 32 to 63, the implementation compares the arrays with two 256-bit vector comparisons, i.e., from position 0 to 31, and from position `len-32` to `len-1` (lines 16–21). Arrays with length < 32 are compared using scalar instructions.

We now suggest a few code improvements that could lead to better performance by exploiting vector instructions also on arrays shorter than 32 bytes. The core loop described in Figure 8, applicable for lengths ≥ 64 bytes (lines 6–11), remains unchanged. The code for lengths from 32 to 63 also remains unchanged, i.e., the two vector comparisons (32 bytes each). However, we use the scalar loop for byte-to-byte comparison only for arrays with less than 8 bytes. For arrays with lengths from 8 to 16, our implementation executes two 64-bit vector comparisons. Finally, for arrays with lengths from 17 to 31, our tuned implementation executes two 128-bit vector comparisons.

5.2 Evaluation of Equals for Byte Arrays

For evaluating `equals`, we use again the same methodology described in Section 3.2. However, since we have to compare two arrays, we initially duplicate all the byte arrays. We always compare two identical arrays, meaning that `equals` has to process all the array content to find that the arrays are equal.

In Figure 9, we compare the overall steady-state performance of our implementations (recast and tuned) with the intrinsic. As can be seen in the graph, the range of lengths where the performance difference is most visible is between 0 and 32. In this range, both the intrinsic and the recast versions use scalar computations to process short strings. The intrinsic compares multiple bytes reading integers and shorts out of the byte array, while the recast performs a byte-wise comparison, explaining their performance difference in this range. The tuned version uses scalar computation only for arrays with less than 8 bytes, and only vector instructions otherwise. Compared with the intrinsic, our experimental results on this micro-benchmark show an overall speedup (geometric mean of the speedup factors for all measured lengths) of 0.95x for the recast version, and of 1.04x for the tuned version.

6 EVALUATION ON BENCHMARK SUITES

In this section, we evaluate steady-state performance of the tuned `hashCode` and `equals` implementations using the JVA that we described in Sections 3 and 5, respectively. To modify the Java source code of the JCL, we employ the approach described later in Section 7. We conduct a performance evaluation on popular realistic benchmark suites for the JVM. We note that our goal is not to significantly outperform the (already optimized) vectorized intrinsics, but to demonstrate that the easy-to-tune pure-Java alternative, in addition to being easier to implement, test, debug, and maintain, can replace the intrinsic implementation without impairing steady-state performance.

Our evaluation considers the Renaissance [36], DaCapo [7], and ScalaBench [42] benchmark suites. For Renaissance and ScalaBench, we use the latest versions of the suites at the time of writing (Renaissance GPL v0.15.0 and ScalaBench v0.1.0). For DaCapo, we use both v9.12-Bach (released in December 2009) and v23.11-Chopin (released in November 2023). We note that the two DaCapo versions are substantially different in their workloads. We exclude benchmarks whose execution on Java 21 is not supported or with known bugs [40]. We use the *default* input size. Benchmarks can execute multiple iterations, which can either be considered as warm-up or steady-state. We run warm-up iterations until dynamic compilation and GC ergonomics are stabilized, as follows. For DaCapo and ScalaBench, we follow the approach described by Lengauer et al. [29], executing 40 warm-up iterations for each benchmark. For Renaissance, we use the same number of warm-up iterations as specified in the documentation [37]. All other iterations after warm-up are classified as steady-state. Our measurements consider only steady-state iterations, and we execute 10 such iterations for each benchmark. Finally, we run every benchmark 10 times in different JVM processes, collecting a total of 100 steady-state iterations (from 10 different JVM processes). For the measurements, we use the same experimental setup described in Section 3.2, except we use the default G1 garbage collector [11].

Table 1 reports the execution time of each benchmark when run 1) with the original vectorized intrinsic as implemented in OpenJDK 21, and 2) with our tuned implementation using the JVA.

Table 1: Performance comparison on popular benchmark suites: original OpenJDK 21 with `hashCode` and `equals` intrinsics, vs. modified OpenJDK 21 using our tuned JVA implementation.

Benchmark	Intrinsic		Vector API		
Renaissance GPL 0.15.0	**Time (ms)**	**Std. dev.**	**Time (ms)**	**Std. dev.**	**Rel. diff.**
akka-uct	6604.54	140.23	6610.62	160.85	-0.09%
als	1415.16	11.77	1416.98	13.85	-0.13%
chi-square	621.24	84.75	610.75	86.91	1.69%
dec-tree	740.63	23.93	747.20	24.21	-0.89%
dotty	767.41	15.16	766.05	16.61	0.18%
finagle-chirper	1793.15	25.29	1779.18	27.60	0.78%
finagle-http	2068.98	24.63	2112.00	15.25	-2.08%
fj-kmeans	1084.49	4.16	1136.04	9.38	-4.75%
future-genetic	1851.89	34.66	1866.27	30.41	-0.78%
gauss-mix	683.33	51.74	679.78	53.18	0.52%
log-regression	711.14	44.19	735.34	58.01	-3.40%
mnemonics	2667.95	6.47	2766.90	263.81	-3.71%
movie-lens	6478.20	50.63	6492.78	38.31	-0.23%
naive-bayes	336.51	29.15	340.15	32.34	-1.08%
neo4j-analytics	1526.52	16.99	1539.31	29.80	-0.84%
page-rank	2843.99	56.93	2866.12	52.79	-0.78%
par-mnemonics	2048.16	25.53	2080.03	70.24	-1.56%
philosophers	2793.10	99.25	2761.93	170.18	1.12%
reactors	9097.17	543.53	9117.86	497.19	-0.23%
rx-scrabble	113.89	11.09	113.73	12.20	0.14%
scala-stm-bench7	810.70	37.92	811.33	50.56	-0.08%
scrabble	87.12	9.08	86.89	8.49	0.27%
Geo. mean	**1246.00**		**1254.88**		**-0.71%**

Benchmark	Intrinsic		Vector API		
ScalaBench 0.1.0	**Time (ms)**	**Std. dev.**	**Time (ms)**	**Std. dev.**	**Rel. diff.**
apparat	5122.33	169.34	5129.38	145.43	-0.14%
factorie	10716.55	217.71	10852.47	426.89	-1.27%
kiama	222.29	17.26	222.62	18.93	-0.15%
scalac	761.20	36.63	762.31	35.45	-0.15%
scaladoc	1281.05	7.72	1269.87	27.39	0.87%
scalap	90.10	4.76	90.33	4.62	-0.26%
scalariform	323.94	18.69	327.00	19.29	-0.94%
scalaxb	180.26	6.64	180.10	6.71	0.09%
tmt	3217.71	45.83	3279.01	50.93	-1.91%
Geo. mean	**836.92**		**840.47**		**-0.42%**

Benchmark	Intrinsic		Vector API		
DaCapo 23.11-Chopin	**Time (ms)**	**Std. dev.**	**Time (ms)**	**Std. dev.**	**Rel. diff.**
avrora	3383.00	15.69	3369.85	17.59	0.39%
biojava	8109.00	124.73	8642.29	89.29	-6.58%
eclipse	12693.16	53.74	12683.34	45.49	0.08%
fop	583.33	3.00	585.46	3.41	-0.37%
graphchi	5034.84	70.58	4994.41	35.61	0.80%
h2	2679.13	38.83	2686.87	42.92	-0.29%
jme	6920.72	2.75	6924.44	2.80	-0.05%
jython	4279.78	88.23	4296.74	85.05	-0.40%
kafka	5046.25	28.54	5008.61	30.20	0.75%
luindex	4900.54	58.55	4899.83	57.11	0.01%
lusearch	3137.30	39.01	3149.17	41.59	-0.38%
pmd	1539.85	13.91	1615.78	18.57	-4.93%
spring	2382.96	183.22	2463.59	91.42	-3.38%
sunflow	3690.26	357.97	3691.84	328.14	-0.04%
xalan	589.07	5.76	601.11	6.10	-2.04%
zxing	1231.35	7.89	1246.63	9.20	-1.24%
Geo. mean	**3068.70**		**3101.97**		**-1.08%**

Benchmark	Intrinsic		Vector API		
DaCapo 9.12-Bach	**Time (ms)**	**Std. dev.**	**Time (ms)**	**Std. dev.**	**Rel. diff.**
fop	175.91	5.38	177.77	4.01	-1.06%
h2	2275.57	116.82	2282.13	120.95	-0.29%
jython	1286.18	81.78	1297.37	50.33	-0.87%
luindex	367.44	4.85	369.65	5.06	-0.60%
lusearch	109.90	7.95	110.39	7.80	-0.45%
lusearch-fix	111.75	7.12	109.86	7.90	1.69%
pmd	482.10	30.15	480.73	29.96	0.28%
sunflow	501.29	34.37	493.01	29.16	1.65%
xalan	119.72	1.63	120.96	1.41	-1.04%
Geo. mean	**343.87**		**344.11**		**-0.07%**

For fairness, since OpenJDK 21 does not use any intrinsics for `hashCode` on long arrays, we do not use the version described in Section 4 in this evaluation. The execution times reported in the table represent the arithmetic mean of 100 steady-state iterations for each benchmark. We also report the standard deviation, the relative difference of the execution time of the JVA implementation w.r.t. the intrinsic (i.e., the difference between the execution time of the intrinsic and the one of the JVA implementation, divided by the one of the intrinsic) and the overall per-suite geometric-mean execution times and relative difference.

As one can see from the table, our tuned implementation using the JVA does not impair steady-state performance w.r.t. the intrinsic in the evaluated benchmarks. Overall, our version results in very similar steady-state performance to the vectorized intrinsic. Considering the average (geometric mean) of all benchmarks in a suite, our version results in a relative difference of -0.71% (Renaissance), -0.42% (ScalaBench), -1.08% (DaCapo-Chopin) and -0.07% (DaCapo-Bach). Considering all benchmarks, the average execution time (geometric mean) relative difference is 0.67%.

Overall, our implementations provide similar steady-state performance than the vectorized intrinsics, but with the benefit of being written purely in Java, being much more portable, easier to understand, maintain, and fine-tune. In addition, our approach avoids writing platform-specific code to exploit particular vector extensions a processor may support—the JVA automatically uses the vector instructions supported by the underlying architecture.

7 MODIFYING CORE CLASSES IN THE JCL

In this section, we detail our approach to modify the Java source code of the JCL, such that our Java implementations making use of

the JVA can substitute the use of intrinsics in core classes of the JCL (particularly in class `Arrays`). Even though the JCL is implemented as part of the JVM, our approach does not require one to modify the native code of the JVM and hence can be exploited by developers without the need to recompile the JVM. We note that our approach is fully compliant with the JVM specification. We describe our approach in the context of OpenJDK 21 and we remark that we evaluated our approach on state-of-the-art benchmark suites in Section 6.

Modifying core classes in the JCL (such as `Arrays`) is challenging [6, 33, 39, 41] since during the early phases of JVM initialization, many Java features (including the JVA, as well as e.g. the Java Reflection API [34]) cannot be used. Modifying JCL methods (such as `Arrays.hashCode` and `Arrays.equals`) to use such features will cause JVM crashes. The reason is that modified JCL classes may alter the order in which classes are initialized, leading to premature class initializations. Moreover, when modifying methods in the JCL classes, it is crucial to avoid cyclic dependencies that would lead to an infinite recursion in class initialization. This behaviour is particularly subtle because a modified method may trigger the initialization of some classes, and the initializers of these classes may use (directly or indirectly) the modified method.

To ensure proper JVM initialization, our approach introduces *initialization guards* and leverages class redefinition. In particular, our modified JCL methods make use of a custom `JVMInitialization` class that allows checking whether JVM initialization has completed via its `isInitialized` static method, which returns a boolean flag. Below, we describe how our approach enables the use the JVA in JCL core classes.

7.1 Modified JCL Methods and Initialization Guards

We modify methods in the JCL by inserting our implementation into the body of the original method implementation, guarded by a conditional invoking the `JVMInitialization.isInitialized` method. If this method returns `true`, our implementation (using the JVA) is executed. Otherwise, the original JCL implementation is used. Thanks to our guard (that checks whether JVM initialization has completed) and because the JVM ensures lazy class initialization upon the first use of a class [32], the classes used by our implementation (i.e., the classes of the JVA) will not be initialized during JVM initialization.

7.2 Class JVMInitialization

Since Java does not expose an interface to check whether JVM initialization has completed, we implement the static `JVMInitialization .isInitialized` method, which returns a boolean flag that is toggled right after JVM initialization. This method could be trivially implemented by storing the boolean flag in a volatile static field and by setting this field to `true` in the beginning of an application's `main` method. However, this incurs a serious performance issue: the cost of a volatile read upon each invocation of the `isInitialized` method can be significant and jeopardize the optimizations introduced by our implementations.

For this reason, we implement the flag as a static `isInit- ialized` method that initially returns the boolean constant `false`, and we use a Java agent [15] to redefine the `isInit- ialized` method to return `true` upon the execution of the agent's `premain` method (i.e., when the JVM is ready to execute arbitrary Java code). This strategy allows exploiting the branch-elimination optimization of the JIT compiler, improving performance even further. From the JIT-compiler perspective, the static `isInitialized` method returns a constant and hence only one of either our implementation or the original JCL implementation will be executed. The compiler will therefore perform branch elimination and remove the code of the implementation that will never be executed, increasing the code size budget for other optimizations, such as method inlining. In the (unlikely) case that the JIT compiler already optimized the modified method during JVM initialization, before our Java agent redefines the `isInitialized` method, we rely on OpenJDK's deoptimization feature to ensure the correctness of our solution. Moreover, the JIT compiler in OpenJDK will re-compile and optimize hot code after our class redefinition, ensuring that our initialization guards incur no overhead in steady state.

Finally, to avoid cyclic dependencies that would lead to infinite recursions in class initialization, we make the Java agent (which is allowed to execute arbitrary Java code) initialize all the classes used by our implementations before redefining the `isInitialized` method.

8 RELATED WORK

In this section, we discuss related work. We first discuss techniques to improve performance of high-level and managed languages (Section 8.1). Then, we detail benchmarks for vector computations in high-level languages (Section 8.2).

8.1 Portability without Sacrificing Performance

Substantial research effort has been made with the goal of proposing portable and high-level programming languages, domain-specific languages (DSL), and libraries without sacrificing performance [3, 48]. As an example, in the "abstraction without overhead" line of work, [9, 44, 45] show that by leveraging staged compilation [38], the cost of abstractions can be removed at compilation time, allowing developers to implement high-performance applications in high-level languages. Within this line of work, an alternative vector API for Scala and Java has been proposed by Stojanov et al. [43] to express vector computations with a high-level DSL, showing that an implementation leveraging the proposed DSL for expressing vector operations can outperform pure Java code that relies on auto-vectorization. In contrast, we evaluate our implementations based on the JVA against vectorized intrinsics, which are not the result of automatic program transformation, but fine-tuned implementations carefully written by expert JVM developers.

Generally, the need for expressing low-level operations within high-level programming languages has been discussed and motivated by Frampton et al. [12]. This work proposes techniques to safely integrate low-level components within high-level languages, as done in the JVA. Similarly, we demonstrate that the JVA allows optimizing the JCL without the burden of writing intrinsics.

8.2 Benchmarking Vector Computations on Managed Runtimes

Basso et al. propose JVBench [5], a benchmark suite for the JVA, showing that by leveraging the API, developers can write applications that result in higher performance with respect to auto-vectorization. The Swan benchmark suite [27] has been proposed for benchmarking vectorized operations in the context of mobile applications. In contrast to these benchmark suites, we analyze the performance of the JVA against fine-tuned vectorized intrinsics. To the best of our knowledge, our work is the first to demonstrate that the JVA can be used to easily optimize core JCL classes without the burden of writing vectorized intrinsics.

9 LIMITATIONS

A limitation of our work is that, while the most popular architectures and JVMs support the JVA, not all of them fully support the API. Nonetheless, this API exhibits continuous performance improvements in subsequent versions and enables higher performance than relying on the JVM's limited auto-vectorization capabilities [5]. Another limitation is that our evaluation was conducted on a single machine and architecture (x86). We plan to experiment more extensively on a wide range of architectures.

By delegating the generation of vector instructions to the JVA, the code (if hot) will be processed by the JIT compiler, which may apply decisions resulting in suboptimal performance. In contrast, in OpenJDK 21 the code specified in the intrinsic is emitted without undergoing further optimizations, giving more control to the developers.

Finally, using the JVA incurs extra JIT-compilation costs, while machine-code generation for an intrinsic may be faster. This may impair the performance of short-running applications, as steady-state performance may be reached later in the application's execution. As part of our ongoing research, we are investigating whether our approach negatively affects startup performance or not.

10 CONCLUSIONS

In this paper, we support the claim that replacing platform-dependent template-generated assembly code implemented via vectorized intrinsics with equivalent, portable, pure-Java code using the JVA does not impair steady-state performance, making the approach attractive even in production-level JVMs. We show this by replacing the vectorized intrinsics of `hashCode` and `equals` for byte arrays with equivalent Java code. Moreover, in addition to reducing the effort in understanding, extending, debugging, testing, and maintaining the code, the resulting code is much easier to fine-tune to further improve performance. We propose code improvements for `hashCode` and `equals`, where tuning is based on the array length; such tuning would be difficult to implement in vectorized intrinsics. Furthermore, we proposed a new vectorized `hashCode` computation for long arrays, for which a corresponding intrinsic is missing in OpenJDK 21.

Our evaluation shows that our tuned implementations provide similar performance than the vectorized intrinsics on four popular and realistic benchmark suites. Finally, we present a technique that can be adopted by developers to modify core classes of the JCL without disrupting JVM initialization; our technique does not require any changes to the native codebase of OpenJDK.

The findings in this work highlight the potential of the JVA as a viable alternative for vectorizing other compute-intensive methods in the JCL using only platform-independent Java code, avoiding writing complex platform-dependent native code that is hard to understand, extend, maintain, debug, and test.

As part of our future work, in addition to tackling the limitations discussed in Section 9, we plan to expand our use cases, replacing vectorized intrinsics in other commonly executed methods in the JCL with pure-Java code using the JVA, evaluating the performance improvements. We also plan to identify methods that could benefit from vectorization via the JVA and for which a vectorized intrinsic is currently missing. Finally, we will evaluate more metrics (e.g., energy consumption) and hardware architectures (e.g., ARM).

ACKNOWLEDGEMENTS

The research presented in this paper was supported by Oracle (ERO project 1332) and the Swiss National Science Foundation (project 200020_188688).

REFERENCES

[1] 2013. Intel Xeon Phi Coprocessor High Performance Programming. Morgan Kaufmann.
[2] 2015. Software Analysis and Optimization. In *Power and Performance*, Jim Kukunas (Ed.). Morgan Kaufmann.
[3] Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417.
[4] Andrew Binstock. 2019. Epsilon: The JDK's Do-Nothing Garbage Collector. https://blogs.oracle.com/javamagazine/post/epsilon-the-jdks-do-nothing-garbage-collector.
[5] Matteo Basso, Andrea Rosà, Luca Omini, and Walter Binder. 2023. Java Vector API: Benchmarking and Performance Analysis. In *CC*. 1–12.
[6] Walter Binder, Philippe Moret, Éric Tanter, and Danilo Ansaloni. 2016. Polymorphic Bytecode Instrumentation. *Softw. Pract. Exper.* 46, 10 (2016), 1351–1380.
[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*. 169–190.
[8] Brian Goetz. 2019. dont intrinsify Objects::hash. https://mail.openjdk.org/pipermail/amber-dev/2019-April/004264.html.
[9] Kevin J Brown, Arvind K Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 89–100.
[10] Maximilian Böther, Lawrence Benson, Ana Klimovic, and Tilmann Rabl. 2023. Analyzing Vectorized Hash Tables Across CPU Architectures. *Proceedings of the VLDB Endowment* 16 (8 2023), 2755–2768. Issue 11.
[11] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management*. 37–48.
[12] Daniel Frampton, Stephen M Blackburn, Perry Cheng, Robin J Garner, David Grove, J Eliot B Moss, and Sergey I Salishev. 2009. Demystifying Magic: High-Level Low-Level Programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 81–90.
[13] GitHub. 2023. 8282664: Unroll by hand StringUTF16 and StringLatin1 polynomial hash ... · openjdk/jdk21u@e37078f<. https://github.com/openjdk/jdk21u/commit/e37078f5bb626c7ce0348a38bb86ca2ca62ba915.
[14] GitHub. 2023. jdk21. https://github.com/openjdk/jdk21.
[15] GitHub. 2023. Package java.lang.instrument. https://docs.oracle.com/en/java/javase/21/docs/api/java.instrument/java/lang/instrument/package-summary.html.
[16] GitHub. 2023. Source code of arrays_equals in c2_MacroAssembler_x86.cpp. https://github.com/openjdk/jdk21/blob/

890adb6410dab4606a4f26a942aed02fb2f55387/src/hotspot/cpu/x86/c2_MacroAssembler_x86.cpp#L4086.
[17] GitHub. 2023. Source code of arrays_hashcode in c2_MacroAssembler_x86.cpp. https://github.com/openjdk/jdk21/blob/890adb6410dab4606a4f26a942aed02fb2f55387/src/hotspot/cpu/x86/c2_MacroAssembler_x86.cpp#L3285.
[18] GitHub. 2023. Source code of java.util.zip.ZipCoder. https://github.com/openjdk/jdk21u/blob/89aea0dace4afd2b1bbc5d636322969afcf0072c/src/java.base/share/classes/java/util/zip/ZipCoder.java#L294.
[19] GitHub. 2023. Source code of vmIntrinsics.hpp. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp.
[20] GitHub. 2023. Source code of vmIntrinsics.hpp, comment on bytecode instrinsics. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp#L78.
[21] GitHub. 2023. Source code of vmIntrinsics.hpp, comment on instrinsics. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp#L72.
[22] GitHub. 2023. Source code of vmIntrinsics.hpp, comment on library instrinsics. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp#L74.
[23] Tobias Groth, Sven Groppe, Thilo Pionteck, Franz Valdiek, and Martin Koppehel. 2022. Accelerated Parallel Hybrid GPU/CPU Hash Table Queries with String Keys. In *Database and Expert Systems Applications*. 191–203.
[24] Intel. 2007. Intel SSE4 Programming Reference. https://www.intel.com/content/dam/develop/external/us/en/documents/\d9156103-138479.pdf.
[25] Intel. 2015. Intel Advanced Vector Extensions 512. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.
[26] Java Platform, Standard Edition & Java Development Kit – Version 21 API Specification. 2023. Class Vector. https://docs.oracle.com/en/java/javase/21/docs/api/jdk.incubator.vector/jdk/incubator/vector/Vector.html.
[27] Alireza Khadem, Daichi Fujiki, Nishil Talati, Scott Mahlke, and Reetuparna Das. 2023. Vector-Processing for Mobile Devices: Benchmark and Analysis. In *2023 IEEE International Symposium on Workload Characterization*. 15–27.
[28] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc.
[29] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 3–14.
[30] Chung Yu Liao and Cheng Hung Lin. 2017. A Novel Parallel Dual-character String Matching Algorithm on Graphical Processing Units. In *ICA3PP*. 197–210.
[31] Cheng-Hung Lin, Jin-Cheng Li, Chen-Hsiung Liu, and Shih-Chieh Chang. 2017. Perfect Hashing Based Parallel Algorithms for Multiple String Matching on Graphic Processing Units. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2639–2650.
[32] Tim Lindholm, Frank Tellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2023. The Java Virtual Machine Specification – Java SE 20 Edition – Chapter 5. Loading, Linking, and Initializing. https://docs.oracle.com/javase/specs/jvms/se20/html/jvms-5.html#jvms-5.5.
[33] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-Specific Language for Bytecode Instrumentation. In *AOSD*. 239–250.
[34] Oracle. 2023. Java Reflection API. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html.
[35] Oracle Corporation. 2023. JEP 448: Vector API (Sixth Incubator). https://openjdk.org/jeps/448.
[36] Prokopec, A. and Rosà, A. and Leopoldseder, D. and Duboscq, G. and Tůma, P. and Studener, M. and Bulej, L. and Zheng, Y. and Villazón, A. and Simon, D. and Würthinger, T. and Binder, W. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *PLDI*. 31–47.
[37] Renaissance Suite. 2019. Renaissance Suite - Documentation. https://renaissance.dev/docs.
[38] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 127–136.
[39] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2017. Accurate Reification of Complete Supertype Information for Dynamic Analysis on the JVM. In *GPCE*. 104–116.
[40] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2019. Analysis and Optimization of Task Granularity on the Java Virtual Machine. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 19 (jul 2019), 47 pages.
[41] Andrea Rosà and Walter Binder. 2018. Optimizing Type-specific Instrumentation on the JVM with Reflective Supertype Information. *Journal of Visual Languages & Computing* 49 (2018), 29–45.
[42] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 657–676.
[43] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. 2018. SIMD intrinsics on managed language runtimes. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2–15.
[44] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 609–616.
[45] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–25.
[46] Tran Ngoc Thinh, Surin Kittitornkun, and Shigenori Tomiyama. 2007. Applying Cuckoo Hashing for FPGA-based Pattern Matching in NIDS/NIPS. In *FPT*. 121–128.
[47] Kuo-Kun Tseng, Ying-Dar Lin, Tsern-Huei Lee, and Yuan-Cheng Lai. 2005. A Parallel Automaton String Matching with Pre-hashing and Root-indexing Techniques for Content Filtering Coprocessor. In *ASAP*. 113–118.
[48] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (jan 2013), 24 pages.
[49] Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. 2020. SAHA: A String Adaptive Hash Table for Analytical Databases. *Applied Sciences* 10, 6 (2020), 1–18.
[50] Zoltán Majó. 2016. C1 arraycopy intrinsic type checks missing. https://mail.openjdk.org/pipermail/hotspot-compiler-dev/2016-June/023527.html.

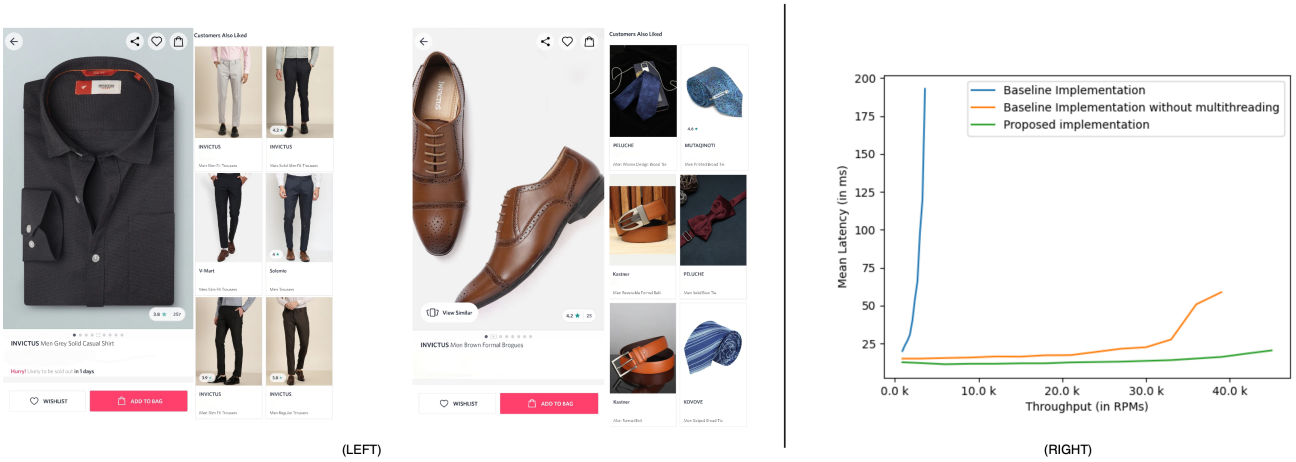
Rethinking 'Complement' Recommendations at Scale with SIMD

Shrey Pandey
shreypandey1509@gmail.com
Myntra Designs Pvt. Ltd.
Data Science
Bangalore, India

Saikat Kumar Das
saikat.kumar@myntra.com
Myntra Designs Pvt. Ltd.
Data Science
Bangalore, India

Hrishikesh V. Ganu*
hrishikeshvganu@gmail.com
Independent
Bangalore, India

Satyajeet Singh
satyajeet.singh@myntra.com
Myntra Designs Pvt. Ltd.
Data Science
Bangalore, India



(LEFT) (RIGHT)

Figure 1: 'Complement' Recommendations service in Myntra. Left: Primary product Added-to-Cart by user and corresponding 'complement' recommendations to increase discovery and item count in cart. Right: Comparative graph depicting the improvement in service scalability due to proposed vectorized reformulation of recommendation components.

ABSTRACT

Maximizing cart value by increasing the number of items in electronic carts is one of the key strategies adopted by e-commerce platforms for optimal conversion of positive user intent during an online shopping session. Recommender systems play a key-role in suggesting personalized candidate items that can be added to cart by the user. However, it is important to serve a diverse set of personalized recommendations that 'complement' user's cart content to practically increase item count in cart and also contribute towards product discovery. Borrowed from Quantum Physics, Determinantal Point Processes (DPP) are used widely in recommender systems to diversify personalized product recommendations for improved user engagement. However, vertically scaling DPP for recommendation sets, personalized with vector similarity metric like cosine similarity, to serve large scale real-time concurrent user requests is non-trivial. We propose a vectorized reformulation of cosine similarity and conditional DPP implementation to best utilize the highly improved vector computation capabilities (SIMD) of modern processors. Experimental evidence on real-world traffic shows that the proposed method can handle upto 15x more concurrent traffic while improving latency. The proposed method also uses portable SIMD constructs from Python libraries which can be easily adopted in most available SIMD supported CPUs with minimal code changes.

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; • **Applied computing** → **Online shopping**; • **Computing methodologies** → **Linear algebra algorithms**.

*work was done while affiliated to Myntra Designs Pvt. Ltd.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05…$15.00
https://doi.org/10.1145/3629526.3645041

KEYWORDS

Vectorization, SIMD, Diversification, Recommender Systems, Recommendations, Determinantal Point Processes

ACM Reference Format:

Shrey Pandey, Saikat Kumar Das, Hrishikesh V. Ganu, and Satyajeet Singh. 2024. Rethinking 'Complement' Recommendations at Scale with SIMD. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645041

1 INTRODUCTION

'Add-To-Cart' (ATC) is a key event in a user session on e-commerce platforms. ATC indicates positive intent from user to complete a purchase from current session. However, this also presents a unique opportunity for the platform to improve cart value by increasing the number of items in cart and optimize conversion, generally referred to as *cart completion* or *cart filling*. Cart completion has been a subject of extensive research for decades, evolving continually to cater to changing customer behaviour and research community's understanding of the same. Early works on cart completion works on the principle of finding likely items to be added to cart based on previous co-click events [4] and browsing history from a cohort of similar users [48]. Though effective, these approaches do not differentiate browsing patterns and user motivation based on occurrence of key-events like ATC or 'Wish-listing'. Close *et al.* [15] conducted a detailed study for understanding user behaviour patterns related to the usage of electronic carts. Building on the broader patterns of user motivation of adding items in cart beyond immediate purchase as outlined by the researchers, we can identify scope of optimizing conversion by increasing the number of items in cart. Mcauley *et al.* [38] explored the concept of 'complement' and 'substitute' in their work, where they claim complements are items that can be bought alongside items already in cart and substitutes are used for replacing current cart items. For users looking to take advantage of price promotion or browsing for entertainment [15], recommending similar items or 'substitutes' to those already in cart can be detrimental to the eventual cart value. In the first case, user looking for price promotions may chose to replace a cart item with a new item of lower or same value, which may not lead to optimizing conversion. In the second case, users looking for entertainment may find it monotonous to keep browsing similar items which they have already explored. Apart from this, for users with immediate purchase intent, substitute recommendations from their cart can induce doubts about their selection, which may lead to session abandonment. Similar item recommendations can be useful to users looking for research or information search, but studies [15] have shown that purchase probability for such sessions are much lower than the cases mentioned above.

Thus, to optimize conversion through increasing items in cart, diversification of personalized recommendations to suggest 'complements' based on cart-items appears to be the better strategy as it complements user selections rather than competing. However naive diversification of personalized recommendations can lead to degrading quality of suggestion. Diversification should be closely aligned with user preferences and user journey on platform along with cart-items. Determinantal Point Process [6, 21, 30] is a popular method for diversifying personalized recommendations, personalized with vector similarity metric like cosine similarity. However, one practical consideration in using automated recommendation system on e-commerce platform [1, 11, 16, 29, 43, 63] is the scale and concurrency it has to handle in real-time shopping scenarios. Online recommendation systems can receive millions of requests per minute (RPM) on high traffic event days. Moreover, most e-commerce platform will host millions of items in their catalogue to serve millions of customers. Thus, it is imperative to build a system that can handle large scale traffic of few million RPM while also being flexible enough to be deployable without costly hardware requirements. Cosine similarity and DPP [59], although being an excellent algorithms to consider diversification of personalized recommendations, but is not scalable vertically, especially when deployed on CPU-only cloud servers. Tensorized versions of DPP has been proposed [58] to harness GPU powers in-order to provide higher order scalability but as per our knowledge, re-thinking DPP formulation and Cosine similarity implementation to best utilize the untapped compute capacity of SIMD (Single Instruction Multiple Data) enabled modern CPUs has not been explored previously in literature. To this end, we propose a reformulation of the DPP algorithm and vectorized implementation of Cosine Similarity, tailored for SIMD acceleration on CPU that is highly scalable yet portable across various available server CPUs (Sec. 3). Proposed, reformulated DPP reduces number of computation involving N candidate recommendations and k items from $O(x) + O(N^2)$ to $O(x) + O(N \times k))$, where $N >> k$ and $N^2 > x > N^3$; x being the number of computations for matrix multiplication.

Our proposed diversifying recommendation system (Fig. 1 (Left)) enabled us to reduce hardware requirement by nearly 12 times in production environment (Fig. 1 (Right)) while also improving the top 99th-percentile (P99) latency under the maximum throughput by 13 times (realtime traffic observations discussed in Sec. 5). Our key contributions in this work are

- Evaluation of parallel processing strategies for CPU for real-time concurrent services on e-commerce platform.
- Vectorized implementation of cosine similarity used for user personalization of recommendations that can leverage the superior SIMD compute capability of modern CPUs.
- Reformulation and vectorization of DPP, commonly used for diversification of recommendations, to best utilize the gain in efficiency with SIMD.
- A portable implementation of vectorized SIMD acceleration using Python libraries that can be migrated amongst various CPUs and architectures with minimal to no code changes.
- Provide long-term scalability solution to recommendation pipelines through SIMD vector optimization instead of parallel processing.

2 RELATED WORK

Recommendation system has been an active area of research for decades with ever evolving requirements to filter out information overload and generate optimal set of recommendations for consumers. Early works in recommendation systems started with finding correlations of amongst items in large scale databases from browsing sequences of users [4]. The widely studied *'GroupLens'*

project [27, 28, 46] introduced collaborative information filtering in recommendation systems inspired from the real world news groups. Sarwar *et al.* [48] proposed to use item based collaborative filtering in-place of the conventional user centric filtering, while Shahabi *et al.* [49] introduced large scale recommendation system by augmenting collaborative filtering with content-based querying. Amazon also published their report [34] on personalizing shopping with item-to-item collaborative filtering using co-browsing data in 2003. Hijikata *et al.* [23] proposed to use discovery driven collaborative filtering that for each user recommends undiscovered items with higher probability while personalizing the ranking based on user's discovery till that point of time.

On the other hand, the importance of diversity and maximizing representation of the entire collection in the recommended subset gained traction almost at the same time as well. Ko *et al.* [26] proposed to select the most informative subset of recommendations by maximizing entropy of the set. Adomavicius *et al.* [1] use multicriteria ranking to provide better and diverse recommendations. Carbinell *et al.* [7] proposed re-ranking of document search results to maximize information retrieved. Diversification of search results and its impact on information filtering was studied widely in the early 2010s [3, 9, 14, 17, 18, 22, 45, 47]. The impact of diversification on auto recommendations was also studied widely at the same time [55, 61, 62, 64]. Lathia *et al.* [33] studied relevance of temporal diversity in recommendations. Chapelle *et al.* [10] explored combining diversification objective with intent based metric ERR-IA to evaluate relevance and diversity together. Genre and category based diversification was also explored for generation better recommendations [43, 54]. Diversification also played a important part in improving relevance of video recommendations. Covington *et al.* [16] use a deep-learning based recommendation model for generating diverse recommendations. While, Mark *et al.* [59] used DPP for personalized diversification of video search results in Youtube. Another widely adopted diversification strategy are the submodular functions [42] which operates on the principle of maximizing information from a subset. Researcher's in [41, 51] use submodular functions to diversify recommendations in e-commerce.

Lot of research has been also done to better understand and interpret user-behaviour and inferred signals in e-commerce shopping. McNee *et al.* [39] attempted to establish a user-centric metric with informal arguments to better evaluate recommendation quality. Close and Kukar-Kinney [15] attempted to breakdown and analyse various user motivation behind the use of electronic carts. In their work they identify motivations such as entertainment, research also to play a major role apart from usual factors like price promotions and immediate purchase. The data evidence backed hypothesis from this work helped to identify opportunities to increase cart items and in effect the gross revenue. Hohnhold *et al.* [24] proposed to focus on long term user retention rather than short-term gains. Mcauley *et al.* [38] explored item relations based on substitutes and complements, here substitutes refer to products which can replace the item already in cart and complements are products which can be bought along with the item already planned for purchase. Zheng *et al.* [63] also explore substitute and complement relations between items to enrich recommendations list.

Determinantal Point Processes originated in Quantum Physics [6, 25, 35] and are a natural choice for modelling informative subset selection problem like document summarization, diversifying search and recommendations. Kulesza *et al.* [31] parameterized conditional DPP and attempted to learn DPPs from the resulting convex and tractable learning formulation. Kulesza *et al.* also proposed to improve the efficiency of DPP computation with the fixed size k-DPP formulation [30] and also advocated for using DPPs over MRF in ML tasks mentioned above for better tractability of the former. Considerable effort has been invested collectively to improve the efficiency of the DPP computation [5, 19, 20, 30, 36, 37]. Chen *et al.* [11] used DPP to improve recommendations with diversity. Mark *et al.* [59] also used DPP to generate diverse video recommendations for Youtube. However, reformulating DPP to reduce complexity of the algorithm without compromising on precision or exact solution has not been explored in-depth in literature. In our work, we introduce a reformulated personalization and DPP based diversification process, highly scalable for online recommendations by using vectorized computations efficiently. Proposed recommendation pipeline is able to serve $15x$ more traffic than the existing solutions without comprising on the quality of the results.

3 METHOD

For a given query product p_q and user u_q, Yan *et al.* [60] and Agarwal *et al.* [2] described the process of calculating relevant cross-category or 'complement' recommendations. A product-embedding vector e represents a product p present in the product catalogue $\mathbb{C}$ of millions of products. Similarly, an user-embedding vector v represents a user u of the e-commerce platform in vector space. Every embedding vector is part of a k-dimensional vector space $\mathcal{V}$ stored in the form of a one-dimensional array in computer memory (Eq. 1).

$$e, v \in \mathcal{V} \subset \mathbb{R}^k \tag{1}$$

For a query product p_q with product embedding e_q , a set of N products containing the relevant cross-category recommended products is generated using the product-embedding vectors e in $\mathcal{V}$. This set which acts as a candidate set of recommendations, is called the recall set C for the query product p_q. The sequence of products in the recall set is not personalized according to the user preference as it does not consider user-embedding vectors. It is essential to suggest products in a personalized sequence tailored to user preference to increase the relevance of the recommendations. Each product in the recall set, represented by $p_i \; \forall i \in [1..N]$, is assigned a user relevance score s_i, using user-embedding vector v_q of user u_q , and the product embedding e_i to personalize the product recommendations. The *personalized* recall set is then re-ranked by Determinantal Point Process (DPP) [35], which uses the relevance scores and product-embeddings for diversifying the recommendations. The re-ranked set of products are the recommended 'complements' of the product p_q, personalized to user u_q.

Fig. 2 shows the workflow for the cross-category-related product recommendations. The recommendation workflow is broadly divided into three steps:- (1) Recall set generation, (2) Personalised scoring, and (3) Diversified re-ranking.

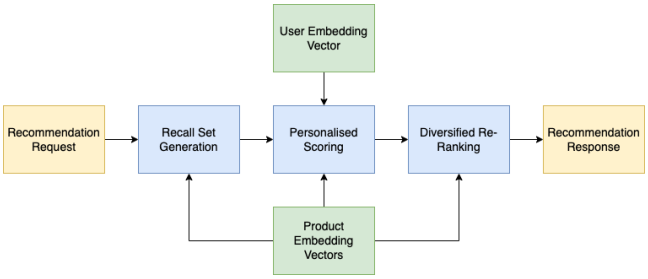


Figure 2: 'Complement' recommendation service workflow

3.1 Recall Set Generation

This step acts as a filter on a catalogue of millions of products. It filters out the less relevant products to generate the the recall set C, a set of the most relevant N products to the query product p_q. This step calculates the approximate nearest neighbours (ANN) of the query product p_q in the vector space $\mathcal{V}$ using the product-embedding vectors e_q of the query product and remaining products $e_l \forall\, l \in [1..\mathcal{N}_{\mathbb{C}}]$ in the catalogue $\mathbb{C}$.

$$sim(g, h) = \sum_{i=1}^{N} g_i \times h_i \tag{2}$$

The cosine similarity distance Eq. 2 between the embedding vectors is the metric for the relevance between two products. Distance between the embedding vector e_q and embedding vectors $e_i \forall i \in [1...N]$ of the products p_i in the recall set C gives the style relevance score r_i for the recommendation as shown in Eq. 3.

$$\forall \{i \in \mathbb{N}, i \leq N\},\ r_i = \sum_{j=1}^{k} e_{qj} \times e_{ij} \tag{3}$$

User embedding vector v_q depends on the real time in-session data and user browsing behaviour. Thus, it is expected to evolve with each action in a browsing session. Product attributes remain mostly unchanged over their lifetime, barring few exceptions. This makes user embeddings more volatile than product embeddings. This property allows us to calculate the recall set of products, offline, once a day without risking the consistency of the pipeline. Recall set is stored in appropriate data-stores like Aerospike [50] and Redis [8]. During online recommendations, recall set C for the query product p_q is fetched from the corresponding data-store and passed into next step of recommendation generation pipeline.

3.2 Personalised Scoring

The Recall set C generated till now is non-personalized. For personalised recommendations, the candidate recommendations in C should be re-scored, taking user affinity into account. The user-personalized score of a product in the recall set is the distance of the user embedding vector and product embedding vector in the k-dimensional space. A combination of the user-personalised score with the style relevance score of each product acts as input for re-ranking the recommendations.

Consider, v is the user embedding vector, V is the row-major matrix of size $N \times k$ with each row representing product embedding e_i for each product in recall set C *i.e.* candidate recommendation p_i . S is the vector containing user personalised score for every candidate recommendation. User-personalized score s_i of product p_i in the recall set C can be computed by Eq. 4.

$$\forall \{i \in \mathbb{N}, i \leq N\},\ s_i = \sum_{j=1}^{k} v_j \times V_{ij} \tag{4}$$

User and Product embedding are normalized, i.e $\|v\| = 1$ and $\|e_i\| = 1, \forall\, i \in [1..N]$. The baseline algorithmic implementation is described in Alg.1

Algorithm 1 Baseline implementation of cosine similarity

Require: (u, V)
Ensure: *user personalisation scores* $= S$
  $S \leftarrow \{0.0, 0.0, 0.0....0.0\}_N$
  $i \leftarrow 1$
  while $i \leq N$ **do**
    $score \leftarrow 0.0$
    $j \leftarrow 1$
    while $j \leq k$ **do**
      $X \leftarrow u[j] \times V[i][j]$
      $score \leftarrow score + X$
      $j \leftarrow j + 1$
    end while
    $S[i] \leftarrow score$
    $i \leftarrow i + 1$
  end while

Since each iteration of outer loop in the baseline implementation as mentioned in Alg. 1 calculates score s_i for each product p_i, every iteration of outer loop is independent of each other and can be computed in parallel. This limits the optimal efficiency of the baseline implementation as it works on one vector element at a time, increasing the latency of the cosine similarity calculation. For optimising the latency of the user-personalised score calculation, parallelism should be incorporated with the goal of the minimum overhead in data transfer, data copy, context switching and cache miss. Generally, parallelization is incorporated in the following fashion:-

3.2.1 Multithreading. This method of parallelization is helpful in case CPU cores are under-utilised and idle most of the time to share the compute-heavy work and reduce the latency. In a user-facing online system, thousands of concurrent users request recommendations at a given time, so CPU cores are occupied most of the time to compute recommendations for the users, and there are no free cores to incorporate more threads. Spawning more threads will increase the load average of the system due to increase in number of waiting threads. This will lead to frequent context switching due to OS scheduler which adds the extra overhead latency of context switching, thread creation and cache miss. This leads to the scenario that the amount of time spent on context switching can exceed the amount of time spent on computation. This behaviour suggests that multi-threading will not reduce the latency and cannot scale well for online system involving compute intensive machine learning workloads. Efficient use of CPU cores is the key to increasing the throughput, latency per request and performance.

3.2.2 Register level parallelism/SIMD. Modern processors usually have a wider register length which can consume more data in a single instruction cycle. The proposed method is efficient when the same operation is performed on multiple continuous and independent data, and the result is stored in multiple independent yet continuous memory locations. Thus, enabling parallel computations with added advantage of more cache hits results in efficiently generating user-personalised scores.

3.2.3 Proposed Cosine Similarity. Since cosine similarity calculation is a reduction operation, in which dependent data is continuous in computer memory, the implementation mentioned in Alg.1 cannot leverage register level parallelism. We propose a batch cosine similarity operation to calculate similarity between user u with N products in recall set C that is hardware-agnostic, optimized and can be easily implemented in any high level programming language like Python [53]. It is implemented according to instruction set architecture, processors vector length, cache length and scope, temporal and spatial locality of data. It is powered by the SIMD (or vector parallelism) which performs 216× better than baseline implementation.

The column-major matrix $\hat{V}$ ($\hat{V} = V^T$, V is the original product embedding matrix V) represents the product embeddings, where i^{th} column contains the embedding e_i for the product p_i in the candidate set C and j^{th} row represents the j^{th} feature element in the embedding of every product in C, *i.e.* $\hat{V}_{ij}$ represents j^{th} feature element of the i^{th} product in C. Let us consider Intel® Xeon® Platinum 8171M CPU @ 2.60GHz processor as the processing unit. It has a maximum register length of 512 bits, implying it can consume 16 consecutive 32-bit floating point numbers in a single instruction cycle, essentially processing 16 elements in parallel. In addition to wide register length, it also has the functionality of Fused Multiply Add (FMA) in the instruction set, which allows multiplication and addition in a single clock cycle.

$$fma(a, b, c) = a + b \times c$$

The proposed optimised cosine similarity calculations can be represented by Eq. 5,

$$S = \sum_{j=0}^{k} (\forall \{ i \in \mathbb{N}, i \leq N \} \; u_j \times V_{ji}) \tag{5}$$

Algorithm 2 Optimised implementation of cosine similarity

Require: $(u, \hat{V})$
Ensure: *user personalisation scores* $= S$
  $S \leftarrow \{0.0, 0.0, 0.0....0.0\}_N$
  $i \leftarrow 1$
  while $i \leq k$ **do**
    $j \leftarrow 1$
    while $j \leq N$ **do** ▷
      $S[j : j+16] \leftarrow fma(S[j : j+16],\ u[i],\ \hat{V}[i][j : j+16])$
      $j \leftarrow j + 16$
    end while
    $i \leftarrow i + 1$
  end while

The proposed algorithm Alg. 2 ensures the maximum utilisation of accumulator and CPU registers with the least data transfer, data copy and cache miss overhead because of FMA instruction, as illustrated in Fig. 3a. It processes 16 elements in a single clock cycle, which is efficient and has lower latency than other implementations. It also ensures maximum spatial and temporal locality for cache hits, reducing the latency. Additionally, the proposed algorithm reduces number of total computations bringing down the processing footprint of each request, enough to localize each request to a single core of processor. Thus, providing significant scaling opportunity by enabling multiple instance of computation process to run in a single node. The number of instances is equal to number of cores available in the node which is empirically calculated.

3.3 Diversified Re-Ranking

After generating the user-personalized score s_i from the scoring layer, the score is merged with the product relevance score r_i to obtain the recommendation quality for each product. Let q_i be the recommendation quality score for the product p_i in the recall set C.

$$q_i = \beta s_i + (1 - \beta) r_i \tag{6}$$

where β is personalization hyperparameter which controls the personalization in recommendations, whose value is derived empirically. The candidate recommendations are then re-ranked ensuring diversity and relevance in the recommendations, and the top M products are shown to the user. The ranking of each product should maximize the recommendation quality score q_i and minimize the repetition of similar products to maximize user engagement and product discovery. In other words, the cumulative quality score and the distance between the top M recommended products should be maximized jointly. Cosine similarity between two product embedding can be used to calculate the distance between two products.

Let W be a subset of products sampled from set of candidate recommendations C such that $|W| = M$; For each $W \subseteq C$, let $\mathcal{P}(W)$ be the probability that the user will browse and add products to cart from the recommended product set W. $\mathcal{P}(W)$ should be maximized to get the most optimal set of top M recommendations. This behaviour can be modelled as a Determinantal Point Process (DPP) as illustrated by Mark *et al.* [59] and Warlop *et al.* [58].

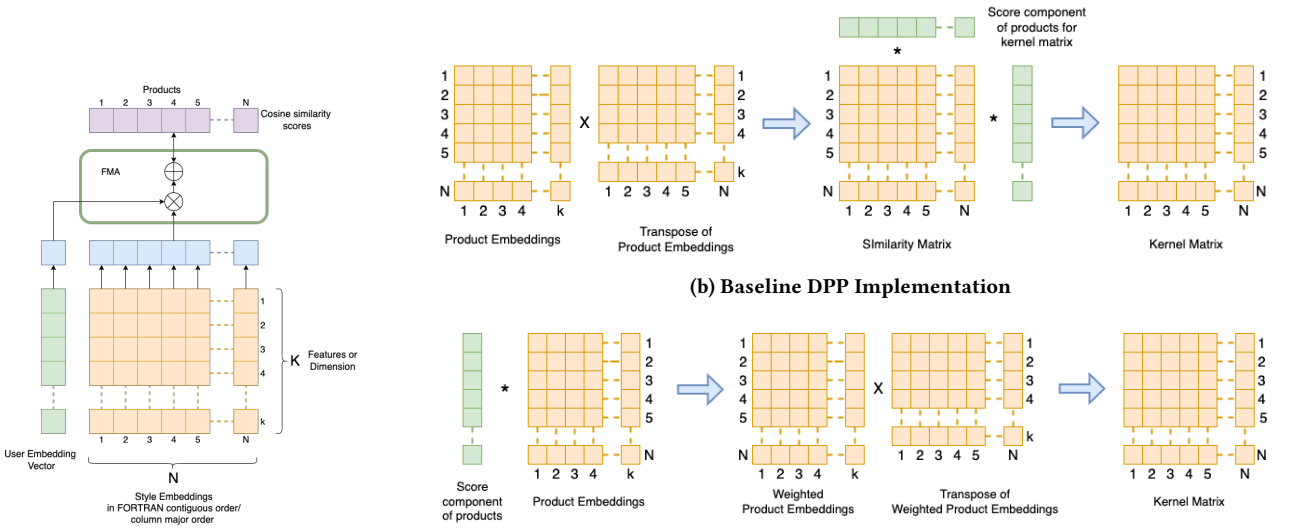
The process of diversification in the recommendations comprise of the following steps:-

(1) Learning the positive semi-definite kernel matrix L that can represent the point process.
(2) Sampling the top M products from the candidate from the DPP kernel matrix.

3.3.1 Learning the DPP kernel. Diversification in the recommended products considers the quality scores q and product-embedding vectors in recall set C.

Let d_{ij} is the cosine similarity distance between the embedding vector of the i^{th} and j^{th} product in the candidate set. The kernel matrix can be parameterized as follows:-

$$\begin{aligned} L_{ij} &= \exp(\alpha^2 q_i q_j) d_{ij} \\ \alpha &= \frac{\theta}{2 \times (1 - \theta)} \end{aligned} \tag{7}$$



(a) Optimised cosine similarity workflow

(b) Baseline DPP Implementation

(c) Proposed DPP Implementation

Figure 3: Optimization of cosine similarity and reformulation of DPP Kernel matrix generation

For diagonal elements since $i = j$, cosine distance between the same products will be 1,i.e. $d_{ij} = 1$. Eq. (7) can be simplified to

$$\begin{aligned} L_{ii} &= \exp(\alpha^2 q_i^2) \\ L_{ij} &= \exp(\alpha^2 q_i q_j) d_{ij} \quad for\ i \neq j \\ \alpha &= \frac{\theta}{2 \times (1-\theta)} \end{aligned} \tag{8}$$

$\theta \in [0, 1]$ is the tunable hyperparameter that controls the recommendations' relevance and diversity. In our experiments, we use $\theta = 0.7$, empirically derived, for best results in recommendations in terms of relevance and diversity. A high value of θ ensures a higher priority to the quality score of recommendations as the kernel matrix will be parameterized heavily on the quality of recommendations. Similarly, a small value of θ will prioritize the diversity of recommended products as the kernel matrix will be parameterized by cosine similarity distance between the embedding vectors.

Constructing the kernel matrix involves getting the cosine similarity distance d_{ij} for every (i, j) pair in candidate recommendations. Since the embedding vectors are normalized in the candidate generation phase, d_{ij} is the dot product of vector pair V_i and V_j. Dot product for every vector pair can be calculated by matrix multiplication of $\hat{V}$ by its transpose $\hat{V}^\intercal$. Let D is an $N \times N$ matrix in which each element D_{ij} contains the cosine similarity distance between i^{th} and j^{th} product in recall set C. In other words, $D_{ij} = dis(p_i, p_j) = d_{ij}$.

$$D = \hat{V}^\intercal \times \hat{V} \tag{9}$$

OpenBLAS [57] and Intel® MKL [56], which use the BLAS interface for OS's kernel-level optimized routines for linear algebra operations, can be utilized for computation of D. Each component of quality scores for product p_i, $\exp(\alpha \times q_i)$, is multiplied by the corresponding element of the similarity matrix D indexed by i to get the parameterized kernel matrix [12, 59]. The implementation is explained in Alg. 3 and illustrated in Fig. 3b. Let x be number of computations involved in matrix multiplication to generate similarity matrix D. Also, number of computations required to parameterise similarity matrix D to generate kernel matrix L is proportional to N^2 as shown in Alg. 3. Therefore, approx number of computations of generating kernel matrix from baseline implementation Alg.3 can be estimated to

$$Number\ of\ Computations = O(x) + O(N^2) \tag{10}$$

Algorithm 3 Baseline implementation of kernel matrix generation

Require: θ ▷ DPP hyperparamter for controlling relevance
Require: $q, \hat{V}$ ▷ Quality score and Product embeddings
Require: N ▷ Number of candidate recommendations
Ensure: L ▷ Parametrised Kernel Matrix

```
α ← θ ÷ (2 × (1 − θ))
L ← V̂ᵀ × V̂   ▷ Using BLAS matrix multiplication subroutine
i ← 1
while i ≤ N do
    while j ≤ N do
        L[i][j] ← exp(αq_i) × exp(αq_j) × L[i][j]
        j ← j + 1
    end while
    i ← i + 1
end while
```

3.3.2 Proposed DPP. We propose an efficient way to compute kernel matrix which involves less number of computation to generate kernel matrix than Alg. 3. Let $(*)$ represents element wise multiplication of a vector and matrix. In case of element wise multiplication of row vector and matrix, each element of vector is multiplied to every element of the corresponding columns. Similarly in case of element wise multiplication of column vector and matrix, each element

of vector is multiplied to every element of the corresponding rows. Let S be the row vector with component of scores, $S_i = \exp(\alpha \times q_i)$. The algorithm Alg. 3 can be represented using $(*)$ as follows:-

$$L = S^\intercal * (\hat{V}^\intercal \times \hat{V}) * S \tag{11}$$

Eq. 11 can be further simplified as,

$$\begin{aligned} & L = S^\intercal * (\hat{V}^\intercal \times \hat{V}) * S \\ \Leftrightarrow\ & L = S^\intercal * \hat{V}^\intercal \times \hat{V} * S \\ \Leftrightarrow\ & L = (S^\intercal * \hat{V}^\intercal) \times (\hat{V} * S) \\ \Leftrightarrow\ & L = (\hat{V} * S)^\intercal \times (\hat{V} * S) \end{aligned} \tag{12}$$

The term $\hat{V} * S$ represents a matrix with each column is the embedding vector e_i of each product p_i is elongated by the factor of the component of quality score $\exp(\alpha \times q_i)$. Using the simplification mentioned in Eq. 12, the implementation of kernel matrix creation can be further optimised by prior computation of $\hat{V} * S$ and the result of the matrix multiplication of the transpose of $\hat{V} * S$ with itself will give the parameterised kernel matrix L as illustrated in Fig. 3c.

Algorithm 4 Proposed Optimised implementation of kernel matrix generation

Require: θ ▷ DPP hyperparamter for controlling relevance
Require: $q, \hat{V}$ ▷ Quality score and Product embeddings
Require: N ▷ Number of candidate recommendations
Ensure: L ▷ Parametrised Kernel Matrix

```
α ← θ ÷ (2 × (1 − θ))
i ← 1
while i ≤ k do
    while j ≤ N do
        V[i][j] ← exp(αq_i) × exp(αq_j) × V[i][j]
        j ← j + 1
    end while
    i ← i + 1
end while
L ← V̂^⊺ × V̂    ▷ Using BLAS matrix multiplication subroutine
```

Since, the number of computations required to parameterise similarity matrix D in Alg. 4 to generate kernel matrix L is proportional to $N \times k$. Therefore, number of computations of generating kernel matrix from proposed implementation Alg. 4 can be estimated to

$$Number\ of\ Computations = O(x) + O(Nk) \tag{13}$$

For production use cases, since k can be treated as constant as it is independent on number of products in the recall set C and for various practical cases $k << N$, the algorithm Alg. 4 works better than Alg. 3 as it iterates over less number of elements to generate the parameterised kernel matrix.

3.3.3 Sampling the top relevant recommendations. Sampling the top M relevant products involves fetching the optimal set of products satisfying Eq. 7. Since, finding the optimal subset of products W_M is NP-Hard, a greedy algorithm [12] for submodular maximisation [42] is used.

4 IMPLEMENTATION DETAILS

We have implemented the proposed workflow of the cross-category recommendations system using Python3.8 with Numba [32] and NumPy [52].

- NumPy is a Python library used to store numerical data in the form of arrays. Internally, NumPy uses low-level functions and kernel libraries for fast mathematical operations. Also, it stores the arrays in a continuous memory buffer like any other low-level programming language like C or Fortran, making it easier to utilize SIMD due to the increase in memory colocation. User and Product embedding vectors are stored in the form of NumPy arrays.
- Numba is a JIT(just-in-time) python compiler that compiles high-level python functions to low-level machine code using the LLVM compiler library. SIMD capabilities can be utilized in by translating into low-level machine code according to embedding vectors memory layout, CPU specification and available registers on the cloud machines without manually specifying the compilation flags.

We avoided assembly-level SIMD to make the implementation portable across systems within x86 architecture. This also makes the implementation easy to develop, debug and maintain as programmer does not have to understand the underlying hardware.

5 EXPERIMENTS AND RESULTS

In this section we discuss the benchmarking of the proposed reformulated components and its impact on the whole recommendation pipeline against the non-optimized baseline method. We benchmarked the baseline workflow and the proposed workflow and established the latency and throughput gains on a private recommendation dataset. We also benchmarked the latency of various compute intensive components and the throughput and latency gain on baseline and proposed workflows.

5.1 Experimental setup

Recommendation dataset consists of 5000 query products and user pairs. These query products are randomly selected from a product catalogue of 1.5 million products. Each embedding are in 81-dimensional vector space $\mathcal{V}$ ($k = 81$) and each product have a recall set of 500 products($N = 500$) and 60 products are sampled as the top relevant products to recommend to user. The experiments are performed on Microsoft Azure cloud based virtual machines which uses Microsoft Azure Cloud Hypervisor based on Microsoft Hyper-V. The underlying hardware on the virtual machine has Intel® Xeon® Platinum 8171M CPU with frequency 2.60GHz with 16 cores and 64GB memory.

5.2 Experiments

We evaluate the impact of our proposed method on scalability of user-facing online services through a series of experiments on the existing and proposed pipelines. We first evaluate the baseline process pipeline (*P-Base*), which is a implementation of the personalization and diversification algorithm as described in [59]. *P-Base* is agnostic of the concurrency in online services and does not contain any explicit thread or multiprocessing control. We introduce

Table 1: Latency(in milliseconds) benchmarking of the baseline implementation(*P-Base*), single threaded baseline implementation(*P-Base-OnlineServ*) and proposed implementation(*P-Proposed*) of recommendation service. KO indicates request timeout.

Traffic	*P-Base*					*P-Base-OnlineServ*					*P-Proposed*				
	Latency (in ms)			Components (in ms)		Latency (in ms)			Components (in ms)		Latency (in ms)			Components (in ms)	
	Mean	P95	P99	Cos Sim	DPP	Mean	P95	P99	Cos Sim	DPP	Mean	P95	P99	Cos Sim	DPP
0.9 k	20.1	43.7	54	3.70	2.78	14.6	24.7	25.2	2.03	1.32	11.1	16.5	19.7	0.052	0.677
3.0 k	98.1	348	555	11.5	50.7	15.1	27.0	29.9	2.14	1.40	11.2	17.2	19.9	0.053	0.678
15.0 k	KO	KO	KO	KO	KO	16.7	32.7	41.5	2.74	1.63	12.0	19.2	25.3	0.053	0.776
42.0 k	KO	KO	KO	KO	KO	196	742	1050	4.18	2.28	18.4	41.5	61.3	0.059	0.909
45.0 k	KO	KO	KO	KO	KO	KO	KO	KO	KO	KO	20.5	43.5	**63.5**	0.061	0.936

Table 2: Ablation study (in milliseconds) of individual proposed components of the recommendation pipeline.

Throughput	Latency (in ms)											
	Baseline [P-Base-OnlineServ]			Baseline + Vec. Similarity [P-VecSim]			Baseline + Vec. DPP [P-VecDPP]			Baseline + Vec. Sim. & DPP [P-Proposed]		
	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99
3.0 k	15.1	27.0	29.9	11.9	17.7	19.7	14.6	25.3	28.2	11.2	17.2	19.9
12.0 k	16.5	31.7	40.2	12.6	20.4	25.5	15.1	29.1	36.6	11.7	17.8	22.2
18.0 k	17.3	34.7	47.4	13.6	22.7	30.5	15.9	31.7	41.5	12.0	19.2	26.5
24.0 k	19.5	41.5	62.0	14.6	28.7	31.5	17.4	35.9	49.1	14.1	27.2	29.0
30.0 k	22.6	51.3	75.2	15.1	34.5	34.9	22.1	45.7	64.3	14.5	32.5	33.1
36.0 k	50.9	150.0	217.0	19.4	44.7	49.3	39.1	62.5	78.2	18.2	37.6	41.9

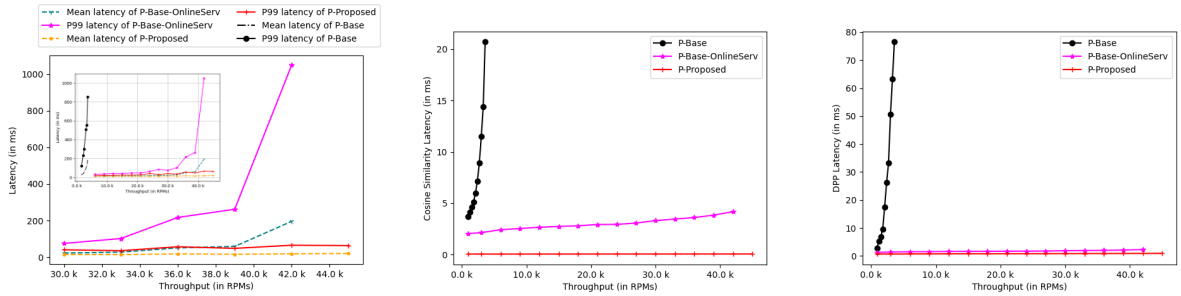
Table 3: Benchmarking the impact of using multithreading in the proposed implementation of the recommendation service.

Throughput	Latency (in ms)											
	1 Threads			4 Threads			8 Threads			16 Threads		
	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99
0.9 k	11.1	16.5	19.7	15.0	24.0	26.7	16.6	25.3	33.4	20.1	43.7	54
1.5 k	11.2	16.9	19.8	16.2	25.8	29.7	17.9	34.4	59.3	26.9	75.2	120
3.0 k	11.2	17.2	19.9	18.2	30.6	45.7	27.9	79.1	131	98.1	348	508

optimal multiprocessing conditions, suited to online services handling concurrent requests, to *P-Base* and establish new pipeline *P-Base-OnlineServ* by controlling number of intra and inter process threads spawned for each request. Finally, we experiment with our proposed pipeline *P-Proposed* which contains the thread control mentioned before and also the vectorized implementation of personalization (Alg. 2) and diversification (Alg. 4). To analyze the scalability of the pipelines under consideration we subject each to varying load of concurrent requests ranging from 900 to 45000 Requests Per Minute (0.9K-45K RPM). We monitor the change of latency under the increasing throughput of requests for each of the pipeline and ascertain the breaking-point as the load under which the pipelines go out-of-service due to congestion and non-serviced timeout of incoming requests. The metrics used for comparative analysis of the experiments are average and peak latency of end-to-end service of a request. Average latency is nothing but the mean of all observed latency in a observation set and peak latency refers to the 99th percentile of all observations, also written as P-99. The permissible limit of P-99 latency is 100ms. The observations are recorded in Table. 1, where latency numbers in red represents unacceptable because P-99 latency exceeds permissible limit and 'KO' represents the experiment could not complete at higher level of concurrency and requests being terminated by system.

5.3 Comparative analysis

Table. 1 shows the observed latency numbers of the *P-Base*, *P-Base-OnlineServ* and *P-Proposed* under various loads on the system simulated with the real-world traffic simulator. As mentioned above, *P-Base* does not contain any explicit thread control built into it and runs with out-of-box settings of high level production APIs in Python. At 0.9K RPM traffic the pipeline runs without any issue, serving requests in 20.1ms on average and the peak latency lies around 54ms. However, with increase in the load, the pipeline starts



(a) Comparison of P-99 latencies **(b) Cosine Similarity** **(c) DPP**

Figure 4: a) Comparison of mean and peak P-99 latency of single threaded baseline implementation (*P-Base-OnlineServ*) and proposed implementation (*P-Proposed*). Complete benchmarking including *P-Base* in inset. Benchmarking of b) Optimization of cosine similarity and c) reformulation of DPP Kernel matrix generation

to choke up and finally gives away under 3.0K RPM load with average latency of 98.1ms and the peak latency climbing upto 555ms. Thus, it can be concluded that *P-Base* is not vertically scalable. From our analysis we ascertained the over usage of multi-threaded parallelization to be the source of bottleneck in the pipeline. Python APIs, in general, are designed with goal of achieving optimal performance of single request on dedicated infra. Thus, lift-and-shift implementation of such APIs in user facing online services that are expected to handle large concurrency is not suitable. Hence, we decided to re-establish the baseline after optimizing the multi-threaded behaviour to make the implementation more suitable to online services for fair comparison. Next we benchmark the thread controlled implementation (*P-Base-OnlineServ*) of Alg. 1 and Alg. 3. The pipeline becomes much more resilient under scale as it can easily serve requests in 15.1ms on average under a load of 3.0K RPM. It is easily able to handle even 15.0K RPM with 16.7ms average and 41.5ms Peak latency. However, breaking-point for *P-Base-OnlineServ* occurs around 42.0K RPM traffic. Next we benchmark our proposed pipeline *P-Proposed* with vectorized implementation of Alg. 2 and Alg. 4 and optimal thread control, derived empirically. *P-Proposed* performs much better under the basic load of 0.9K throughput with average serving latency of 11.1ms and peak latency at P99 of 19.7ms. We also observe that *P-Proposed* scales efficiently under increasing load of 3K and 15K RPM load with less than 10% increase in average serving latency. Under 15.0K RPM, *P-Proposed* has peak latency of only 25.3ms which is significantly improved over *P-Base-OnlineServ*. We validated the proposed pipeline *P-Proposed* to be operational under traffic load of 45.0K RPM. Under this extreme load, the pipeline is still able to serve requests under 20.5ms on average and with peak latency (P99) of 63.5ms. Fig. 4a shows the latency vs. throughput plot of *P-Base*, *P-Base-OnlineServ* and *P-Proposed* which visually depicts the improvement in vertical scalability between the pipelines. Considering horizontal scaling to serve production traffic beyond 45k RPM, we can conclude that *P-Proposed* requires 50-times less hardware instances than *P-Base* and 3-times less hardware instances than *P-Base-OnlineServ* to support a certain amount of traffic. Next, we benchmark each of the individual components impacted by our proposed optimization in *P-Proposed*. Table. 1 also shows the observed average latency for cosine similarity and DPP calculations in *P-Base*, *P-Base-OnlineServ* and *P-Proposed* under increasing traffic. Both cosine similarity and DPP module latency grows aggressively in *P-Base* and reaches 11.5ms and 50.7ms on average respectively. This validates the earlier observations that 3.0K RPM is the breaking-point traffic for *P-Base*. At 0.9K RPM traffic, in *P-Base-OnlineServ*, latency of cosine similarity computation is 2.03ms while that of DPP is 1.32ms on average which is acceptable but on the higher side for a real-time recommendation service. However, with increase in the traffic, latency of both components increase rapidly. This causes 'thrashing' in the system *i.e.* more time is wasted in context switching than computing. This causes congestion and non-served request queue build-up in the system, leading to rapid increase in latency. At 42.0K RPM traffic, average latency of cosine similarity goes upto 4.18ms and DPP latency reaches upto 2.28ms. Thus, it can be concluded that the baseline implementation of cosine similarity and DPP are not entirely vertically scalable. However, in contrast, the cosine similarity and DPP component serving latency in the proposed vectorized implementation *P-Proposed* is significantly lower compared to the baseline implementation *P-Base-OnlineServ*.

In case of *P-Proposed*, vectorized cosine similarity incurs a average serving latency of 52μs under nominal load of 0.9K RPM, which only increases upto 61μs under 45.0K RPM traffic. This indicates the proposed vectorized implementation of Alg. 2 is highly scalable and can perform with almost constant latency under aggressively increasing requests traffic. Fig. 4b shows the comparative plots of mean latency of cosine similarity pipeline. Similarly, the reformulated vectorized DPP operates with average execution latency of 677μs under 0.9K RPM traffic and is able to constrain the average latency to just under 1ms (936μs) when subjected to a traffic of 45.0K RPM. Fig. 4c shows the comparative plots of mean latency of DPP pipeline. Although the plots of DPP component in *P-Base-OnlineServ* and *P-Proposed* seem to be close to each other but that is due to the unconstrained growth of the DPP latency in *P-Base*.

Thus, the comparative analysis of critical components of our proposed method comprehensively support our claims of significantly improving the scalability of the diversified recommendation pipeline. We take a further closer look at the impact of these critical components later in Sec. 6 by evaluating their influence in isolation.

6 ABLATION STUDY

In this section, we analyse the results of our ablation study of the individual components of the proposed method to assess their individual and collective influence on improving the scalability of the pipeline. We also evaluate the impact of using traditional optimization methods like threaded parallelization on our proposed method.

6.1 Impact of individual components

We analyse the influence of individual proposed components in improving the scalability of the recommendation pipeline through a set of ablation experiments. Table. 2 shows the result of the ablation experiments. The behaviour of *P-Base-OnlineServ* is already discussed above and noted to be not scalable to a high degree. However, upon introducing the proposed vectorized Cosine similarity (Alg. 2) to this pipeline, we observed that the resultant pipeline *P-VecSim* becomes significantly more scalable. *P-VecSim* can serve with an average and P-99 latency of 19.4ms and 49.3ms respectively under 36.0K RPM traffic. This shows significant improvement over *P-Base-OnlineServ* which have average and P-99 latency of 50.9ms and 217.0ms respectively under similar conditions. Next, we evaluate the influence of the proposed vectorized DPP (Alg. 4) module. Upon introducing same to *P-Base-OnlineServ*, the resultant pipeline *P-VecDPP* also becomes readily more scalable. The average and peak latency of *P-VecDPP*, 39.1ms and 78.2ms respectively, when subjected to a traffic of 36.0K RPM, remains decently under control to keep the pipeline operational. Introducing both vectorized similarity and vectorized DPP to *P-Base-OnlineServ* creates our proposed pipeline *P-Proposed*, which by virtue of including both the scalable components demonstrates highest scalability. Subjected to a traffic of 36.0K RPM, *P-Proposed* incurs average and peak latency of 18.2ms and 41.9ms respectively, growing only 60% over the average latency of 11.2ms under 3.0K RPM. The peak latency growth under maximum traffic is also constrained under only to 3 times the original peak latency for a 12x growth in traffic. These observations prove our claims of proposing individual scalable components, whose sum is even greater than the parts in terms of improving scalability of the recommendation pipeline.

6.2 Impact of threaded parallelization

We also evaluated impact of using conventional optimization of multithreaded parallelism on scalability of proposed recommendation system pipeline. Table. 3 shows that the single threaded pipeline scales most efficiently under increasing traffic with less than 10% increase in the average serving latency over the range of loads. However, even with 4 threads per request, the average and peak latency doubles at 3.0K RPM than the 1-thread pipeline. 8 and 16 threaded implementations cross real-time SLA limits under only 3K RPM load with nearly 7x and 25x growth in peak latency over 1-thread pipeline. Thus, it can be concluded that naive usage of multithreading without vectorization does not work well in practice for online recommendation systems.

7 OPEN PROBLEMS AND FUTURE WORK

We have conducted in-depth experiments to evaluate the performance of our proposed modification to the similarity and DPP algorithms to leverage benefits of SIMD multiprocessing in server and virtual machine environments. Evaluating the performance of the same algorithms with SIMD multiprocessing in a containerized environment e.g. Kubernetes (k8s) is one of the goals of our future work. We have also been exposed solely to x86 and x86_64 Instruction Set Architecture (ISA) as these are the most commonly found instruction set in server and cloud VM environments. Evaluating the impact of SIMD multiprocessing on different ISAs such as ARM also remains an open problem to be addressed in future publications from this body of work. On the other hand, we have focused on constituting portable SIMD accelerated method and thus had to look beyond Assembly level SIMD which becomes bound to specific CPUs with specific register counts. However, this problem can be solved alternatively by bypassing the dependency of register lengths in Assembly level SIMD and we believe considerable amount of research scope is present in that area.

8 CONCLUSION

In this paper, we explored the opportunities of moving past conventional optimization strategies like threaded parallelization for CPU only online recommendation systems and adopting the SIMD optimizations by redesigning critical components, with motivations of vectorization, of a 'complement' recommendation system. We experimentally show that vectorized implementation of the well known blocks like personalization and diversification can be made significantly more scalable by utilizing SIMD compute powers of modern day CPUs. We use portable SIMD constructs in Python to make the implementation easily portable across different CPUs and architectures. The approach advocated-for in this work can be extended to many available recommendation systems that are deployed in CPU only servers and can be pivotal shift towards making efficient large scale e-commerce services. In addition to endorsing the use in recommender systems, we propose and encourage further exploration of proposed algorithm in vector similarity operations, particularly within applications such as vector databases, which serve as efficient storage and retrieval systems for vector embeddings (e.g., Milvus [40], Pinecone [44], ChromaDB [13]). As these applications are heavily reliant on similarity algorithms of the kind proposed in this work, we believe further research into same will help make these applications more optimized and real-time. Furthermore, considering the influence of these applications in the increasingly popular Generative-AI paradigm, optimizing them holds the promise of rendering Generative-AI applications more responsive and suitable for large-scale deployments in real-world scenarios.

REFERENCES

[1] Gediminas Adomavicius and YoungOk Kwon. 2007. New Recommendation Techniques for Multicriteria Rating Systems. *IEEE Intelligent Systems* 22, 3 (2007), 48–55. https://doi.org/10.1109/mis.2007.58

[2] Pankaj Agarwal, Sreekanth Vempati, and Sumit Borar. 2018. Personalizing similar product recommendations in fashion e-commerce. *AI for fashion, The third international workshop on fashion and KDD* (2018).

[3] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. 2009. Diversifying Search Results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining* (Barcelona, Spain) *(WSDM '09)*. Association for Computing Machinery, New York, NY, USA, 5–14. https://doi.org/10.1145/1498759.1498766

[4] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining Association Rules between Sets of Items in Large Databases. *SIGMOD Rec.* 22, 2 (jun 1993),

207–216. https://doi.org/10.1145/170036.170072
[5] Rémi Bardenet and Michalis Titsias RC AUEB. 2015. Inference for determinantal point processes without spectral knowledge. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2015/file/2f25f6e326adb93c5787175dda209ab6-Paper.pdf
[6] Alexei Borodin. 2015. 231Determinantal point processes. In *The Oxford Handbook of Random Matrix Theory*. Oxford University Press. https://doi.org/10.1093/oxfordhb/9780198744191.013.11 arXiv:https://academic.oup.com/book/0/chapter/365880931/chapter-ag-pdf/45289415/book_43656_section_365880931.ag.pdf
[7] Jaime Carbonell and Jade Goldstein. 1998. The Use of MMR, Diversity-Based Reranking for Reordering Documents and Producing Summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Melbourne, Australia) *(SIGIR '98)*. Association for Computing Machinery, New York, NY, USA, 335–336. https://doi.org/10.1145/290941.291025
[8] Josiah Carlson. 2013. *Redis in action*. Simon and Schuster.
[9] Olivier Chapelle, Shihao Ji, Ciya Liao, Emre Velipasaoglu, Larry Lai, and Su-Lin Wu. 2011. Intent-based diversification of web search results: metrics and algorithms. *Information Retrieval* 14 (2011), 572–592.
[10] Olivier Chapelle, Shihao Ji, Ciya Liao, Emre Velipasaoglu, Larry Lai, and Su-Lin Wu. 2011. Intent-based diversification of web search results: metrics and algorithms. *Information Retrieval* 14, 6 (2011), 572–592. https://doi.org/10.1007/s10791-011-9167-7
[11] Laming Chen, Guoxin Zhang, and Hanning Zhou. 2017. Improving the diversity of top-N recommendation via determinantal point process. (2017).
[12] Laming Chen, Guoxin Zhang, and Hanning Zhou. 2018. Fast Greedy MAP Inference for Determinantal Point Process to Improve Recommendation Diversity. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 5627–5638.
[13] ChromaDB. 2023. *Chroma : the AI-native open-source embedding database*. https://www.trychroma.com/
[14] Charles LA Clarke, Maheedhar Kolla, Gordon V Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian MacKinnon. 2008. SIGIR - Novelty and diversity in information retrieval evaluation. *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '08* NA, NA (2008), 659–666. https://doi.org/10.1145/1390334.1390446
[15] Angeline G. Close and Monika Kukar-Kinney. 2010. Beyond buying: Motivations behind consumers' online shopping cart use. *Journal of Business Research* 63, 9 (2010), 986–992. https://doi.org/10.1016/j.jbusres.2009.01.022 Advances in Internet Consumer Behavior & Marketing Strategy.
[16] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems* (Boston, Massachusetts, USA) *(RecSys '16)*. Association for Computing Machinery, New York, NY, USA, 191–198. https://doi.org/10.1145/2959100.2959190
[17] Van Dang and W. Bruce Croft. 2012. Diversity by Proportionality: An Election-Based Approach to Search Result Diversification. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Portland, Oregon, USA) *(SIGIR '12)*. Association for Computing Machinery, New York, NY, USA, 65–74. https://doi.org/10.1145/2348283.2348296
[18] Marina Drosou and Evaggelia Pitoura. 2010. Search result diversification. *ACM SIGMOD Record* 39, 1 (2010), 41–47. https://doi.org/10.1145/1860702.1860709
[19] Mike Gartrell, Ulrich Paquet, and Noam Koenigstein. 2016. RecSys - Bayesian Low-Rank Determinantal Point Processes. *Proceedings of the 10th ACM Conference on Recommender Systems* NA, NA (2016), 349–356. https://doi.org/10.1145/2959100.2959178
[20] Jennifer Gillenwater. 2014. Approximate inference for determinantal point processes. *NA* NA, NA (2014), NA–NA.
[21] Jennifer A Gillenwater, Alex Kulesza, Emily Fox, and Ben Taskar. 2014. Expectation-Maximization for Learning Determinantal Point Processes. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2014/file/4462bf0ddbe0d0da40e1e828ebebeb11-Paper.pdf
[22] Sreenivas Gollapudi and Aneesh Sharma. 2009. WWW - An axiomatic approach for result diversification. *Proceedings of the 18th international conference on World wide web - WWW '09* NA, NA (2009), 381–390. https://doi.org/10.1145/1526709.1526761
[23] Yoshinori Hijikata, Takuya Shimizu, and Shogo Nishida. 2009. Discovery-Oriented Collaborative Filtering for Improving User Satisfaction. In *Proceedings of the 14th International Conference on Intelligent User Interfaces* (Sanibel Island, Florida, USA) *(IUI '09)*. Association for Computing Machinery, New York, NY, USA, 67–76. https://doi.org/10.1145/1502650.1502663
[24] Henning Hohnhold, Deirdre O'Brien, and Diane Tang. 2015. KDD - Focusing on the Long-term: It's Good for Users and Business. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* NA, NA (2015), 1849–1858. https://doi.org/10.1145/2783258.2788583
[25] J Ben Hough, Manjunath Krishnapur, Yuval Peres, and Bálint Virág. 2006. Determinantal processes and independence. *Probability surveys* 3 (2006), 206–229.
[26] Chun-Wa Ko, Jon Lee, and Maurice Queyranne. 1995. An exact algorithm for maximum entropy sampling. *Operations Research* 43, 4 (1995), 684–691. https://doi.org/10.1287/opre.43.4.684
[27] Joseph A Konstan, Bradley N Miller, David Maltz, Jonathan L Herlocker, Lee R Gordon, and John Riedl. 1997. Grouplens: Applying collaborative filtering to usenet news. *Commun. ACM* 40, 3 (1997), 77–87.
[28] Joseph A Konstan, John Riedl, Al Borchers, and Jonathan L Herlocker. 1998. Recommender systems: A grouplens perspective. In *Recommender Systems: Papers from the 1998 Workshop (AAAI Technical Report WS-98-08)*. AAAI Press Menlo Park, 60–64.
[29] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37. https://doi.org/10.1109/MC.2009.263
[30] Alex Kulesza and Ben Taskar. 2011. K-DPPs: Fixed-Size Determinantal Point Processes. In *Proceedings of the 28th International Conference on International Conference on Machine Learning* (Bellevue, Washington, USA) *(ICML'11)*. Omnipress, Madison, WI, USA, 1193–1200.
[31] Alex Kulesza and Ben Taskar. 2011. Learning Determinantal Point Processes. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence* (Barcelona, Spain) *(UAI'11)*. AUAI Press, Arlington, Virginia, USA, 419–427.
[32] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162
[33] Neal Lathia, Stephen Hailes, Licia Capra, and Xavier Amatriain. 2010. SIGIR - Temporal diversity in recommender systems. *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '10* NA, NA (2010), 210–217. https://doi.org/10.1145/1835449.1835486
[34] G. Linden, B. Smith, and J. York. 2003. Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing* 7, 1 (2003), 76–80. https://doi.org/10.1109/MIC.2003.1167344
[35] Odile Macchi. 1975. The coincidence approach to stochastic point processes. *Advances in Applied Probability* 7, 1 (1975), 83–122.
[36] Zelda Mariet and Suvrit Sra. 2015. Fixed-point algorithms for learning determinantal point processes. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 2389–2397. https://proceedings.mlr.press/v37/mariet15.html
[37] Zelda E. Mariet and Suvrit Sra. 2016. Kronecker Determinantal Point Processes. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2016/file/bad5f33780c42f2588878a9d07405083-Paper.pdf
[38] Julian McAuley, Rahul Pandey, and Jure Leskovec. 2015. Inferring Networks of Substitutable and Complementary Products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2783258.2783381
[39] Sean M McNee, John Riedl, and Joseph A Konstan. 2006. CHI Extended Abstracts - Being accurate is not enough: how accuracy metrics have hurt recommender systems. *CHI '06 Extended Abstracts on Human Factors in Computing Systems* NA, NA (2006), 1097–1101. https://doi.org/10.1145/1125451.1125659
[40] Milvus. 2023. *Milvus: Vector database*. https://milvus.io/
[41] Houssam Nassif, Kemal Oral Cansizlar, Mitchell Goodman, and SVN Vishwanathan. 2016. Diversifying Music Recommendations. *arXiv: Multimedia* NA, NA (2016), NA–NA.
[42] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functions–I. *Mathematical Programming* 14, 1 (1978), 265–294. https://doi.org/10.1007/bf01588971
[43] Yonathan Perez, Michael Schueppert, Matthew Lawlor, and Shaunak Kishore. 2015. Category-Driven Approach for Local Related Business Recommendations. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management* (Melbourne, Australia) *(CIKM '15)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2806416.2806495
[44] Inc. Pinecone Systems. 2023. *Pinecone: Vector Database for Vector Search*. https://www.pinecone.io/
[45] Davood Rafiei, Krishna Bharat, and Anand Shukla. 2010. WWW - Diversifying web search results. *Proceedings of the 19th international conference on World wide web - WWW '10* NA, NA (2010), 781–790. https://doi.org/10.1145/1772690.1772770
[46] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. 1994. GroupLens: An Open Architecture for Collaborative Filtering of

Netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work* (Chapel Hill, North Carolina, USA) *(CSCW '94)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/192844.192905

[47] Rodrygo LT Santos, Craig Macdonald, and Iadh Ounis. 2010. WWW - Exploiting query reformulations for web search result diversification. *Proceedings of the 19th international conference on World wide web - WWW '10* NA, NA (2010), 881–890. https://doi.org/10.1145/1772690.1772780

[48] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-Based Collaborative Filtering Recommendation Algorithms. In *Proceedings of the 10th International Conference on World Wide Web* (Hong Kong, Hong Kong) *(WWW '01)*. Association for Computing Machinery, New York, NY, USA, 285–295. https://doi.org/10.1145/371920.372071

[49] Cyrus Shahabi, Farnoush Banaei-Kashani, Yi-Shin Chen, and Dennis McLeod. 2001. Yoda: An Accurate and Scalable Web-Based Recommendation System. In *Cooperative Information Systems*, Carlo Batini, Fausto Giunchiglia, Paolo Giorgini, and Massimo Mecella (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 418–432.

[50] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. 2016. Aerospike: Architecture of a Real-Time Operational DBMS. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1389–1400. https://doi.org/10.14778/3007263.3007276

[51] Choon Hui Teo, Houssam Nassif, Daniel Hill, Sriram Srinivasan, Mitchell Goodman, Vijai Mohan, and SVN Vishwanathan. 2016. RecSys - Adaptive, Personalized Diversity for Visual Discovery. *Proceedings of the 10th ACM Conference on Recommender Systems* NA, NA (2016), 35–38. https://doi.org/10.1145/2959100.2959171

[52] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30. https://doi.org/10.1109/MCSE.2011.37

[53] Guido Van Rossum and Fred L Drake. 2009. *Python 3 reference manual*. CreateSpace.

[54] Saúl Vargas, Linas Baltrunas, Alexandros Karatzoglou, and Pablo Castells. 2014. RecSys - Coverage, redundancy and size-awareness in genre diversity for recommender systems. *Proceedings of the 8th ACM Conference on Recommender systems - RecSys '14* NA, NA (2014), 209–216. https://doi.org/10.1145/2645710.2645743

[55] Erik Vee, Utkarsh Srivastava, Jayavel Shanmugasundaram, Prashant Bhat, and Sihem Amer Yahia. 2008. ICDE - Efficient Computation of Diverse Query Results. *2008 IEEE 24th International Conference on Data Engineering* NA, NA (2008), 228–236. https://doi.org/10.1109/icde.2008.4497431

[56] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. *Intel Math Kernel Library*. Springer International Publishing, Cham, 167–188. https://doi.org/10.1007/978-3-319-06486-4_7

[57] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on X86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/2503210.2503219

[58] Romain Warlop, Jérémie Mary, and Mike Gartrell. 2019. Tensorized Determinantal Point Processes for Recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1605–1615. https://doi.org/10.1145/3292500.3330952

[59] Mark Wilhelm, Ajith Ramanathan, Alexander Bonomo, Sagar Jain, Ed H. Chi, and Jennifer Gillenwater. 2018. Practical Diversified Recommendations on YouTube with Determinantal Point Processes. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management* (Torino, Italy) *(CIKM '18)*. Association for Computing Machinery, New York, NY, USA, 2165–2173. https://doi.org/10.1145/3269206.3272018

[60] An Yan, Chaosheng Dong, Yan Gao, Jinmiao Fu, Tong Zhao, Yi Sun, and Julian Mcauley. 2022. Personalized Complementary Product Recommendation. In *Companion Proceedings of the Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 146–151. https://doi.org/10.1145/3487553.3524222

[61] Cong Yu, Laks Lakshmanan, and Sihem Amer-Yahia. 2009. EDBT - It takes variety to make a world: diversification in recommender systems. *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09* NA, NA (2009), 368–378. https://doi.org/10.1145/1516360.1516404

[62] Mi Zhang and Neil Hurley. 2008. RecSys - Avoiding monotony: improving the diversity of recommendation lists. *Proceedings of the 2008 ACM conference on Recommender systems - RecSys '08* NA, NA (2008), 123–130. https://doi.org/10.1145/1454008.1454030

[63] Jiaqian Zheng, Xiaoyuan Wu, Junyu Niu, and Alvaro Bolivar. 2009. Substitutes or Complements: Another Step Forward in Recommendations. In *Proceedings of the 10th ACM Conference on Electronic Commerce* (Stanford, California, USA) *(EC '09)*. Association for Computing Machinery, New York, NY, USA, 139–146. https://doi.org/10.1145/1566374.1566394

[64] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. 2005. WWW - Improving recommendation lists through topic diversification. *Proceedings of the 14th international conference on World Wide Web - WWW '05* NA, NA (2005), 22–32. https://doi.org/10.1145/1060745.1060754

An Adaptive Logging System (ALS): Enhancing Software Logging with Reinforcement Learning Techniques

Amirmahdi Khosravi Tabrizi
Brock University
St. Catharines, ON, Canada
akhosravitabriz@brocku.ca

Naser Ezzati-Jivan
Brock University
St. Catharines, ON, Canada
nezzati@brocku.ca

Francois Tetreault
Ciena
Ottawa, ON, Canada
ftetreau@ciena.com

ABSTRACT

The efficient management of software logs is crucial in software performance evaluation, enabling detailed examination of runtime information for postmortem analysis. Recognizing the importance of logs and the challenges developers face in making informed log-placement decisions, there is a clear need for a robust log-placement framework that supports developers. Existing frameworks, however, are limited by their inability to adapt to customized logging objectives, a concern highlighted by our industrial partner, Ciena, who required a system for their specific logging goals in resource-limited environments like routers. Moreover, these frameworks often show poor cross-project consistency. This study introduces a novel performance logging objective designed to uncover potential performance-bugs, categorized into three classes—Loops, Synchronization, and API Misuses—and defines 12 source code features for their detection. We present an Adaptive Logging System (ALS), based on reinforcement learning, which adjusts to specified logging objectives, particularly for identifying performance-bugs. This framework, not restricted to specific projects, demonstrates stable cross-project performance. We trained and evaluated ALS on Python source code from 17 diverse open-source projects within the Apache and Django ecosystems. Our findings suggest that ALS has the potential to significantly enhance current logging practices by providing a more targeted, efficient, and context-aware logging approach, particularly beneficial for our industry partner who requires a flexible system that adapts to varied performance objectives and logging needs in their unique operational environments.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software post-development issues**; • **General and reference → Performance**; **Empirical studies**; • **Computing methodologies → Reinforcement learning**; **Machine learning**.

ACM Reference Format:
Amirmahdi Khosravi Tabrizi, Naser Ezzati-Jivan, and Francois Tetreault. 2024. An Adaptive Logging System (ALS): Enhancing Software Logging with Reinforcement Learning Techniques . In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645033

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05…$15.00
https://doi.org/10.1145/3629526.3645033

1 INTRODUCTION

Logging, as an ubiquitous programming technique, involves the insertion of code that records key runtime information. Careful consideration of log placement is imperative, as the data captured by logs constitute a crucial source of information for postmortem analysis. In the event of system failures, logs often remain the only available source of data. For successful log analysis, it is crucial to have a strong underlying logging, as it directly influences the quality of the collected logs.

Given the importance of logging, it is crucial to strike a balance [22]. Logging too little could result in missing essential runtime information needed for postmortem analysis, making it challenging to diagnose failures in the field. On the other hand, logging too much brings its own set of problems. This includes an increased code volume that requires time for writing and maintenance. Moreover, it consumes additional system resources, impacting overall system performance, especially when dealing with high log volumes. Importantly, excessive logging may generate numerous trivial and unnecessary logs, masking crucial information and complicating issue identification.

Despite its importance, not all developers possess the necessary expertise to make informed logging decisions [4]. Previous research has presented various frameworks to aid developers in making logging decisions. Some frameworks help developers determine which parts of the system to log [4, 22], while others help to select appropriate log-levels for log statements [10, 11], and to effectively structure log messages [5, 8]. Mastropoalo et al. [12] introduced a comprehensive framework that integrates these three logging aspects using transformer models. However, there are some limitations towards existing frameworks, 1) they are limited to logging objective of the project that trained on and are not capable of adapting themselves to a desired logging objective. 2) None of them have considered performance-bugs as a logging objective. 3) Poor cross-project performance.

In collaboration with Ciena, we recognized the need for a logging system that not only adapts to various performance objectives but also addresses the specific logging needs in resource-constrained environments. In response, we introduce the Adaptive Logging System (ALS) that leverages reinforcement learning, along with a new logging objective—Performance Bug Logging Objective—designed to capture and reveal performance-bugs through logs. This dual approach of ALS, combining adaptability with a targeted logging objective, makes it particularly suitable for varied operational needs,

enabling it to efficiently handle different performance objectives and logging requirements. Such a system is invaluable in environments where resource constraints and the need for efficient logging and performance analysis are critical.

Our contributions in this paper are as following:

- Proposing an Adaptive Logging system being able to adapt itself to self-defined logging objectives using reinforcement learning.
- Introducing *performance-bugs* logging objective to capture and reveal performance-bugs through logs.
- Creating a dataset that includes static source code features related to performance-bugs at the function-level for 17 different Apache and Django projects.
- Illustrating the cross-project efficiency of the proposed adaptive logging system by evaluating it on unseen environments.

This paper is structured as follows: Section 2 provides a background on software logging and reinforcement learning. Section 2.3 explores the motivations for our study. Sections 3 and 4 detail our empirical study on performance-bugs and the ALS framework, respectively. Section 5 presents our evaluation methodology and results. Finally, Section 5.4 discusses the study's limitations and future directions, and Section 6 summarizes our findings.

2 BACKGROUND AND LITERATURE REVIEW

2.1 Software Logging

The placement of log statements is guided by one or multiple logging objectives chosen by developers. These objectives include *Performance* [21], which focuses on minimizing performance overhead of logs; *Unexpected Situations* [4], which aim to identify errors; and *Execution Points* [4], which track system runtime states and execution path for root-cause analysis.

Developers face three crucial decisions when implementing a logging strategy. Firstly, they must determine the most appropriate location in the source code for logging (*Where to log?*) [4, 19, 21, 22]. Secondly, they must select the information to be logged and the log statements to be used (*What to log?*) [8]. Lastly, they must choose the appropriate log-level from a range of options including trace, debug, warn, info, error, and fatal (*Which log-level to choose?*) [13].

Prior studies have introduced log-placement frameworks to aid developers. Yuan et al. introduced ErrLog [19], a static programming method that adds logging statements using generic error patterns. Zhao et al. proposed Log20 [21], a DP-based framework that recommends near optimal log placements with low overhead. J. Zhu et al. developed LogAdvisor [22], a machine learning framework that automatically learns common logging rules and provides guidance.

2.2 Reinforcement Learning

In this part, we present the fundamental ideas of Reinforcement Learning (RL), a subfield of machine learning (ML), which we utilize in our research.

Definition. *Reinforcement Learning* is a goal-directed learning method that allows agents to solve sequential decision problems through trial-and-error and interaction [6, 17]. RL aims to determine the optimal mapping of situations to actions by maximizing a reward signal that represents the problem's *goal*.

A RL model consists of two components: Agent and environment. The Environment provides information on the system state, and the Agent selects actions based on that information and interacts with the Environment. The Environment updates the state and returns a reward after each action, creating a cycle of (state → action → reward), depicted in Figure 1, until a predetermined terminal state or timestep is reached.

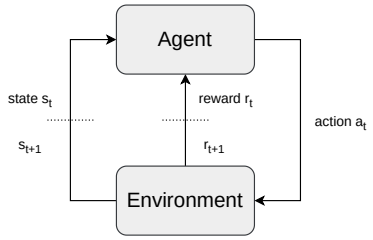


Figure 1: The reinforcement learning control loop

Difference from Other Methods. RL differs from other ML techniques such as supervised and unsupervised learning. Supervised learning requires external knowledge in the form of a training set, which specifies the correct behavior. However, in RL, this knowledge is not provided and must be acquired through the pursuit of objectives. In contrast, supervised learning only aims to learn the correct behavior provided, limiting it to that specific behavior. The objectives of RL and unsupervised learning are distinct from each other. Unsupervised learning aims to uncover hidden structures, while RL aims to maximize a reward signal to reach a specific goal.

The defining feature of RL models is their goal-seeking ability, which aligns them with the learning processes in humans and animals. This property provides RL with a high degree of adaptiveness. Its adaptive nature allows it to continually update and improve upon its knowledge, even when faced with unseen projects. This is a capability that is unique to RL, as other ML methods lack this adaptiveness.

2.3 Research Gaps

While the preceding sections have laid out the fundamental concepts in software logging and reinforcement learning, it is necessary to identify the existing limitations within these domains. This section briefly explains the motivation and research gaps in the current state of the art of log-placement frameworks, particularly highlighting the areas that remain unaddressed or inadequately tackled by existing solutions.

Logging Objectives. Traditional log-placement frameworks are typically limited in their scope, being designed to address specific logging objectives based on the datasets or environments they were originally trained in. This rigor presents significant challenges in dynamic and varied operational contexts, such as resource-limited environments, like network systems or routers. The logging objectives can vary greatly depending on the specific performance requirements and resource constraints of each system.

For instance, in a resource-constrained environment, the primary logging goal may be to minimize performance overhead while maximizing the utility of each log entry for effective bug detection and system monitoring. This differs from resource-abundant environments, where the emphasis may be on capturing comprehensive data for in-depth analysis. The ability to dynamically adjust log-placement strategies based on these different goals is crucial for maintaining system efficiency and reliability, yet it is currently lacking in existing frameworks [4, 22].

Moreover, the detection of performance-bugs, a critical aspect in ensuring the smooth operation of resource-limited systems, is often underrepresented in existing logging frameworks. While these frameworks primarily focus on capturing system errors and exceptions to aid in debugging and ensuring system reliability, they typically do not prioritize the identification and logging of performance anomalies. Effective logging of such anomalies is vital for preemptive maintenance and avoiding system downtimes, making the need for adaptable and performance-oriented logging frameworks even more essential.

Therefore, there is a strong need for log-placement frameworks that are not only adaptable to a broad spectrum of logging objectives but also sensitive to the unique demands of resource-limited settings. This adaptability is essential for tailoring logging strategies to effectively balance performance, resource utilization, and diagnostic needs, thereby enhancing the overall resilience and efficiency of the system.

Implementation Method. The complexity of log-placement, influenced by multiple factors such as system architecture, operational context, and specific performance requirements, poses significant challenges for a comprehensive and adaptable implementation. Traditional methods like *LogAdvisor* [22], while effective in certain settings, are limited by their lack of flexibility and cross-project accuracy. This constraint limits their effectiveness across diverse projects, particularly when transitioning from one domain or technology stack to another.

To address these limitations, we propose the Adaptive Logging System, which leverages Reinforcement Learning—a form of machine learning that excels in making decisions under uncertainty and adapting to new environments. By employing RL, ALS can dynamically learn from the specific characteristics and requirements of each project, continually refining its log-placement strategy to maximize efficiency and relevance. This capability enables ALS to provide tailored logging solutions that maintain high levels of accuracy and utility across various projects and environments, aligning with the diverse and evolving needs of modern software development.

3 LOGGING FOR PERFORMANCE-BUGS

To introduce performance-bugs as a logging objective, we must first gain a clear understanding of what they are. This understanding will enable us to subsequently establish appropriate metrics and features for their description. To achieve this, we conducted an empirical study of existing studies on performance issues. We have categorized performance issues into three distinct categories (Loops, Synchronization Issues and API Misuses) and devised 12 static source code features to characterize each of these categories

Table 1: performance-bugs and their defined features.

	Categories	Features
performance-bugs	Loops	- number-of-loops - nested-loop-level - loop-input-dependent-level
	Synchronization issues	- number-of-defined-threads - number-of-started-threads - number-of-join-threads - number-of-defined-locks - number-of-acquired-locks-threads - number-of-released-locks
	API misuses	- number-of-usage-of-extra - number-of-usage-of-order-by - number-of-usage-of-select-related

(refer to Table 1). In the following parts of this section, we will provide detailed explanations for each category along with their corresponding features.

3.1 Loops

When addressing performance-bugs, loops create a challenging context for their occurrence. This is because loops have the potential to worsen the impact of performance-bugs, often accumulating issues across multiple iterations of the loop [3, 15, 16]. What's even more significant is that a large portion of performance-bugs occur within loops that depend on input data, accounting for nearly three-quarters of such cases [9]. This highlights the importance of paying close attention to loops in our efforts to mitigate performance-bugs.

We have established three distinctive features to encapsulate the vital aspects concerning loops, aimed at characterizing pertinent factors for integration into our RL model. These features are as follows:

(1) **Number of Loops** (number-of-loops): This feature quantifies the number of distinct loop constructs defined within a given function. This feature only counts the outer loops in case of having nested loops. For example, if a code snippet contains one nested loop (with a nesting level of 2) and one non-nested loop, the total number of loops in the snippet is 4, but the Number of Loops metric for the snippet is 2.
(2) **Nested Loop Level** (nested-loop-level): As a descriptor of loop complexity, this feature indicates the total number of loops within a function, including any nested loops. Using the previous example of a nested loop with a nesting level of 2 and a non-nested loop, the *Nested Loop Level* value here would be 4.
(3) **Loop Input Dependency Level** (loop-input-dependent-level): This feature identifies dependent variables, which are variables that rely on a function to determine their values, within the loop declaration. It provides information about the count of these dependent variables if any are found.

3.2 Synchronization Issues

Synchronization challenges can arise in concurrent programming, leading to performance issues [1, 7]. These problems often stem from the improper use of synchronization techniques, especially when selecting the wrong types of locks. Among the notable synchronization challenges, two stand out: deadlocks and race conditions.

Race conditions happen when two different threads try to change the same information at the same time without following a specific order. On the other hand, Deadlocks occur when several threads get stuck in their work because they are all waiting for something that another thread in the same group is using. We have identified six unique attributes related to synchronization challenges. These features are customized for Python's Thread library:

(1) **Number of Thread Objects** (number-of-defined-threads): This attribute indicates the count of threads that are defined within a given function.
(2) **Number of start() Function Calls** (number-of-started-threads): Following the creation of a Thread object, its activation is initiated by invoking the start() function associated with the created object. This feature quantifies the instances of start() function calls, representing thread activation within a specific function.
(3) **Number of join() Function Calls** (number-of-join-threads): The join() function, when invoked, causes the calling thread (typically the main thread) to wait until the thread on which join() is called terminates. This feature quantifies the instances of join() function calls.
(4) **Number of Lock Objects** (number-of-defined-locks): This attribute signifies the count of locks defined within a given function.
(5) **Number of acquire() Function Calls** (number-of-acquired-locks-threads): The acquire() method is used to acquire a lock. When a thread invokes acquire() for a lock, it gains ownership of the lock if it is available. If the lock is currently held by another thread, the calling thread will enter a blocking (waiting) state until the lock becomes available. Subsequently, when the thread successfully acquires the lock, it is able to execute the critical section of code that should be accessed by a single thread at a time. This attribute records the frequency of this method being called for a lock within a specific function.
(6) **Number of release() Function Calls** (number-of-released-locks): The release() method is employed to release a lock that the calling thread currently possesses. Upon the completion of the critical section of code protected by the lock, the thread is expected to invoke lock.release() to release the lock. This action enables other waiting threads to acquire the released lock. This attribute quantifies the instances of this method being called for a lock within a specific function.

3.3 API Misuses

API misuse refers to the incorrect use of an Application Programming Interface (API), which violates the implicit usage constraints set by the API. These constraints are in place to prevent errors and exceptions that can occur when the API is not used as intended. API misuse is a common cause of software bugs, crashes, and vulnerabilities [2, 9, 14, 20]. Guoliang Jin et al., [9] assert that more than one-quarter of software bugs are linked to API misuses. Based on our empirical study, we have categorized API misuses associated with performance-bugs into three distinct groups: Object-Relational Mapping (ORM) APIs, Deep Learning APIs, and Machine Learning Cloud API misuses. However, for the scope of this study, we focus exclusively on examining ORM API misuses, with a specific emphasis on Django ORM API misuses. We prioritize Django ORM because our projects primarily involve Python. The exploration of the other two categories is deferred to future research, as each requires a separate study to accurately identify the necessary features for precise characterization.

When it comes to Django ORM[1] API misuses there are some common misuses which can cause performance issues and need to be avoided [18]:

- Making complex queries: The more complex the query, the harder it is for the ORM to transform it into an actual database query. This can cause performance regressions, especially when dealing with large datasets. To avoid this, it is recommended to keep queries as simple as possible and avoid using sub-queries unless absolutely necessary.
- Retrieving too much data: When querying the database, it is vital to fetch only the required data. Retrieving excessive data can result in performance problems, particularly with large datasets. In Django ORM, there is a built-in method called select-related() that allows you to retrieve all relevant data in a single query instead of making multiple database queries. However, it is important to exercise caution and avoid overusing this method, as it can lead to fetching too much data and cause performance regression.
- Not using database-level constraints: Sorting records after fetching data from the database instead of sorting the database once can cause performance regressions. It is recommended to use database-level constraints instead of using order-by() to sort records at the query level.

Following three features represent the occurrence of three Django ORM functions which overusing them could potentially lead to one one of the performance issues mentioned above.

(1) **extra()** (number-of-usage-of-extra): Overusing extra() can make queries more complex by using sub-queries.
(2) **order_by()** (number-of-usage-of-order-by): Overusing order by() instead of using database-level constraints can lead to performance issues.
(3) **select_related()** (number-of-usage-of-select-related): Overusing select related() can also lead to retrieving too much data.

4 ADAPTIVE LOGGING SYSTEM (ALS)

To address the research limitation mentioned earlier, this study proposes an Adaptive Logging System, which utilizes Reinforcement Learning to provide a comprehensive log-placement framework. The adaptiveness of the system offers two key benefits: (1) it allows for the definition of any desired logging objective by modeling it into the RL's reward function, and (2) it ensures project independence of the framework by eliminating low cross-project accuracy.

ALS takes Python source code files as input and guides developers in identifying which functions in the provided source code should be logged to capture performance issues, along with the recommended log-level. In Figure 2, we can observe that ALS comprises three primary modules: web scraping, feature extraction, and an RL model. We will explore each of these modules in greater

[1] https://docs.djangoproject.com/en/3.2/topics/db/optimization/

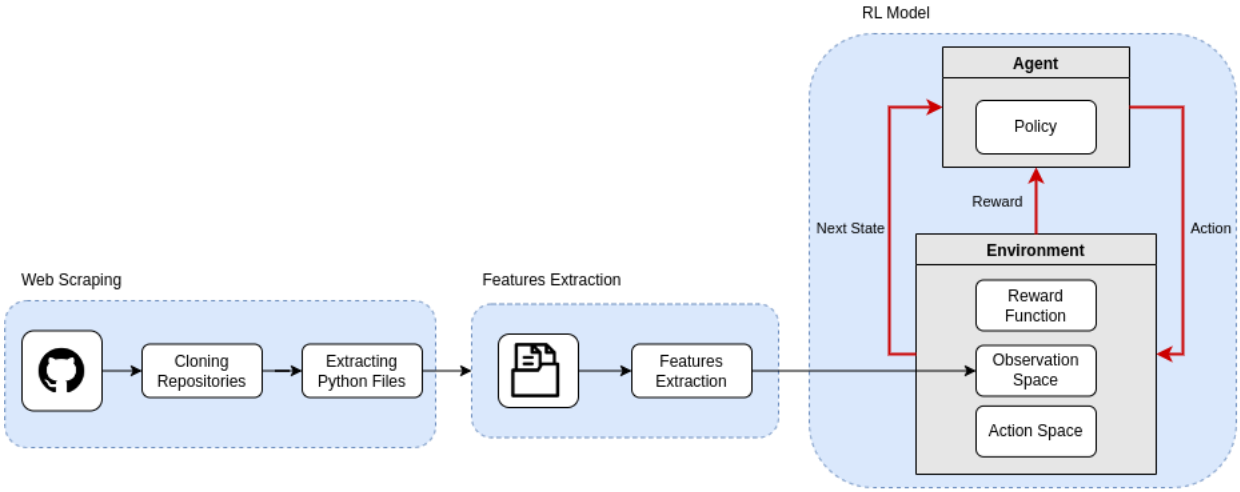


Figure 2: Adaptive logging system overview

detail in the subsequent parts of this section. The source code and training dataset for ALS are accessible via our git repository [2].

4.1 Web Scraping

Since ALS relies on an RL model at its core, the collection of a substantial amount of data is crucial for effective model training. Additionally, ALS is specifically designed for Python, requiring primary source code files to be in .py format. Manual data collection becomes impractical due to this requirement. The ALS web scraping module is designed to address these challenges.

This module's task is to download Python files from a given list of GitHub repositories using git command-line tool (PyGithub[3]). It accomplishes this in two main steps:

(1) clones each of the GitHub repositories to a specified local directory.
(2) Then, it scans through these cloned repositories to identify Python files and saves them in a separate directory.

4.2 Feature Extraction and Data Collection

This module has the important task of carefully extracting the features we defined in our previous discussion in part 3. However, successfully implementing it presents a significant challenge. This challenge revolves around obtaining the necessary access to comprehensively analyze the components of the source code.

To overcome this challenge, we have utilized Python's Abstract Syntax Tree (**AST**) library[4]. This advanced library provides us with the tools to explore the complex structures of the source code. This allows us to gain the required perspective to extract the specific static features accurately and with precision.

The process of building our dataset by collecting and extracting relevant features involves three distinct phases. In the following, we offer a comprehensive explanation of each of these crucial steps in our dataset preparation process.

Function Extraction (Phase 1). In the first step, we identify and isolate individual functions from the source code of the projects under investigation. For this, the feature extraction module takes the source code as input, generates its AST, and searches "FunctionDef" nodes in the generated tree to find the functions in the source code. The module do the same thing to every Python file obtained through web scraping, resulting in a comprehensive list of function nodes that serve as the foundation for our subsequent analysis and feature extraction.

Function Identification (Phase 2). Subsequently, in the second step, we assign meaningful and distinctive identifiers to each extracted function. This step is essential to facilitate seamless access to the origin source code which these functions belong to, laying the groundwork for subsequent in-depth investigations.

We follow a three-part approach: combining the function's name, the name of the file where the function is located, and its position (index) in the list of functions. Then, we compute the hash value of this combined string using the "hashlib" library[5]. This process generates the final, unique identifier (ID) for the function.

Feature Extraction (Phase 3). The third and final step involves the extraction of relevant features from the previously mentioned functions. These extracted features are then integrated into our dataset. Importantly, each row within the dataset is dedicated to a single function, along with its corresponding set of extracted features, ensuring a comprehensive and organized representation.

With the list of function nodes obtained from the preceding phase, we iterate through this list, conduct feature searches, extract the identified features, and seamlessly incorporate them into the

[2]https://github.com/amirmahdiKhosravi/Adaptive-Logging-System
[3]https://pygithub.readthedocs.io/en/stable/introduction.html
[4]https://docs.python.org/3/library/ast.html
[5]https://docs.python.org/3/library/hashlib.html

dataset. The methodology employed for this process remains consistent across all the features, involving a systematic traversal of the AST.

4.3 RL Model

The RL model is the core component of ALS. It enables ALS to adapt itself to different defined logging objectives through trial and error. It is divided into two main parts: the Agent and the Environment, which interact with each other to make decisions and learn from their interactions. This section focuses on detailing the various components of the RL model.

Environment. It is where the agent operates. It includes everything outside the agent and serves as the backdrop for the agent's actions and interactions.

To accurately describe the environment, it is crucial to identify its key components: the observation space, action space, and reward function. By precisely defining these three essential elements, we enable the RL model to make informed decisions and take suitable actions within this well-defined environment.

- **Observation Space** Observation space is the part that models the space in which the agent interacts. In our study, it is to demonstrate functions within the source code files. To represent functions as our observation space we take advantage of the collected dataset in 4.2, as each row of it represents a function by its related features. Equation 1 shows the observation space of the environment in the timestep t (S_t). It is a 1-dimensional Box[6] containing 12 features of our dataset. Also, our environment here is deterministic, meaning that the probability of the agent ending up in next state (S_{t+1}), while it is in S_t taking action A_t, is 1 (Equasion 2). The next state here is the next function of our dataset that the agent needs to take an action upon. Upon reaching the end of the dataset and the last function (terminal state), signifying the end of an episode, the agent seamlessly transitions back to the starting state of the environment—the first function of the dataset—to initiate a new episode of learning.

$$S_t = [feature_1, feature_2, feature_3, ..., feature_{12}] \quad (1)$$

$$P(S_{t+1}|S_t, A_t) = 1 \quad (2)$$

- **Action Space** It represents all the possible actions which the agent can take within the environment. In this study we defined a discrete action space which contains 5 different actions. These actions are defined in a way that makes it possible for the agent to make decision about either a function should be logged, and if so, what log-level should be chosen for it.
  Of the six existing log-levels (Trace, Debug, Info, Warn, Error, Fatal), we excluded Error and Fatal from the set of log-levels that our agent can select. This decision was based on the observation that Error and Fatal are primarily associated with runtime behavior, whereas our model relies solely on static source code features to make logging decisions.
  Equation 3 presents the action space of our environment. This set encompasses 5 distinct actions, each uniquely identified by a numerical assignment (Action ID). It is important to note that while there is an action labeled as "not-log," we deliberately omitted a separate "log" action. This decision was made because actions associated with specific log-levels inherently indicate the logging action. For example, selecting the "Trace" action implies that the corresponding function should be logged at the Trace level.

$$A = \{0 : not_log,\ 1 : Trace,\ 2 : Debug,\ 3 : Info,\ 4 : Warn\} \quad (3)$$

- **Reward Function** The reward function guides the agent toward the goal it aims to achieve through its actions. To appropriately define the reward function, we need to incorporate our logging goal—deciding whether to log a function and the appropriate log-level—into it. This is accomplished through three main steps:
  - **Step 1:** We establish our fundamental rules for achieving our logging goal. These rules include: 1) logging functions that exhibit potential performance-bugs by examining performance-bug features in the observation space, and 2) assigning lower verbosity log-levels to performance-bugs with higher significance. This ensures that we capture all important information about performance-bugs, even when monitoring logs with the lowest verbosity. For this purpose, we assigned Trace and Debug log-levels to Synchronization issues, and Info and Warn log-levels to API misuses and Loops, respectively. If a function does not fall into one of these categories, it will not be logged.
  - **Step 2:** Based on the rules established in the previous step, we classify the actions performed by the agent into three distinct categories: "Good", "Intermediate" and "Bad". This categorization is determined through a comprehensive assessment of potential action outcomes: 1) Good: In scenarios where the agent's decision fully aligns with the criteria defined in the previous step, we provide the agent with a positive reward. For example, when a function exhibits potential synchronization issues based on its feature set in the observation space, the correct action is "Trace". The agent's action is deemed "Good" if it also selects "Trace" as the action. 2) Bad: Conversely, if the agent opts to log a function that should not be logged, or fails to log a function that should be logged based on the features extracted from the observation space, it incurs a negative reward as a punitive measure. 3) Intermediate: Any scenario falling outside the "Good" or "Bad" classifications is deemed "Intermediate." For example, if the agent correctly decides to log a function but selects an incorrect log-level, or if the action should have been categorized as "IDK" but the agent selects a different action, it does not receive a positive reward, as its behavior remains suboptimal. However, the negative reward incurred in such cases is comparatively smaller than the penalty associated with "Bad" behavior.
  - **Step 3:** The agent's reward depends on the category of its action and falls within a range of -4 to +2. We offer a

[6] https://stable-baselines.readthedocs.io/en/master/index.html

detailed breakdown of this reward scale for each possible category (good, bad, and intermediate) in the following section: **1) Good Reward (R_g)**: When the agent's action is categorized as "good," it receives an immediate reward of +1 (dense reward), supplemented by a sparse reward ranging from 0 to 1. The amount of the sparse reward depends on the function's vulnerability to performance-bugs, which is determined by the features within the function (S_t). To compute the sparse reward, we employ the Sigmoid function, with the coefficient providing adjustability for the slope of the Sigmoid curve (detailed elaboration is presented in chapters 4 and 5). Equation 5 shows the reward function for this specific category, denoted as R_g . ST is the sum of the 12 features in the vector S_t. **2) Bad Reward (R_b)**: The reward for the "bad" category is relatively straightforward, with the agent receiving the lowest reward within the range. In instances where the agent's action is categorized as "bad," it incurs a negative reward of -4 ($R_b = 4$). **3) Intermediate Reward (R_i)**: It is calculated as the negative absolute value of the difference between the action ID of the chosen action and the action that was intended to be selected. For instance, if the intended action is "Trace" (action ID = 1), but the agent selects "Warn" (action ID = 4), the reward will be -3. This relationship is depicted in Equation 6.

$$S_T = \sum_{f=0}^{11} S_t[f] \tag{4}$$

$$R_g(S_T) = 1 + \frac{1}{2}\left(\frac{1}{1 + e^{-\gamma S_T}}\right) \tag{5}$$

$$R_i = -|supposedActionID - takenActionID| \tag{6}$$

Agent. In this study, our primary focus is not on creating new RL algorithms. Instead, we choose to use well-established RL algorithms that already exist and apply them in our custom-designed environment. To make this possible, we rely on the Stable Baselines library, known for its high-quality implementations of Reinforcement Learning algorithms, all of which are based on OpenAI Baselines6 . Therefore, our approach involves making use of these pre-built and thoroughly developed algorithms from the Stable Baselines library. In part 5, dedicated to evaluation, we perform a comparative analysis to identify the most effective algorithm for our specific application.

5 EVALUATION

Our goal in this part is to thoroughly assess the practical usefulness and effectiveness of the Adaptive Logging System (ALS) in real-world scenarios by subjecting it to real-world software projects ranging from large-scale web applications to more specialized software, reflecting the diverse challenges faced by industry practitioners, such as those at Ciena. We will specifically explain how it performs in two distinct case studies, each offering its own set of challenges and opportunities.

In the upcoming sections, we detail our evaluation process. Section 5.1 explains our experimental setup, covering data collection, choice of RL algorithms, and libraries. Section 5.2 is the core, examining each case study, including ALS performance utilizing different RL methods and evaluations across projects. These sections showcase ALS performance in diverse scenarios.

Table 2: Dataset information

		Project Names	Number of Python files	Number of Functions
Training	Apache Projects	- Kibble - Libcloud - Allura - Spark	1421	56755
	Django Projects	- Connect - Chat-app - TrackTV		
Testing	Apache Projects	- Avro - Beam - Cloudstack - IoTDB - PLC4X - Thrift - Yetus	2814	36729
	Django Projects	- Django Website - Djangogirls Website - Django-jet		
Total			4235	93484

5.1 Experiment Setup

In this study, we provide a detailed account of our data collection process, shedding light on the datasets that serve as the foundation for our case studies. Then we shift our focus to configuring the RL model, using established RL algorithms from the Stable-Baselines library[7], which is implemented based on the OpenAI Baselines[8].

Data Collection. To prepare our dataset for training and testing the RL model of ALS, we have utilized the first two modules of the ALS framework. We conducted our experiments on a diverse set of 17 projects (Table 2), encompassing 11 Apache projects and 6 Django-based projects. The inclusion of Django projects was a deliberate choice, aimed at introducing diversity into our dataset, given that certain features, such as Django ORM API Misuses, tend to be less prevalent in Apache projects. These projects were selected based on their continued active status, ensuring their reliability as representative samples of ongoing software development projects.

As illustrated in Table 2, our dataset is partitioned into two distinct groups: training and testing. The training set is dedicated to training the RL model and comprises a larger volume of data, with 56,755 unique functions. The Testing dataset, explored in forthcoming sections, serves as the basis for our second case study and contains 36,729 functions. This approach not only ensures that our RL model is well-prepared with an extensive training dataset but also facilitates the assessment of its cross-project performance, as both datasets encompass a mix of Apache and Django projects. This diverse dataset allows us to evaluate the adaptability and effectiveness of our RL model on a wide array of software projects.

RL Model and Algorithms. With our experiment dataset in place, the next step involves configuring the RL model. For its environment we use our own environment which we defined in 4.3. On the other hand, for the agent, we use well-established RL algorithms available in the Stable-Baselines library.

[7]https://stable-baselines.readthedocs.io/en/master/index.html
[8]https://github.com/openai/baselines

Table 3: Stable-Baselines algorithms and features

	A2C	ACER	ACKTR	DDPG	DQN	GAIL	PPO	SAC	TD3	TRPO
Discrete Action Support	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	Yes
Box Observation Support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
On/Off Policy	On	On	On	Off	Off	Off	On	Off	Off	On

For our RL model, we selected Deep Q-Network (DQN), Advantage Actor-Critic (A2C), and Proximal Policy Optimization (PPO) based on their proven efficacy in complex decision-making tasks. DQN's stability in discrete action spaces, A2C's balance between policy and value-based methods, and PPO's robustness in varying environments make them ideal for evaluating ALS's performance in log-placement.

Table 3 presents an overview of the built-in model-free RL algorithms integrated into the stable baselines framework. Within this set of algorithms, there are a total of 11 options at our disposal. However, due to the discrete nature of our action space, three of these algorithms, namely DDPG, SAC, and TDT, are not compatible.

It is noteworthy that among the remaining nine algorithms, only DQN and GAIL operate as off-policy algorithms, while the remainder are on-policy methods. This distinction is pivotal as it affects the manner in which these algorithms update their policies based on historical data.

This experimental setup, encompassing a diverse dataset and a range of RL algorithms, is specifically designed to test ALS's core objectives. The varied dataset ensures ALS's adaptability to different software architectures, while the selection of RL algorithms allows us to assess the system's effectiveness in making accurate log-placement decisions under varying conditions.

5.2 Case Studies

To evaluate feasibility of ALS and to see whether it fulfills our research goals we defined two different case studies which we introduce and explain in more details in the following. We used the reward received by the model as the evaluation metric, which we illustrate by demonstrating the learning curve of the RL model.

ALS and Different RL Methods. This case study is to assess how well ALS performs with various combinations of RL algorithms. For this we test the RL model by setting three different RL methods (DQN, A2C and PPO) to its agent and demonstrate the learning curve of each of them while they are applied on the training environment (dataset). This case study is to evaluate the following aspects of the ALS:

- The effectiveness and functionality of the ALS in adapting itself to the defined logging objective.
- A comprehensive assessment of the performance of various RL algorithms, shedding light on how effective suitable each of them are for our environment.

Cross-project Evaluation. In this case study we apply the three agents that have been previously trained on the training environment on the testing environment which includes functions from projects that the agents have not been introduced to. This case study is to evaluate the cross-project performance of the ALS.

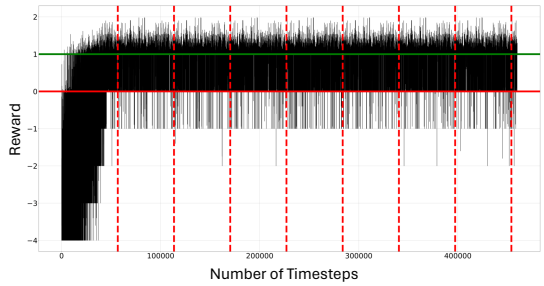


Figure 3: RL model's learning curve for DQN

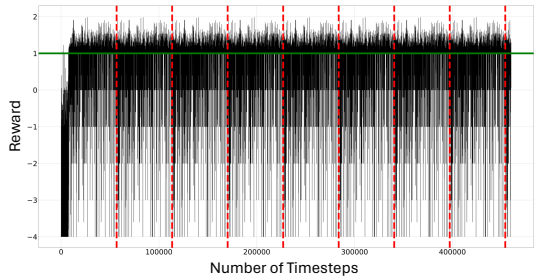


Figure 4: RL model's learning curve for A2C

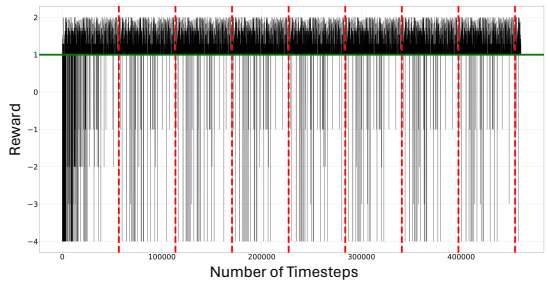


Figure 5: RL model's learning curve for PPO

5.3 Results and Discussion

Figures 3, 4 and 5 illustrate the outcomes of our first case study for DQN, A2C, and PPO, respectively. The broken vertical lines mark the end of each episode. Each model interacted with the environment across eight episodes. Initially, in the early timesteps, all three RL methods incurred negative rewards, signifying their initial struggle to make appropriate logging decisions, resulting in negative rewards. However, by the end of the first episode, all three models display a positive trend in reward values, showing

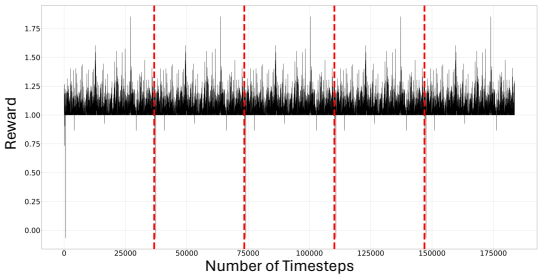


Figure 6: Evaluating DQN on testing environment

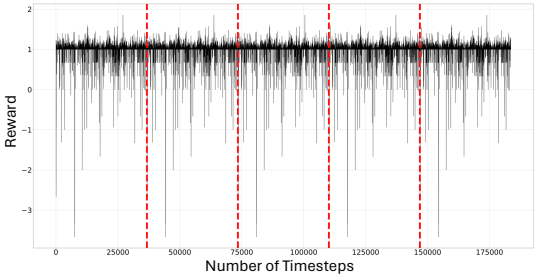


Figure 7: Evaluating A2C on testing environment

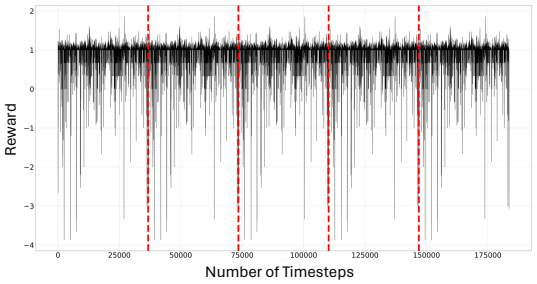


Figure 8: Evaluating PPO on testing environment

their capability to learn effective logging decisions and receive positive rewards. Notably, among the examined RL methods, DQN demonstrates the most stable results.

The results of our second case study are presented in Figures 6, 7 and 8. Observing the results, all three models exhibit a positive reward trend from the initial episode, even though they are being evaluated in a testing environment containing functions not encountered during their training. This underscores the reliable cross-project performance of ALS. The rationale behind this lies in the nature of RL methods, distinct from supervised learning; RL methods do not attempt to imitate the behavior of the training data or projects they encounter. Instead, they leverage this data to acquire more generalized knowledge applicable across diverse environments. In this context, DQN demonstrates the most stable results in the second case study.

Further analysis of these results reveals insights into the adaptability and effectiveness of different RL methods within ALS. The superior stability of DQN, for instance, suggests that its approach to learning and decision-making is particularly well-suited for the complexities of log-placement in varied software projects. This stability is critical when deploying ALS in real-world environments, where consistent performance is key to maintaining system reliability and efficiency. Additionally, the positive reward trends across all models highlight ALS's overall robustness and potential as a scalable solution for diverse logging needs. These findings are significant for industry applications, where adaptable and reliable logging strategies are essential for optimizing system performance and minimizing downtime, particularly in resource-constrained settings like those encountered by companies such as Ciena.

In summary, our evaluation results demonstrate: 1) ALS's functionality and its ability to adapt to a predefined logging objective, 2) its reliable cross-project performance, and 3) the compatibility of DQN as the most stable RL method for our environment, outperforming A2C and PPO in terms of stability.

The adaptability and effectiveness of ALS in diverse settings, as demonstrated by our evaluations, are particularly pertinent for our industrial partner, Ciena. In their resource-limited environments, the ability of ALS to dynamically adjust log-placement strategies is critical for maintaining system efficiency and reliability. These attributes of ALS not only meet the specific requirements of Ciena but also exemplify the system's potential for broader application in similar industrial contexts, where flexible and efficient logging solutions are critical.

5.4 Limitations

While our proposed Adaptive Logging System framework, utilizing Reinforcement Learning, effectively achieved its predefined logging objective of identifying performance-bugs, we identified certain limitations during the evaluation. These limitations open avenues for further enhancements:

(1) **Covered Programming Languages**: At present, ALS is limited to use in Python projects. This limitation arises from the tool[9] we use to search the Abstract Syntax Tree of the source code, which is designed specifically for Python. To enhance the feature extraction module of ALS, future studies could explore either using more comprehensive tools that support other programming languages or leveraging Large Language Models (LLMs) such as GPT-4[10], along with feature extraction models like Jina[11].

(2) **Dynamic Performance Features**: The current iteration of ALS focuses on specific performance-related logging objectives. Future versions could benefit from incorporating dynamic performance metrics, such as CPU and memory

[9] https://docs.python.org/3/library/ast.html
[10] https://openai.com/gpt-4
[11] https://huggingface.co/jinaai/jina-embeddings-v2-base-code

overhead, which were not explored in this study. Integrating these metrics into the RL model's reward function could enable ALS to tackle more complex performance logging objectives, enhancing its applicability in various software environments.

(3) **RL Model Architecture**: In our study, ALS employs a single RL agent for both logging decisions and log-level determination. Future research might explore a dual-agent approach, with one agent dedicated to logging decisions and another to log-level determination. This two-tiered approach could provide more precise control and potentially improve overall effectiveness.

These limitations highlight areas for potential improvement and demonstrate the evolving nature of adaptive logging systems in the field of software performance engineering.

6 CONCLUSION, AND FUTURE DIRECTIONS

In this research, we developed the Adaptive Logging System, a novel logging framework that utilizes Reinforcement Learning to adapt dynamically to varying logging objectives, with a focus on identifying and mitigating performance bugs. Our approach began with an empirical study categorizing performance bugs into three main types: Synchronization issues, Loops, and API misuses. This classification guided the definition of 12 distinct static source code features, which formed the basis of our dataset. Training ALS's RL model on this dataset, we evaluated its performance across 17 Django and Apache projects. The results were promising, demonstrating ALS's effectiveness in adapting to different logging objectives. Notably, the Deep Q-Network (DQN) model showed the most stable results in terms of learning curve, performing well in both training and cross-project evaluation scenarios.

The adaptability and robustness of ALS, while particularly beneficial for Ciena in their resource-limited settings, extend its significance to a wider range of industrial applications. The system's ability to tailor its logging strategies is not only essential for Ciena's operational efficiency and reliability but also indicative of its potential impact in broader software logging and performance evaluation contexts.

However, this research study has certain limitations. The selection of RL algorithms and the scope of our dataset, while extensive, may not fully capture the array of scenarios in different software environments. Moreover, our focus was primarily on function-level decisions, without delving into more detailed log-level determinations.

Looking forward, there are several opportunities to expand this research. One avenue of interest is the potential integration of ALS with LLMs, where ALS could serve as the RLHF (Reinforcement Learning from Human Feedback) reward model to optimize LLMs for logging solutions. Another possibility is integrating ALS with existing logging frameworks, which could lead to a more comprehensive logging solution. By combining ALS's dynamic adaptability with the proven capabilities of language models and traditional systems, this approach could greatly enhance the overall effectiveness of software logging practices.

We further collaborated with them to develop an adaptive logging system for their resource constrained environment. This paper discusses our first steps in that direction, both defining a logging system that solves their challenges and evaluating that system on open source systems. This work is defined as a first gate towards testing a system in production. In such a project, ALS's effectiveness in operational environments would be validated more accurately, and essential feedback from users and developers would be gathered. This feedback is vital for fine-tuning ALS, ensuring it meets the complex demands of real-world applications. Future plans in this regard include extensive deployment and evaluation within Ciena's operational context, aiming to demonstrate the system's practical utility and inform further development.

Further development could also involve incorporating dynamic performance metrics such as CPU and memory usage into the RL model, offering a more detailed understanding of system demands. Moreover, exploring a dual-agent architecture in the RL model—dividing responsibilities between logging decisions and log-level determination—could potentially refine the system's precision and efficiency.

REFERENCES

[1] Alam, M. M. U., Liu, T., Zeng, G., and Muzahid, A. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), pp. 298–313.

[2] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering 45*, 12 (2018), 1170–1188.

[3] Cao, J., Chen, B., Sun, C., Hu, L., Wu, S., and Peng, X. Understanding performance problems in deep learning systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 357–369.

[4] Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D., and Xie, T. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering* (2014), pp. 24–33.

[5] Gholamian, S. Leveraging code clones and natural language processing for log statement prediction. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 1043–1047.

[6] Graesser, L., and Keng, W. L. *Foundations of deep reinforcement learning*. Addison-Wesley Professional, 2019.

[7] Gu, R., Jin, G., Song, L., Zhu, L., and Lu, S. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 426–438.

[8] He, P., Chen, Z., He, S., and Lyu, M. R. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), pp. 178–189.

[9] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices 47*, 6 (2012), 77–88.

[10] Kim, T., Kim, S., Park, S., and Park, Y. Automatic recommendation to appropriate log levels. *Software: Practice and Experience 50*, 3 (2020), 189–209.

[11] Li, H., Shang, W., and Hassan, A. E. Which log level should developers choose for a new logging statement? *Empirical Software Engineering 22* (2017), 1684–1716.

[12] Mastropaolo, A., Pascarella, L., and Bavota, G. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 2279–2290.

[13] Mizouchi, T., Shimari, K., Ishio, T., and Inoue, K. Padla: a dynamic log level adapter using online phase detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)* (2019), IEEE, pp. 135–138.

[14] Ren, X., Ye, X., Xing, Z., Xia, X., Xu, X., Zhu, L., and Sun, J. Api-misuse detection driven by fine-grained api-constraint knowledge graph. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), pp. 461–472.

[15] Sandoval Alcocer, J. P., Bergel, A., and Valente, M. T. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (2016), pp. 37–48.

[16] Song, L., and Lu, S. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 370–380.

[17] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Yang, J., Subramaniam, P., Lu, S., Yan, C., and Cheung, A. How not to structure

your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering* (2018), pp. 800–810.

[19] Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., and Savage, S. Be conservative: Enhancing failure diagnosis with proactive logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 293–306.

[20] Zhang, Y., Kabir, M. M. A., Xiao, Y., Yao, D., and Meng, N. Automatic detection of java cryptographic api misuses: Are we there yet? *IEEE Transactions on Software Engineering 49*, 1 (2022), 288–303.

[21] Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., and Zhou, Y. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 565–581.

[22] Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., and Zhang, D. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 1, IEEE, pp. 415–425.

Time Series Forecasting of Runtime Software Metrics: An Empirical Study

Federico Di Menna
federico.dimenna@graduate.univaq.it
University of L'Aquila
Italy

Luca Traini
luca.traini@univaq.it
University of L'Aquila
Italy

Vittorio Cortellessa
vittorio.cortellessa@univaq.it
University of L'Aquila
Italy

ABSTRACT

Software applications can produce a wide range of runtime software metrics (*e.g.,* number of crashes, response times), which can be closely monitored to ensure operational efficiency and prevent significant software failures. These metrics are typically recorded as time series data. However, runtime software monitoring has become a high-effort task due to the growing complexity of today's software systems. In this context, time series forecasting (TSF) offers unique opportunities to enhance software monitoring and facilitate proactive issue resolution. While TSF methods have been widely studied in areas like economics and weather forecasting, our understanding of their effectiveness for software runtime metrics remains somewhat limited.

In this paper, we investigate the effectiveness of four TSF methods on 25 real-world runtime software metrics recorded over a period of one and a half years. These methods comprise three recurrent neural network (RNN) models and one traditional time series analysis technique (i.e., SARIMA). The metrics are gathered from a large-scale IT infrastructure involving tens of thousands of digital devices. Our results indicate that, in general, RNN models are very effective in the runtime software metrics prediction, although in some scenarios and for certain specific metrics (*e.g.,* waiting times) SARIMA proves to outperform RNN models. Additionally, our findings suggest that the advantages of using RNN models vanish when the prediction horizon becomes too wide, in our case when it exceeds one week.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software evolution**; **Extra-functional properties**; • **Computing methodologies → Neural networks**.

KEYWORDS

time series forecasting, software monitoring, runtime software metrics

ACM Reference Format:
Federico Di Menna, Luca Traini, and Vittorio Cortellessa. 2024. Time Series Forecasting of Runtime Software Metrics: An Empirical Study. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645049

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645049

1 INTRODUCTION

As software systems grow in complexity, the task of ensuring software quality becomes increasingly challenging. Today software systems constantly evolve, with frequent daily software releases [46], and they operate under highly variable workloads [5], which make them susceptible to unforeseen software failures [5, 55, 58]. In such a dynamic environment, traditional proactive strategies, such as software testing, are often insufficient for ensuring consistent operational efficiency [45, 58]. For this reason, monitoring is emerging as a key activity for maintaining operational efficiency of software systems [12, 23, 32].

Modern software applications can produce large volumes of runtime metrics, which are typically stored as time series data in specialized databases [23], (*e.g.,* Prometheus [15]). Dedicated monitoring teams continuously analyze these time series to identify and mitigate potential software issues [12]. However, the vast volume of collected data can make manual analysis costly and potentially ineffective. To address this challenge, researchers started to develop automated techniques that can facilitate the identification of software issues or aid in the debugging process [1, 6, 16, 19, 25, 53].

Despite these advancements, significant opportunities in the realm of data analysis remain unexploited. Time series forecasting (TSF), in particular, presents a promising avenue for enhancing the current practices in software monitoring. Indeed, the ability to predict future trends in runtime software metrics could facilitate the adoption of proactive measures, potentially preventing significant software failures or optimizing resource allocation. Runtime software metrics are notoriously difficult to analyze, due to their inherent instability [5, 20, 38, 54]. However, recent advancements in TSF have demonstrated its successful application across diverse fields, including economics [7, 50], meteorology [11], and healthcare [48]. Moreover, TSF has recently begun to gain attention also in the software domain [3, 10, 31, 35]. For instance, Amin *et al.* [3] employed AutoRegressive Integrated Moving Average (ARIMA) models to predict reliability a software system based on testing results. Krishna *et al.* [35] used time series analysis to forecast bug reports and enhancement requests in software projects. Bauer *et al.* [10] proposed the use of TSF in self-aware systems.

Despite these efforts, to date there is still little knowledge about the effectiveness of TSF methods for the prediction of runtime software metrics. With this paper, we aim to fill this gap by presenting an empirical assessment of multiple TSF methods on runtime software metrics. The study aims at addressing the following research questions:

RQ$_1$ *How effective are TSF methods when applied to predict short-term runtime software metrics?*

RQ$_2$ *Do TSF methods exhibit diverse forecasting accuracy over different classes of runtime software metrics?*

RQ$_3$ *To what extent does forecasting accuracy degrade when applied to predict longer-term runtime software metrics?*

To answer our research questions, we investigate the forecasting accuracy of four TSF methods, including a seasonal autoregressive moving average model (SARIMA) and three different recurrent neural networks, namely: fully-connected recurrent neural networks (FC-RNN), long-short memory networks (LSTM), and gated recurrent unit networks (GRU). The evaluation is performed on 25 real-world runtime software metrics that are categorized into 3 classes (*i.e., crash rate*, *hang time*, and *waiting time*) and gathered from 8 different software applications, resulting in a total of 14,575 individual data points. These metrics were recorded over a period of one and a half years on the large-scale IT infrastructure of a company[1], which comprises thousands digital devices. In this study, we distinguish between *short-* and *long-term* software metrics. This distinction refers to the distance of the forecasting target with respect to the current week of the software metric under analysis.

We show that, overall, RNN models are more effective than SARIMA and naive baselines, with FC-RNN providing the best forecasting accuracy. Nonetheless, our evaluation underscores the lack of a "silver bullet" method that outperforms all others across the considered metrics. For instance, we found that when dealing with specific classes of metrics, such as application waiting times, SARIMA offers the most accurate prediction. Furthermore, we found that benefits of using RNN models are valuable until the forecasting horizon does not exceed approximately one week.

The main contribution of this work are:

- A first empirical assessment of TSF methods when applied to runtime software metrics.
- An investigation of how different TSF methods behave when applied to different classes of metrics.
- A sensitivity analysis of TSF on runtime software metrics while varying forecasting horizons.

Paper Structure. The remainder of this paper is organized as follows. Section 2 provides background on TSF within and outside the software domain. Section 3 outlines the experimental design used for our empirical study. In Section 4, we detail our research questions along with the corresponding findings. Section 5 discusses potential threats to validity, and Section 6 concludes this paper.

2 BACKGROUND

Over the years, a vast variety of phenomena have been captured and modeled by leveraging the concept of time series, which can be defined as an ordered sequence of data points gathered at regular time intervals. Time series forecasting (TSF) approaches aim at predicting future evolution of the variable of interest by learning from its historical data while looking for patterns, trends, and seasonality in the data. Due to the easy porting of the time series concept, the effectiveness of TSF methods has been widely investigated in diverse areas, such as medicine, environment, system engineering, finance and more [3, 8, 17, 22, 44, 47, 50, 63]. Forecasting techniques can be differentiated between one-step-ahead and multistep-ahead ones. In the first case, the goal is to predict just the next value of the time series, whereas multiple values are predicted simultaneously in the second case. Generally, the number of forecast values that are generated at a time is denoted as the *forecasting horizon*.

Earliest techniques relied on straightforward heuristics as they evolved over time in more complex statistical models, such as Exponential Smoothing [13] and Autoregressive Integrated Moving Average (ARIMA) models [14]. The wide popularity of machine learning techniques, also driven by the increasing availability of data, has induced a strong interest in artificial neural networks (ANNs) for this purpose [27]. Some studies also explored the combination of statistical models and deep learning models, for instance Zhang et al. leveraged statistical models to fit the linear part of the time series and ANN to model the residual part [65].

Usually, a time series is fed into a neural network by creating consecutive shifted input windows to predict the datapoint(s) that follows the input sequence, thus modeling the task as a supervised learning problem. As dealing with sequential data is a very frequent task in machine learning, several neural network architectures have been expressly designed for sequence prediction, such as recurrent neural networks (RNN). Examples of most popular RNN are Long Short-Term Memory (LSTM) [26, 28, 47] and Gated Recurrent Unit (GRU) [18] models. Despite the extensive employment of TSF techniques, the *No Free Launch Theorem* [62] denies the possibility to build and select a single method that outperforms all the others across all time series, and for this reason a comprehensive evaluation of several models is often needed for the forecasting effectiveness.

Motivated by the necessity of studying the behavior of software applications over time [24, 56], and driven by the vast amount of data produced by modern software applications, researchers have started to apply TSF methods in software contexts, with the goal of anticipating potential issues that may arise from software evolution [3, 10]. For instance, Jia et al. [31] applied a hybrid prediction framework, namely DGRU, to predict software aging and determine the optimal rejuvenation time. Krishna et al. [35] proposed a model to predict the number of bug reports and enhancement requests that are going to be generated in the next month by exploiting the history of issue reports. Amin et al. [3] investigated the employment of ARIMA models in order to predict software reliability as an alternative solution to fight the Software Reliability Growth Models (SRGMs) limitations. Other researchers have explored the use of TSF for enabling proactive autonomous decisions in self-aware systems. For instance, Bauer et al. [10] conducted a preliminary investigation into how TSF methods can be integrated within self-aware systems. Additional research has investigated the exploitability of TSF in predicting Quality of Service (QoS) attributes [30, 52, 60]. For example, Syu et al. [52] conducted an empirical study on TSF methods applied to web services quality attributes.

Albeit considerable work has been carried out on the application of TSF methods in specific software contexts, their application on runtime software metrics and the derived benefits/drawbacks in a broader monitoring environment should be more thoroughly investigated. To fill this gap, we conducted an empirical evaluation

[1]Due to privacy concerns, the company choose not to reveal itself.

of TSF methods on several runtime software metrics (by varying nature) gathered from a large IT infrastructure.

3 EXPERIMENT DESIGN

In this section, we describe our experiment design, including the dataset, the TSF methods (and baselines) we studied, and the specific experimental procedures we used to gather metric predictions.

3.1 Dataset

Our dataset consists of 25 distinct runtime software metrics collected from the IT infrastructure of a large company, which includes more than 30k digital devices with heterogeneous hardware and software configurations. For each of the 25 metrics, the dataset provides a time series of daily observations collected over a period of one and a half years, thus resulting in a total of 583 measurements per metric. Each metric (*i.e.,* times series) concerns to a specific runtime aspect of a particular software application. For example, one metric might represent the *crash rate* of a specific web browser, while another could indicate the *hang time* of a particular email client. The dataset encompasses 8 distinct software applications from a diverse array of types, including web browsers, communication platforms, and word processors. Each application was monitored on 3 different runtime aspects, which we refer to as *metric classes*. In the following, we describe the semantics of each metric class.

- *Crash rate* denotes the average number of observed crashes of an application per hour of use. It is calculated by dividing the total number of observed application crashes within the IT infrastructure by the cumulative hours of application usage across all devices.
- *Hang time* represents the percentage of application usage time during which the application remains in a "hang" state, *i.e.,* when the application is unresponsive and not performing any active processing tasks. This runtime aspect can be critical for diagnosing potential issues within software applications.
- *Waiting time* is defined as the time users spend in waiting for an application to respond while it is actively running. This metric encompasses periods when the application is unresponsive, known as the "hang" state, as well as application and network loading times. It reports the percentage of the total application usage time that is spent in a "waiting" state across all devices within the IT infrastructure.

As a result, the dataset contains 24 time series (*i.e.,* 3 for each of the 8 software applications), plus an additional one that reports the *crash rate* related to operating system failures, such as *blue screen of death* [36] or *kernel panics* [49].

Due to privacy concerns, we cannot make publicly available the dataset used in our study.

3.2 Models and Baselines selection

For our empirical study, we selected four TSF methods, including one autoregressive moving-average model and three recurrent neural network models. We chose these types of models because they have been successfully applied to time series analysis in previous research [3, 22, 26, 31]. Specifically, our selection comprises the following TSF methods: (i) Seasonal Autoregressive Integrated Moving Average (SARIMA) [59], (ii) Fully Connected Recurrent Neural Network (FC-RNN) [40, 64], (iii) *Long Short-Term Memory* (LSTM) [28] and (iv) *Gated Recurrent Unit* (GRU) [18].

SARIMA is an extension of the autoregressive integrated moving-average (ARIMA) model [14] that addresses the seasonal fluctuations often observed in time series data. It is based on the integration of additional seasonal terms, which allow the model to capture both short- and long-term dependencies, as well as repetitive patterns that occur at fixed intervals. This versatility makes it suitable for forecasting time series where the data exhibit periodicity.

FC-RNN is a variant of neural networks where the outputs are fed back into the network as inputs, thus forming directed cycles. This creates a recurrent connection pattern that allows the network to maintain a state that can theoretically hold information about previous inputs indefinitely. The term "fully connected" denotes that each neuron in a given layer is connected to every neuron in both the preceding and subsequent layers.

LSTM is a specialized form of RNN designed to address the challenge of learning long-term dependencies. Unlike standard RNNs, LSTMs include a series of gated cell states that regulate the flow of information. These gates control the persistence and updating of information within the cell state. This architecture allows LSTMs to effectively retain important information over extended sequences while discarding irrelevant data, thus making them particularly suitable for TSF.

GRU is a particular form of RNN, introduced to solve the vanishing gradient problem inherent to traditional RNNs. GRUs simplify the LSTM approach by combining the forget and input gates into a single "update gate" [18], while also merging the cell state and hidden state. This results in a more streamlined model that requires fewer parameters without sacrificing performance.

To aid the interpretability of results, we compare the forecasting accuracy of TSF models with the ones of two naïve baseline models, namely *seasonal naïve* (sNaïve) and *seasonal monthly mean* (sMM).

sNaïve operates on the assumption of temporal recurrence, by repeating the last observed values for future forecasts. In other words, this method projects the most recent seasonal data forward to the next equivalent season, thus assuming cyclical repetition. For instance, when predicting values for the upcoming week, the method simply replicates the observations from the previous week. This method is commonly used as baseline for comparative analysis in TSF studies [9, 22, 27, 34, 43, 52].

sMM leverages recent seasonal weekly trends to forecast future values. Specifically, it computes predictions by calculating the mean of data points that correspond to the same weekday in the preceding month. For example, to forecast the value for an upcoming Monday, sMM would average the values from all Mondays in the last month. Domain experts from the company that provided the dataset have recommended this baseline approach, as it reflects the established analytical practices of their software monitoring team.

3.3 Experimental procedure

To account for the distinct characteristics of the TSF methods investigated in our study, we adopt two distinct procedures tailored specifically for the RNN and SARIMA models, respectively.

SARIMA evaluation. For SARIMA, we independently create one model instance for each of the 25 time series (*i.e.,* runtime software metrics). The fitting process in SARIMA models consists of estimating the $(p, d, q)(P, D, Q)_s$ parameters, where: p and P represent the order of the autoregressive (AR) terms for the non-seasonal and seasonal parts respectively; d and D denotes the degree of differencing needed to render the series stationary on both non-seasonal and seasonal levels; q and Q indicate the order of the moving average (MA) terms for the non-seasonal and seasonal components; and s denotes the seasonality period of the time series data. To estimate the d and D parameter, we considered the number of times we applied differencing to obtain a stationary time series according to the ADFuller [21] test. The best values of (p, q, P, Q) have been searched by fitting the model with all the values ranging from 0 to 3 for each parameter, and by selecting the parameters configuration that gave the lowest *Akaike Information Criterion* (AIC) [2, 51] value after the model fitting. We set the s parameter to 7, which corresponds to a one week seasonality. The fitting process is performed using the initial 467 consecutive measurements of the time series (*i.e.,* 80% of the data points), with the remaining 20% of the measurements reserved for assessing the forecasting accuracy of the model. For the evaluation, starting from the initial input window of 467 consecutive measurements, we progressively increase the input window by one time unit and use the model to forecast the subsequent 14 measurements. This methodology enables us to simulate a realistic scenario of progressive forecasting, in which the model is continuously tested as new data becomes available in the time series. As a result, we obtain for each time series a set of 89 *forecast segments* F, each consisting of 14 predicted measurements.

RNN evaluation. To train the RNN models, instead, we employ a *sliding window segmentation* technique [29], which divides a longer time series of measurements into smaller, overlapping segments of fixed size. Specifically, we use a sliding window of 28 consecutive measurements (*i.e.,* 4 weeks) to generate multiple overlapping segments. The initial 14 measurements serves as the input segment (or *look-back* period), whereas the remaining measurements form the *forecast segment*. Following the best practice for deep-learning models [42, 63], we scaled the data using the *z-score* normalization technique, thus ensuring that the data falls within a comparable range. Specifically, we standardize each of the 28 measurements within the window segment by using the process outlined in Equation (1):

$$z_i = \frac{x_i - \mu_{input}}{\sigma_{input}} \tag{1}$$

where z_i is the standardized value of the i^{th} data point of the window segment, x_i is the original data point value, μ_{input} is the mean of the input segment measurements, and σ_{input} is their standard deviation. We use the mean and standard deviation of the input segment, instead of the entire window of 28 measurements, to ensure that our evaluation process reflects realistic forecasting scenarios, where future measurements (*i.e.,* the forecast segments) are not yet known.

To evaluate a RNN model, we independently train one model instance for each of the 25 time series, by using the initial 72% consecutive window segments (*i.e.,* 394 segments) for training, the subsequent 8% for validation (*i.e.,* 19 segments), and the remaining 20% for testing (*i.e.,* 89 segments). For the RNN models under consideration (*i.e.,* FC-RNN, LSTM, GRU), the neural network architecture is designed as follows. The first layer of the architecture consists of 14 units of the specific RNN type (either FC-RNN, LSTM, or GRU). The architecture includes a second layer composed of another 14 units of the same type, which is encapsulated within a bidirectional layer. Finally, the processed information is channeled through a fully connected layer with 14 units, by employing the hyperbolic tangent (tanh) activation function to produce the final output. We use the Adam optimizer [33] with an initial learning rate of 0.001, using the mean absolute error (MAE) as the loss function. Additionally, we implement a reduce-on-plateau strategy, which decreases the learning rate when no improvement is observed in the validation loss for 20 epochs. The training is conducted for a maximum of 500 epochs, with an early stopping mechanism that terminates the process if no improvement in the validation loss is observed for 50 consecutive epochs. The best model is selected basing on the lowest validation loss observed throughout the training epochs. As a result of the evaluation process, similarly to SARIMA, we obtain for each RNN model and time series a set of forecast segments F, each consisting of 14 measurements.

Notation. For notational convenience, we use F^i to denote the forecast segment that aims to predict the time series data points beginning from the i^{th} position. For instance, F^{500} represents the forecast segment predicting data from the 500^{th} to the 513^{th} position of the time series. Additionally, we use F_j to denote the j^{th} element of the forecast segment, with $1 \leq j \leq 14$. Figure 1 graphically illustrates the example presented above (F^{500}). The black circles represent the data used as input by the models and the red ones are the predicted measurements.

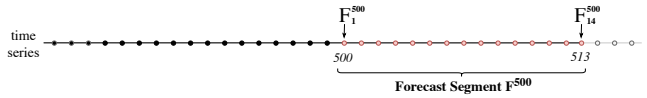


Figure 1: Forecast segment within the time series.

Ultimately, for each TSF method and time series, we obtain 89 forecast segments F^i, with $482 \leq i \leq 570$, where 482 denotes the starting position of the first forecast segment appearing in the time series, and 570 denotes the starting position of the last forecast segment of the time series.

Implementation details. The experiment has been implemented in *Python*. The `SARIMAX` class of *statsmodels*[2] library has been used for implementing SARIMA models. RNNs have been implemented using `SimpleRNN` (for FC-RNN models), `LSTM` and `GRU` classes of *Tensorflow*[3].

[2]https://www.statsmodels.org/stable/index.html
[3]https://www.tensorflow.org/

4 RESEARCH QUESTIONS AND FINDINGS

In this section, we describe in detail the research questions and experimental results of our empirical study. For sake of clarity, we address each research question in a separate subsection. Each subsection is structured as follows: we first outline the objective of the research question, then we describe the methodology used to gather the answers, and finally we discuss the results.

4.1 RQ_1: How effective are TSF methods when applied to predict short-term runtime software metrics?

Objective. With the first research question, we want to study the effectiveness of TSF methods in predicting short-term runtime software metrics. Specifically, our focus is on evaluating the accuracy of these methods in "one-step-ahead" forecasting, which entails predicting runtime software metrics for the immediate next day.

Methodology. To address this research question, we employ a commonly used metric for measuring forecasting accuracy, namely SMAPE (Symmetric Mean Absolute Percentage Error).

The values of SMAPE range from 0% to 200%, where 0% indicates perfect forecasting accuracy, while 200% represents the worst accuracy. Given that the aim of this research question is to evaluate the forecasting accuracy of TSF methods in predicting next-day metrics, we calculate the SMAPE by focusing exclusively on the first element of each forecast segment F^i, *i.e.,* the next-day prediction.

Formally, given a time series Y and a particular TSF method, we calculate the corresponding SMAPE as follows:

$$\text{SMAPE} = \frac{100\%}{n-k+1} \sum_{t=k}^{n} \frac{\left|F_1^t - Y_t\right|}{\frac{1}{2}\left(\left|F_1^t\right| + \left|Y_t\right|\right)} \tag{2}$$

where k and n denote the positions of the first and last elements of the time series used for evaluation (specifically, k=482 and n=513 in our case). F_1^t denotes the first element of the forecast segment F^t (*i.e.,* the next-day prediction of the TSF method for the t^{th} element of the time series), and Y_t denotes the actual time series measurement at position t.

As a result of this process, for each TSF method, we obtain 25 SMAPE results, *i.e.,* one per time series.

In order to assess (and compare) the forecasting accuracy of different TSF methods, we plot the SMAPE distribution of each TSF method using box plots, and report the associated descriptive statistics (*e.g.,* mean, median). Additionally, we conduct an analysis to determine whether each TSF method outperforms the naive baselines. Indeed, the practical applicability of a TSF method may be questioned if it does not improve upon these baselines. To accomplish this, we employ the Wilcoxon signed-rank test [61] for each pair of <*TSF method, baseline*> to compare their respective SMAPE values. We set the significance level at 0.05, meaning that differences with p-values below this threshold are considered statistically significant. In addition to the Wilcoxon signed-rank test, we utilize the common language effect size [39], in the version proposed by Vargha and Delaney ($\hat{A}_{12}$) [57], to assess the magnitude of the differences observed. Given two related paired samples X and Y, the common language effect size is the proportion of pairs where X is higher than Y.

$$\hat{A}_{12} = P(X > Y) + .5 \times P(X = Y) \tag{3}$$

The $\hat{A}_{12}$ value, which ranges from 0 to 1, is interpreted using the thresholds provided by Vargha and Delaney [57]. A $\hat{A}_{12}$ value of 0.5 suggests that there is no significant difference in forecasting accuracy between the TSF method and the baseline. A value of $\hat{A}_{12}$ larger than 0.5 indicates that the TSF method is likely to yield more accurate forecasts than the baseline, *i.e.,* lower SMAPE. Specifically, the magnitude of the difference is considered as *small* (S^+), *medium* (M^+) and *large* (L^+) if the $\hat{A}_{12}$ value is greater than or equal to 0.56, 0.64 and 0.71, respectively. Conversely, a value of $\hat{A}_{12}$ lower than 0.5 indicates that the TSF method yields worse forecasting accuracy than the baseline. In this case, the effect size is considered as *small* (S^-), *medium* (M^-) and *large* (L^-) if the $\hat{A}_{12}$ value is lower than or equal to 0.44, 0.34 and 0.29, respectively. We consider comparisons that report $\hat{A}_{12}$ larger than 0.44 and lower than 0.56 as negligible, and therefore not meaningfully different.

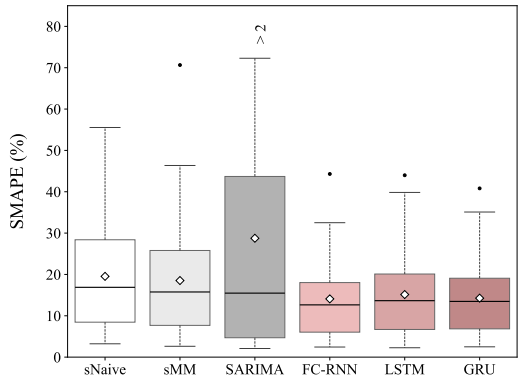


Figure 2: RQ_1. SMAPE distribution for each TSF Method. Boxplots are highlighted in red for RNN models, grey for SARIMA and white/silver for the baselines.

	SMAPE Statistics				
	Mean (μ)	Median ($\tilde{x}$)	Std Dev (σ)	Min	Max
sNaïve	19.53	16.88	14.23	3.23	55.54
sMM	18.52	15.76	15.77	2.62	70.65
SARIMA	28.75	15.48	33.02	2.09	116.95
FC-RNN	14.09	12.64	10.31	2.44	44.30
LSTM	15.14	13.65	10.94	2.27	43.99
GRU	14.26	13.48	9.91	2.49	40.82

Table 1: RQ_1. SMAPE descriptive statistics for TSF methods and baselines.

Results. We reported the results concerning the SMAPE distribution of TSF methods in Figure 2 and Table 1. A first observation is that RNN-based methods demonstrate promising forecasting accuracies, as evidenced by their average SMAPE values: 14.09% for FC-RNN, 15.14% for LSTM, and 14.26% for GRU. Among them,

	sNaïve	sMM
SARIMA	0.50 (-), p=0.596	0.48 (-), p=0.164
FC-RNN	**0.63 (S^+), p=<0.001**	**0.57 (S^+), p=0.001**
LSTM	**0.59 (S^+), p=<0.001**	0.55 (-), p=0.027
GRU	**0.60 (S^+), p=<0.001**	**0.56 (S^+), p=0.006**

Table 2: RQ_1. Results of the comparison between TSF methods and baselines. Each cell is formatted as "$\hat{A}_{12}$, *p-value*", where $\hat{A}_{12}$ represents the Vargha-Delaney effect size, and the p-value is the result of the Wilcoxon signed-rank test. The interpretation of the $\hat{A}_{12}$ value is also provided in brackets. Comparisons where TSF methods outperform baselines with statistical significance (*p-value*< 0.05) and a non-negligible effect size are highlighted in bold.

FC-RNN appears to be the most effective, by providing the lowest median and mean SMAPE values. These results are comparable to, or even better than, those reported in recent TSF research [9, 10, 37]. From Figure 2, it can be observed that RNN-based methods (shown in red in Figure 2) demonstrate better forecasting accuracy than baselines, as indicated by their lower SMAPE values. In support of this, we also notice in Table 1 that RNN models show considerably lower mean and median SMAPE values than those provided by baselines. For example, if we compare the worst-performing RNN model in terms of mean and median SMAPE (*i.e.,* LSTM) with the best-performing baseline (namely, sMM) we still observe an improvement in mean (18.52% versus 15.14%) and median (15.76% versus 13.62%) values. Another interesting result is that RNN-based methods exhibit higher stability in forecasting accuracy when compared to other approaches, *i.e.,* the prediction error tends to vary less from one time series to another. Indeed, as shown in Table 1, RNN-based methods exhibit lower standard deviations in SMAPE. Specifically, among RNN-based methods we observe a maximum standard deviation of 10.94% (LSTM), which is notably lower than the 14.23% and 15.77% standard deviations observed for sNa"ive and sMM baselines, respectively. This observation is remarked by Figure 2, which displays a narrower inter-quartile range (IQR) for RNN-based methods that is also more shifted towards the bottom, thus indicating lower errors. This suggests that RNN-based methods provides more accurate and stable prediction than baselines. Further confirmation of this finding comes from the results of the Wilcoxon signed-rank test presented in Table 2. Both FC-RNN and GRU demonstrate statistically significant improvements over the baselines ($p < 0.05$), with a non-negligible effect size ($\hat{A}_{12} \geq 0.56$). Additionally, LSTM outperforms sNaive with a small effect size. These results indicate considerable benefits in employing RNN for TSF of short-term runtime software metrics.

An analysis of Figure 2 reveals that SARIMA results in the widest SMAPE IQR, thus indicating significant variation in its forecasting accuracy. This observation is further supported by the data in Table 1, which shows SARIMA having the highest standard deviation, at 33.02%, among all evaluated methods. Moreover, SARIMA provides the worst average forecasting accuracy, with an average SMAPE of 28.75%. While these results might suggest that SARIMA is poorly suited for predicting short-term runtime software metrics, a closer examination of its SMAPE distribution reveals some interesting insights. For example, as shown in Table 1, SARIMA achieves the lowest minimum SMAPE value (2.09%) and its median SMAPE (15.48%) is lower than those of the naive baselines. Additionally, Figure 2 shows that SARIMA provides the lowest first quartile in SMAPE. These observations indicate that SARIMA performs quite well on a specific subset of time series, potentially even outperforming other approaches on these instances.

Summary. RNN-based methods are more effective in predicting short-term runtime software metrics, with FC-RNN being the most effective one (average SMAPE of 12.64%). FC-RNN and GRU outperform both baselines with statistical significance ($p < 0.05$) with non-negligible effect sizes ($\hat{A}_{12} \geq 0.56$). SARIMA is less stable in forecasting accuracy, by exhibiting SMAPE values that significantly vary from one time series to another (standard deviation of 33.02%). To answer RQ_1, our analysis demonstrates that TSF methods, particularly RNN-based methods, are quite effective in accurately predicting short-term runtime software metrics. These findings highlight the potential benefits of applying TSF methods into real-world software monitoring contexts.

4.2 RQ_2: Do TSF methods exhibit diverse forecasting accuracy over different classes of runtime software metrics?

Objective. The second research question aims to assess the forecasting accuracy of TSF models for each class of runtime software metric (*i.e., crashes rate/hang times/waiting times*). We want to study whether TSF methods perform consistently across diverse metric classes or their accuracy varies significantly depending on the specific class of metric being forecasted.

Methodology. To address this research question, we reuse the SMAPE values previously calculated for RQ_1. However, rather than examining these values in aggregate, we group them per metric class. We investigate the forecasting accuracy of TSF methods across each metric class by analyzing their corresponding SMAPE distributions through box plots, and by examining the associated descriptive statistics. For each metric class, we also compare the forecasting accuracy of TSF methods with naive baselines using Wilcoxon signed-rank test. This is done for each *<TSF method, baseline>* pair to determine if there is a statistically significant difference in their SMAPE values. Additionally, we employ the Vargha-Delaney $\hat{A}_{12}$ to assess the effect size.

Results. The SMAPE distribution computed over all the metric classes is depicted in Figure 3, and the associated descriptive statistics are reported in Table 3. We examined the results in two ways: (i) by evaluating the general forecasting accuracy of TSF methods on individual software metric classes, and (ii) by analyzing how each TSF method behaves in relation to each software metric class.

With the first analysis, we want to study how TSF methods collectively perform when applied to different classes of metrics. In Figure 3, we observe that SMAPE box plots for same metric classes exhibit similar behavior across various TSF methods. For instance, when observing the SMAPE distribution related to *waiting time*, we notice significantly lower errors compared to those reported for other metric classes. This can be observed in both Figure 3 and Table 3: the IQR, mean, and median of SMAPE for *waiting*

		Baselines		TSF Methods			
		sNaive	sMM	SARIMA	FC-RNN	LSTM	GRU
Metric Class	SMAPE Stat						
Crash Rate	Mean (μ)	24.48	24.05	38.76	17.40	19.07	17.95
	Median ($\tilde{x}$)	18.55	15.76	18.24	13.56	14.70	14.97
	Std Dev (σ)	16.40	20.88	37.06	9.72	11.21	9.82
	Max	55.54	70.65	116.95	32.51	39.83	35.10
	Min	6.12	5.07	4.55	5.24	5.03	5.33
Hang Times	Mean (μ)	21.62	18.33	39.32	16.24	16.52	15.86
	Median ($\tilde{x}$)	19.02	16.61	34.91	13.44	14.01	13.40
	Std Dev (σ)	14.31	13.18	36.52	12.70	12.33	11.48
	Max	50.65	46.35	116.26	44.30	43.99	40.82
	Min	5.65	3.98	3.64	3.97	4.09	4.07
Waiting Times	Mean (μ)	11.87	12.50	6.91	8.20	9.34	8.53
	Median ($\tilde{x}$)	8.72	8.78	4.19	5.90	6.34	6.86
	Std Dev (σ)	8.85	9.98	5.77	6.01	7.38	6.03
	Max	28.39	29.99	17.28	18.04	21.77	19.08
	Min	3.23	2.62	2.09	2.44	2.27	2.49

Table 3: RQ$_2$. SMAPE descriptive statistics for TSF methods and baselines grouped by software metric class.

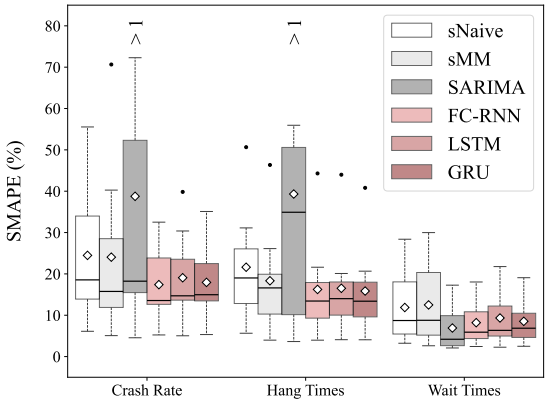


Figure 3: RQ$_2$. SMAPE distribution of TSF methods grouped by metric class.

time are consistently lower than those reported for other metric classes, regardless of the TSF method employed. This suggests that time series pertaining to the same metric classes may share common characteristics that influence the forecasting accuracy of TSF methods, thus leading to consistently better or worse outcomes. Indeed, time series related to *waiting time* appear significantly more predictable than those related to *crash rate* or *hang time*. Both *hang time* and *crash rate* show larger and more dispersed (*i.e.,* higher standard deviation σ) SMAPE values, with *crash rate* looking as the hardest to predict. A possible motivation for this result could be that *waiting time* metrics exhibit a seasonal pattern that recurs over time, while *crash rate* (or *hang time*) metrics are influenced by more unpredictable factors, such as unexpected software bugs or hardware malfunctions.

In our second analysis, we investigate the forecasting accuracy of various TSF methods for each class of runtime software metric. In Figure 3 we observe that RNN-based methods, *i.e.,* FC-RNN, LSTM, and GRU, perform generally well across the different metric classes. RNN-based methods show lower (or comparable) SMAPE distributions than those reported by naive baselines. This is confirmed by the results of the Wilcoxon signed-rank test, reported in Table 4. For instance, FC-RNN outperforms sNaive in all the metric classes with a statistically significant difference ($p < 0.05$) and a medium effect size ($\hat{A}_{12} \geq 0.64$). LSTM also outperforms sNaive with a small effect size ($\hat{A}_{12} \geq 0.56$) on *crash rate* and *hang time*. GRU, similarly, outperforms sNaive with small and medium effect sizes on *crash rate* and *hang time*, respectively. Compared to the sMM baseline, both LSTM and GRU report statistically significant lower SMAPE values on *hang time* and *waiting time*, with either small or medium effect sizes. Overall, FC-RNN demonstrates the best forecasting accuracy across the different metric classes, by providing statistically significant improvement over both naive baselines in all metric classes, except on *crash rate* when compared to sMM.

Another noteworthy result concerns the diverse forecasting accuracy provided by SARIMA over different metric classes. By examining Figure 3, we observe that SARIMA provides substantially different SMAPE values over different metric classes. For instance, while it reports relatively high errors for *crash rate* and *hang time* (*e.g.,* average of 38.79% and 39.32%, respectively), considerably low SMAPE values are reported for *waiting time* (*e.g.,* average of 6.91%). This diversity is remarked in the comparison with naive baselines. According to Table 4, SARIMA outperforms both sNaive and sMM on *waiting time* with statistically significant difference, and large and medium effect sizes, respectively. However, for other metric classes, SARIMA does not provide any improvement over the baselines. Even more, it provides worse forecasting accuracy than sMM on *crash rate*. Nonetheless, by analyzing both Figure 3 and Table 3, we can notice that SARIMA provides the best forecasting accuracy on *waiting time* among different TSF methods, with the lowest mean and median values (respectively, 6.91% and 4.19%). This may suggest that such method performs particularly well when dealing with more predictable time series that show recurring patterns, while it may not fit well on more irregular runtime software metrics, such as *crash rate* or *hang time*.

Summary. When collectively analyzed, TSF methods exhibit diverse forecasting accuracy depending on the class of runtime software metric. For instance, TSF methods show higher effectiveness when dealing with *waiting time* metrics, while they are less effective

TSF Method	Baseline	Software Metric Class		
		Crash Rate	Hang Times	Waiting Times
SARIMA	sNaive	0.42 (-), p=0.129	0.36 (-), p=0.312	**0.73 (L^+), p=0.008**
	sMM	**0.38 (S^-), p=0.008**	0.34 (-), p=0.148	**0.70 (M^+), p=0.008**
FC-RNN	sNaive	**0.65 (M^+), p=0.004**	**0.64 (M^+), p=0.008**	**0.69 (M^+), p=0.008**
	sMM	0.52 (-), p=0.496	**0.59 (S^+), p=0.008**	**0.64 (M^+), p=0.016**
LSTM	sNaive	**0.60 (S^+), p=0.012**	**0.62 (S^+), p=0.008**	0.61 (-), p=0.109
	sMM	0.53 (-), p=0.57	0.56 (-), p=0.055	0.59 (-), p=0.109
GRU	sNaive	**0.59 (S^+), p=0.02**	**0.64 (M^+), p=0.008**	0.61 (-), p=0.055
	sMM	0.52 (-), p=0.652	**0.58 (S^+), p=0.023**	**0.62 (S^+), p=0.023**

Table 4: RQ_2. Results of the comparison between TSF methods and baselines over each class of metric. Each cell is formatted as "$\hat{A}_{12}$, *p-value*", where $\hat{A}_{12}$ represents the Vargha-Delaney effect size, and the p-value is the result of the Wilcoxon signed-rank test. The interpretation of the $\hat{A}_{12}$ value is also provided in brackets. Comparisons where TSF methods outperform baselines with statistical significance ($p\text{-}value < 0.05$) and a non-negligible effect size are highlighted in bold.

on classes of metrics more closely related to software malfunctions, such as *hang times* and *crash rate*. RNN-based methods demonstrate the most consistent effectiveness across different metric classes, with FC-RNN being the most effective one. However, when dealing with classes of more predictable metrics, such as *waiting time*, SARIMA has been shown to be the most effective.

4.3 RQ_3: To what extent does forecasting accuracy degrade when applied to predict longer-term runtime software metrics?

Objective. The goal of this research question is to assess how forecasting accuracy decreases when predicting longer-term runtime software metrics. The TSF methods that we consider are able to generate multi-step ahead forecasts. This means that, at any given time, a TSF method can be queried to predict (for instance) the runtime software metrics of the next 14 days. It is expected that near-term predictions (*e.g.,* for the forthcoming days) will generally be more accurate than those for longer terms (*e.g.,* the following week). Through this research question, we aim to evaluate the extent to which the accuracy decreases as the forecasting horizon is progressively extended.

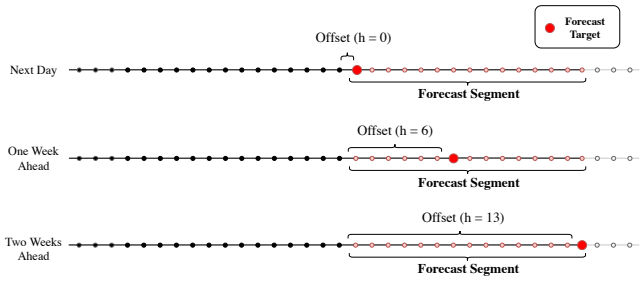


Figure 4: RQ_3. Three representative offset scenarios: next day ($h = 0$), one week ahead ($h = 6$), and two weeks ahead ($h = 13$).

Methodology. To achieve this research goal, we introduce the concept of *offset* (h), which we define as the number of days between the last time step used as input and the *forecast target*. This essentially simulates scenarios where, at a given moment, the goal is to predict the metric for a specific future day. For example, an offset $h = 0$ indicates a scenario where the forecast target is the next day, while an offset $h = 6$ corresponds to a scenario where the goal is to predict runtime software metrics for the same day in the following week. Figure 4 graphically illustrates the concept of offset and forecast target. The first scenario represents the case where the goal is to predict the metrics of next day ($h = 0$), the second scenario targets predictions for the same day in the following week ($h = 6$), and the third scenario uses the same day of two weeks ahead as forecast target ($h = 13$).

For each specific offset h, we calculate the SMAPE of a given TSF method applied to a particular time series Y as follows:

$$\text{SMAPE} = \frac{100\%}{n-k+1} \sum_{t=k}^{n} \frac{\left|F^t_{1+h} - Y_{t+h}\right|}{\frac{1}{2}\left(\left|F^t_{1+h}\right| + |Y_{t+h}|\right)} \tag{4}$$

where F^t_{1+h} represents the $(1+h)^{th}$ element of the forecast segment F^t, corresponding to the prediction for the $1+h$ steps ahead day (*i.e.,* the forecast target). Y_{t+h} indicates the actual measurement observed in the time series on that particular day. For example, with a $h = 6$ offset, SMAPE is calculated by considering exclusively the errors at the 7^{th} elements across all forecast segments of the time series. This corresponds to assessing the forecasting accuracy of a TSF method in predicting the runtime software metric for the same day in the following week.

As results of this process, for each TSF method, we compute 350 SMAPE values, corresponding to each combination of time series and offset h, with $0 \leq h \leq 13$.

We employ box plots and line plots to illustrate the relationship between SMAPE and the offset h. The analysis of these plots aids in gaining a better understanding of how forecasting accuracy degrades as the forecasting horizon is extended.

Results. Figure 5 displays the SMAPE distribution of each TSF method for three representative offset scenarios: next day (h=0), one week ahead (h=6), and two weeks ahead (h=13). Figure 6 shows the trend of the mean SMAPE of each TSF method as the offset h increases. Figure 7 reports the same information for each metric class, separately.

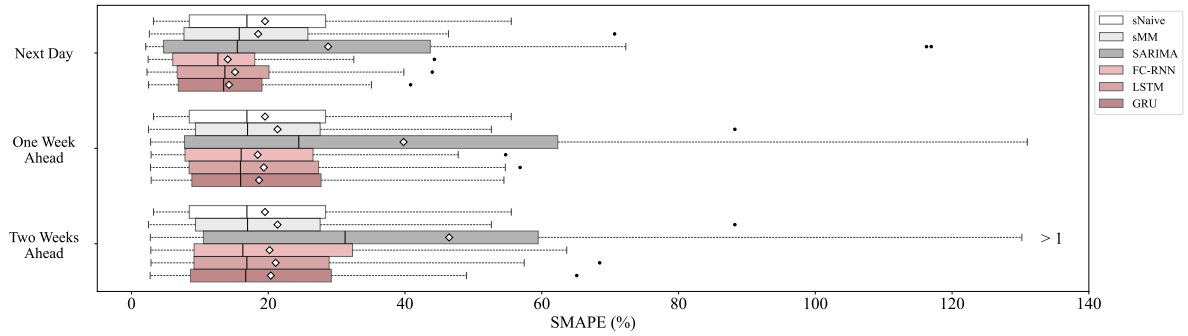
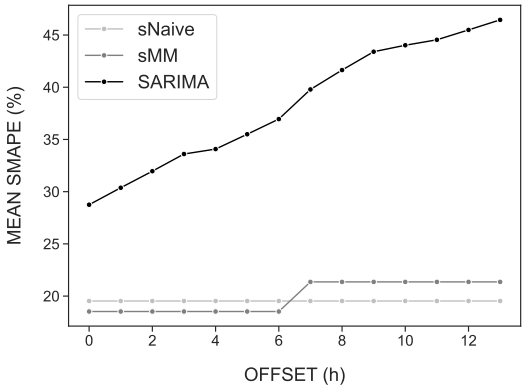
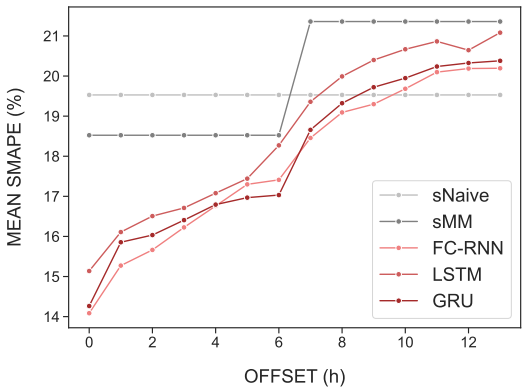


Figure 5: RQ$_3$. SMAPE distribution of TSF methods under three representative offset scenarios: next day ($h = 0$), one week ahead ($h = 6$), and two weeks ahead ($h = 13$).



(a) SARIMA vs baselines.



(b) RNNs vs baselines.

Figure 6: RQ$_3$. Relationship between mean SMAPE and offset (h).

Figure 6 clearly highlights an increasing trend in the mean SMAPE as the offset increases. This increasing trend is also observable in individual classes of metrics, as shown in Figure 7. By looking at Figure 5, we can also notice that the SMAPE tends to become more variable as the offset increases, thus indicating less stability in forecasting accuracy on long-term predictions. This is particularly visible in RNN-based methods, which show an IQR that consistently increases with each offset scenario.

Another interesting outcome of our analysis concerns the comparison of the forecasting accuracy of TSF methods with the baselines. Specifically, upon examining Figure 6, we notice a critical offset beyond which RNN-based methods begin to achieve comparable or even worse forecasting accuracy than the ones of baselines. This specific turning point is observed at approximately a week ahead ($h \approx 7$). A similar behavior is also observable on each individual class of runtime software metric, as shown in Figure 7. Nonetheless, each class of metric exhibits slightly different patterns. For instance, in Figure 7, we notice that mean SMAPE of RNNs begin to exceed those of baselines at offset $h = 9$ on *crashes rate*, instead RNNs start to exceed baselines error at offset $h = 6$ for *waiting times*, . This finding is also evident in Figure 5, where a clear degradation is observed in the SMAPE distribution, moving from the *next day* to the *one week ahead* scenario. Overall, these results suggest that the benefits of employing RNNs for predicting runtime software metrics vanish when the prediction target exceeds the current week. Our results highlight the importance of accounting for the changing dynamics of runtime software metrics, thus outlining the necessity of incorporating recent temporal patterns. In a practical software monitoring context, this emphasizes the need to regularly update predictions as new data becomes available.

Another interesting observation is that, albeit RNN models exhibit similar behavior, in some cases they start to exceed the baselines at different offsets. For example, by looking at Fig. 7, we noticed that LSTM start to exceed the sMM on *hang time* as early as offset is equal to $h = 3$, while for FC-RNN this happens at offset $h = 6$. Furthermore, on *hang times*, GRU outperforms the baselines across the entire offset range, although from offset $h = 6$ its SMAPE becomes very similar to that of sMM. This reveals that the choice of the specific RNN model can have non-trivial impact on the forecasting accuracy of longer-term runtime software metrics.

We then examined the accuracy of TSF models in detail for each class of software metrics. In line with the findings discussed in RQ$_2$, Figure 7 reveals that predictions for *waiting times* metrics consistently displayed lower errors than other metric classes, even in longer-term forecasts, with a maximum mean SMAPE of 16%. It is interesting to note that SARIMA, which is the best-performing

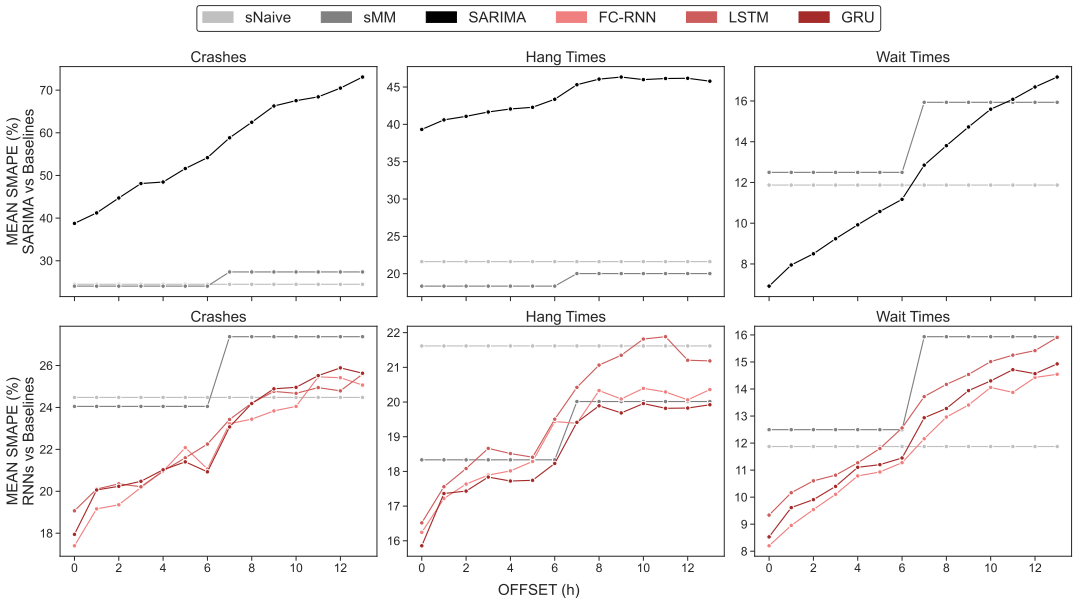


Figure 7: RQ$_3$. Relationship between mean SMAPE and offset (h) grouped by metric class. The first row displays results for SARIMA, while the second row shows the results of RNN-based methods.

method for *waiting time* according to RQ$_2$, shows here a similar behavior to that of RNN-based methods. In fact, we observe that SARIMA only begins to underperform the baselines when the offset exceeds one week ($h = 6$).

Summary. By answering RQ$_3$, our findings reveal that short-term forecasts are consistently more accurate than longer-term ones. Namely, the SMAPE values demonstrate a rising trend as the offset increases. Our results suggest that the advantages of using TSF methods vanish if the offset exceeds approximately one week. These findings offer practical insights into the suitability of TSF methods for predicting long-term runtime software metrics.

5 THREATS TO VALIDITY

Construct validity. A potential threat to construct validity in our study pertains to the selection of TSF methods. The choice of different TSF methods might yield different results. To mitigate this threat, we selected well-established TSF methods, by including a traditional autoregressive moving average model and three types of recurrent neural networks. Another threat concerns the choice of a metric for assessing the effectiveness of the TSF methods. The forecasting accuracy of the studied methods was evaluated using SMAPE, which is a widely accepted scale-independent error measure. However, it is important to note that other error metrics could partially alter the study outcomes.

Internal validity. The implementation of RNN and SARIMA models in our study was carried out by using *tensorflow* and *statsmodels*, respectively. The choice of these specific libraries might introduce biases that could potentially influence the study results. Nonetheless, these are well-established libraries in the field of data analysis. The choice of hyper-parameters can significantly affect the forecasting accuracy of TSF methods. Different hyper-parameter configurations might lead to different outcomes. To mitigate this, we tried to maintain consistent hyper-parameter settings across different TSF methods wherever feasible. For example, all RNN-based methods were configured with an identical number of layers and units, while utilizing the same optimizer. Furthermore, we applied uniform early stopping strategies and epochs.

External validity. The results of our empirical study may not generalize to other different runtime software metrics. However, our analysis was based on a dataset comprising 25 different runtime software metrics collected from 8 distinct software applications, which span three metric classes (*i.e.,* crash rate, hang times, and waiting times). To the best of our knowledge, there are no other TSF studies that have utilized a dataset involving such diversity in runtime software metrics.

6 CONCLUSION

In this empirical study, we investigated the effectiveness of popular TSF methods for predicting runtime software metrics. Our results demonstrate that: (i) TSF methods are indeed effective for short-term predictions, with RNN-based methods emerging as the most effective ones; (ii) no single method offers the best forecasting accuracy across all runtime software metrics; (iii) the benefits of applying TSF methods diminish when the forecasting horizon extends beyond the current week. We encourage future research to expand upon the depth and breadth of our study scope. For instance, future studies could delve into hyper-parameter tuning and explore further TSF methods, including multivariate ones that can simultaneously consider multiple runtime metrics. Additionally, given the rapid expansion of transfer learning in various domains [4, 41, 66], the investigation of the application of pre-trained TSF models for predicting runtime software metrics is a promising avenue for future research.

ACKNOWLEDGEMENTS

This work is partially supported by Italian Government (Ministero dell'Università e della Ricerca, PRIN 2022 PNRR): "RECHARGE: monitoRing, tEsting, and CHaracterization of performAnce Regressions" (cod.P2022SELA7), and by "ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing", funded by European Union – NextGenerationEU.

REFERENCES

[1] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1737–1749. https://doi.org/10.1109/ICSE48619.2023.00149

[2] H. Akaike. 1974. A new look at the statistical model identification. *IEEE Trans. Automat. Control* 19, 6 (1974), 716–723. https://doi.org/10.1109/TAC.1974.1100705

[3] Ayman Amin, Lars Grunske, and Alan Colman. 2013. An approach to software reliability prediction based on time series modeling. *Journal of Systems and Software* 86, 7 (2013), 1923–1932. https://doi.org/10.1016/j.jss.2013.03.045

[4] Ognjen Antonijevi, Slobodan Jeli, Branislav Bajat, and Milan Kilibarda. 2023. Transfer learning approach based on satellite image time series for the crop classification problem. *Journal of Big Data* 10, 1 (2023). https://doi.org/10.1186/s40537-023-00735-2

[5] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 405–417. https://www.usenix.org/conference/nsdi18/presentation/ardelean

[6] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. 2020. DeCaf: Diagnosing and Triaging Performance Issues in Large-Scale Cloud Services. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 201–210. https://doi.org/10.1145/3377813.3381353

[7] Wei Bao, Jun Yue, and Yulei Rao. 2017. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PLOS ONE* 12, 7 (07 2017), 1–24. https://doi.org/10.1371/journal.pone.0180944

[8] Oren Barkan, Jonathan Benchimol, Itamar Caspi, Eliya Cohen, Allon Hammer, and Noam Koenigstein. 2023. Forecasting CPI inflation components with Hierarchical Recurrent Neural Networks. *International Journal of Forecasting* 39, 3 (2023), 1145–1162. https://doi.org/10.1016/j.ijforecast.2022.04.009

[9] André Bauer, Marwin Züfle, Simon Eismann, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. 2021. Libra: A Benchmark for Time Series Forecasting Methods. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Virtual Event, France) *(ICPE '21)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/3427921.3450241

[10] André Bauer, Marwin Züfle, Nikolas Herbst, Albin Zehe, Andreas Hotho, and Samuel Kounev. 2020. Time Series Forecasting for Self-Aware Systems. *Proc. IEEE* 108, 7 (2020), 1068–1093. https://doi.org/10.1109/JPROC.2020.2983857

[11] Peter Bauer, Alan Thorpe, and Gilbert Brunet. 2015. The quiet revolution of numerical weather prediction. *Nature* 525, 7567 (2015), 47–55. https://doi.org/10.1038/nature14956

[12] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.".

[13] Baki Billah, Maxwell L. King, Ralph D. Snyder, and Anne B. Koehler. 2006. Exponential smoothing model selection for forecasting. *International Journal of Forecasting* 22, 2 (2006), 239–247. https://doi.org/10.1016/j.ijforecast.2005.08.002

[14] George.E.P. Box and Gwilym M. Jenkins. 1976. *Time Series Analysis: Forecasting and Control.* Holden-Day.

[15] Brian Brazil. 2018. *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring.* " O'Reilly Media, Inc.".

[16] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding Performance Problems in Deep Learning Systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 357–369. https://doi.org/10.1145/3540250.3549123

[17] Pousali Chakraborty, Marius Corici, and Thomas Magedanz. 2021. System Failure Prediction within Software 5G Core Networks using Time Series Forecasting. In *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*. 1–7. https://doi.org/10.1109/ICCWorkshops50388.2021.9473530

[18] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[19] Vittorio Cortellessa and Luca Traini. 2020. Detecting Latency Degradation Patterns in Service-Based Systems. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/3358960.3379126

[20] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. 2020. The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 67–75. https://doi.org/10.1145/3358960.3375791

[21] D. Dickey and Wayne Fuller. 1979. Distribution of the Estimators for Autoregressive Time Series With a Unit Root. *JASA. Journal of the American Statistical Association* 74 (06 1979). https://doi.org/10.2307/2286348

[22] Dimitrios Effrosynidis, Evangelos Spiliotis, Georgios Sylaios, and Avi Arampatzis. 2023. Time series and regression methods for univariate environmental forecasting: An empirical evaluation. *Science of The Total Environment* 875 (2023), 162580. https://doi.org/10.1016/j.scitotenv.2023.162580

[23] Liz Fong-Jones and Charity Majors. 2022. *Observability Engineering.* O'Reilly Media, Incorporated.

[24] E. Fuentetaja and D. Bagert. 2002. Software Evolution from a Time-Series Perspective. In *Proceedings of the International Conference on Software Maintenance (ICSM'02) (ICSM '02)*. IEEE Computer Society, USA, 226.

[25] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 145–155.

[26] Amirreza Heidari and Dolaana Khovalyg. 2020. Short-term energy use prediction of solar-assisted water heating system: Application case of combined attention-based LSTM and time-series decomposition. *Solar Energy* 207 (09 2020), 626–639. https://doi.org/10.1016/j.solener.2020.07.008

[27] Hansika Hewamalage, Christoph Bergmeir, and Kasun Bandara. 2021. Recurrent Neural Networks for Time Series Forecasting: Current status and future directions. *International Journal of Forecasting* 37, 1 (2021), 388–427. https://doi.org/10.1016/j.ijforecast.2020.06.008

[28] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[29] Robin John Hyndman and George Athanasopoulos. 2018. *Forecasting: Principles and Practice* (2nd ed.). OTexts, Australia.

[30] R. J. Hyndman, E. Wang, and N. Laptev. 2015. Large-Scale Unusual Time Series Detection. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE Computer Society, Los Alamitos, CA, USA, 1616–1619. https://doi.org/10.1109/ICDMW.2015.104

[31] Kai Jia, Xiao Yu, Chen Zhang, Wenhua Hu, Dongdong Zhao, and Jianwen Xiang. 2023. Software Aging Prediction for Cloud Services Using a Gate Recurrent Unit Neural Network Model Based on Time Series Decomposition. *IEEE Transactions on Emerging Topics in Computing* 11, 3 (2023), 580–593. https://doi.org/10.1109/TETC.2023.3258503

[32] Mike Julian. 2017. *Practical monitoring: effective strategies for the real world.* " O'Reilly Media, Inc.".

[33] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[34] Nikolaos Kourentzes. 2013. Intermittent demand forecasts with neural networks. *International Journal of Production Economics* 143 (05 2013), 198–206. https://doi.org/10.1016/j.ijpe.2013.01.009

[35] Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran, and Tim Menzies. 2018. What is the Connection between Issues, Bugs, and Enhancements? Lessons Learned from 800+ Software Projects. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 306–315. https://doi.org/10.1145/3183519.3183548

[36] Microsoft Learn. 2023. Blue screen data. https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/blue-screen-data 01.

[37] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. 2020. The M4 Competition: 100, 000 time series and 61 forecasting methods. *International Journal of Forecasting* 36, 1 (Jan. 2020), 54–74. https://doi.org/10.1016/j.ijforecast.2019.04.014

[38] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 409–425. https://www.usenix.org/conference/osdi18/presentation/maricq

[39] Kenneth O McGraw and Seok P Wong. 1992. A common language effect size statistic. *Psychological bulletin* 111, 2 (1992), 361.

[40] Engin Cemal Mengüç and Nurettin Acır. 2018. Kurtosis-Based CRTRL Algorithms for Fully Connected Recurrent Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 29, 12 (2018), 6123–6131. https://doi.org/10.1109/TNNLS.2018.2826442

[41] Erik Otović, Marko Njirjak, Dario Jozinović, Goran Mausa, Alberto Michelini, and Ivan Štajduhar. 2021. Intra-domain and cross-domain transfer learning for time series data – How transferable are the features? *Knowledge-Based Systems* 239 (12 2021), 107976. https://doi.org/10.1016/j.knosys.2021.107976

[42] Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. 2020. Deep Adaptive Input Normalization for Time Series Forecasting. *IEEE Transactions on Neural Networks and Learning Systems* 31, 9 (2020), 3760–3765. https://doi.org/10.1109/TNNLS.2019.2944933

[43] M. Peixeiro. 2022. *Time Series Forecasting in Python.* Manning.

[44] Francesco Piccialli, Fabio Giampaolo, Edoardo Prezioso, David Camacho, and Giovanni Acampora. 2021. Artificial intelligence and healthcare: Forecasting of medical bookings through multi-source time-series fusion. *Information Fusion* 74 (2021), 1–16. https://doi.org/10.1016/j.inffus.2021.03.004

[45] Casey Rosenthal and Nora Jones. 2020. *Chaos engineering: system resiliency in practice.* O'Reilly Media.

[46] Julia Rubin and Martin Rinard. 2016. The Challenges of Staying Together While Moving Fast: An Exploratory Study. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 982–993. https://doi.org/10.1145/2884781.2884871

[47] Alaa Sagheer and Mostafa Kotb. 2019. Time series forecasting of petroleum production using deep LSTM recurrent networks. *Neurocomputing* 323 (2019), 203–213. https://doi.org/10.1016/j.neucom.2018.09.082

[48] Jeffrey Shaman and Alicia Karspeck. 2012. Forecasting seasonal outbreaks of influenza. *Proceedings of the National Academy of Sciences* 109, 50 (2012), 20425–20430. https://doi.org/10.1073/pnas.1208772109 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1208772109

[49] D.P. Siewiorek and R.S. Swarz. 1998. *Reliable Computer Systems: Design and Evaluation.* A K Peters. https://books.google.it/books?id=EOrVzQEACAAJ

[50] James H. Stock and Mark W. Watson. 2002. Forecasting Using Principal Components from a Large Number of Predictors. *J. Amer. Statist. Assoc.* 97, 460 (2002), 1167–1179. http://www.jstor.org/stable/3085839

[51] Suhartono Suhartono. 2011. Time Series Forecasting by using Seasonal Autoregressive Integrated Moving Average: Subset, Multiplicative or Additive Model. *Journal of Mathematics and Statistics* 7 (01 2011), 20–27. https://doi.org/10.3844/jmssp.2011.20.27

[52] Yang Syu, Jong-Yih Kuo, and Yong-Yi Fanjiang. 2017. Time series forecasting for dynamic quality of web services: An empirical study. *Journal of Systems and Software* 134 (2017), 279–303. https://doi.org/10.1016/j.jss.2017.09.011

[53] Luca Traini and Vittorio Cortellessa. 2023. DeLag: Using Multi-Objective Optimization to Enhance the Detection of Latency Degradation Patterns in Service-Based Systems. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3554–3580. https://doi.org/10.1109/TSE.2023.3266041

[54] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2022. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering* 28, 1 (2022), 13. https://doi.org/10.1007/s10664-022-10247-x

[55] Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. 2021. How Software Refactoring Impacts Execution Time. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 25 (dec 2021), 23 pages. https://doi.org/10.1145/3485136

[56] W.M. Turski. 2002. The reference model for smooth growth of software systems revisited. *IEEE Transactions on Software Engineering* 28, 8 (2002), 814–815. https://doi.org/10.1109/TSE.2002.1027802

[57] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. http://www.jstor.org/stable/1165329

[58] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 635–651. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/veeraraghavan

[59] William Wei. 2006. *Time Series Analysis: Univariate and Multivariate Methods, 2nd edition, 2006.*

[60] Gary White, Andrei Palade, and Siobhán Clarke. 2018. Forecasting QoS Attributes Using LSTM Networks. In *2018 International Joint Conference on Neural Networks (IJCNN)*. 1–8. https://doi.org/10.1109/IJCNN.2018.8489052

[61] Frank Wilcoxon. 1992. *Individual Comparisons by Ranking Methods.* Springer New York, New York, NY, 196–202. https://doi.org/10.1007/978-1-4612-4380-9_16

[62] D.H. Wolpert and W.G. Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (1997), 67–82. https://doi.org/10.1109/4235.585893

[63] Hongju Yan and Hongbing Ouyang. 2018. Financial Time Series Prediction Based on Deep Learning. *Wireless Personal Communications* 102, 2 (01 Sep 2018), 683–700. https://doi.org/10.1007/s11277-017-5086-2

[64] Subhash Chand Yogi, Vibhu Kumar Tripathi, and Laxmidhar Behera. 2021. Adaptive Integral Sliding Mode Control Using Fully Connected Recurrent Neural Network for Position and Attitude Control of Quadrotor. *IEEE Transactions on Neural Networks and Learning Systems* 32, 12 (2021), 5595–5609. https://doi.org/10.1109/TNNLS.2021.3071020

[65] G. Peter Zhang. 2003. Time series forecasting using a hybrid ARIMA and neural network model. *Neurocomputing* 50 (2003), 159–175.

[66] Xiaofeng Zhou, Naiju Zhai, Shuai Li, and Haibo Shi. 2023. Time Series Prediction Method of Industrial Process With Limited Data Based on Transfer Learning. *IEEE Transactions on Industrial Informatics* 19, 5 (2023), 6872–6882. https://doi.org/10.1109/TII.2022.3191980

An Empirical Analysis of Common OCI Runtimes' Performance Isolation Capabilities

Simon Volpert
simon.volpert@uni-ulm.de
Ulm University
Institute of Information Resource Management
Ulm, Germany

Sascha Winkelhofer
sascha.winkelhofer@gini.net
Gini GmbH
Munich, Germany

Stefan Wesner
wesner@uni-koeln.de
University of Cologne
Cologne, Germany

Jörg Domaschka
joerg.domaschka@benchant.com
BenchANT GmbH
Ulm, Germany

ABSTRACT

Industry and academia have strong incentives to adopt virtualization technologies. Such technologies can reduce the total cost of ownership or facilitate business models like cloud computing. These options have recently grown significantly with the rise of Kubernetes and the OCI runtime specification. Both enabled virtualization technology vendors to easily integrate their solution into existing infrastructures, leading to increased adoption. Making a detailed decision on a technology selection based on objective characteristics is a complex task. This specifically includes the instrumentation of performance characteristics that are an important aspect for a fair comparison. Moreover, a subsequent quantification of the isolation capability based on performance metrics is not readily available.

In this paper, we instrument and determine the OCI runtime isolation capability by measuring virtualized system resources. We hereby build on two previous contributions, a proven isolation measurement workflow engine, and meaningful isolation metrics. The existing workflow engine is extended to integrate OCI runtime instrumentation as well as the novel isolation metrics.

We indicate a quantifiable distinction between the isolation capabilities of these technologies. Researchers and industry alike can use the results to make decisions on the adoption of virtualization technology based on their isolation characteristics. Furthermore, our extended measurement workflow engine can be leveraged to conduct further experiments with new technologies, metrics, and scenarios.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computing methodologies** → *Modeling methodologies.*

KEYWORDS

Isolation, Virtualization, Benchmarking, Framework, Container

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645044

ACM Reference Format:
Simon Volpert, Sascha Winkelhofer, Stefan Wesner, and Jörg Domaschka. 2024. An Empirical Analysis of Common OCI Runtimes' Performance Isolation Capabilities. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645044

1 INTRODUCTION

Virtualization technologies are consistently driving the vision of a software-defined infrastructure. Implementing virtualization is motivated by various factors, from facilitating business models like cloud computing to potentially reducing total ownership costs. Since the early days of Virtual Machines (VMs)[5], the landscape has been massively enriched by novel approaches such as containerization and other lightweight virtualization concepts. The rise and growing market share of the Kubernetes container orchestration engine, as well as the definition of the Open Container Initiative (OCI) specification, led to an increasing number of tools, methodologies, runtimes and engines in the domain of virtualization. This enabled virtualization technology vendors to implement their runtimes according to the OCI specification to be utilized interchangeably with orchestrators like Kubernetes [14]. For industry and academia alike, this vast number of options makes it difficult to objectively decide on what technology to utilize. The rapid expansion in this area makes it hard to stay abreast of all the most recent developments.

Comparing different virtualization technologies is a multidimensional decision problem with criteria ranging from security considerations, isolation capabilities, the type of virtualization, and many more. Various research studies are conducted on the comparison of virtualization technologies in several aspects, from impact on startup times[16], security considerations[15], to performance and isolation analysis[12].

Due to this mentioned rapid extension of the virtualization landscape, we need effective means to make informed decisions based on objective metrics. Multi-criteria decisions are typically complex and can hardly be reduced to a single metric [3]. Thus, we focus on the single distinct "isolation" metric to compare virtualization technologies against each other.

In systems with multiple competing workloads, isolation efficiency can be quantified by the impact that a disruptive workload

has on its competing but behaving counterpart. Therefore, it is necessary to measure the performance characteristics of the different resources that are contended. To objectively analyze these characteristics for different runtimes, the instrumentation across all technologies has to be done from a black-box perspective. It needs to acquire similar performance metrics for all technologies and requires low instrumentation overhead. We take advantage of extended Berkeley Packet Filter (eBPF) to get an unobstructed view of the performance characteristics. For the sake of acquiring similar metrics for all technologies and simplification of implementation, we focus on OCI compliant virtualization runtimes.

In this work, we answer the following research questions.

RQ 1 (instrumentation). *How can performance and derived isolation characteristics of OCI compatible runtimes be instrumented and subsequently measured?*

RQ 2 (automatability). *How can the instrumentation across different OCI runtimes be conducted in an efficient and uniform manner?*

RQ 3 (comparision). *How do the isolation capabilities among OCI compatible virtualization technologies compare?*

This paper uses an existing benchmark-based evaluation methodology that supports the instrumentation of performance degradation and the determination of isolation capabilities. More precisely, we present the following contributions.

C 1 (isolation determination framework). *We release the codebase of the evaluation framework including the extensions developed during the work on this paper [20]. This relates to* **RQ 1** *and* **RQ 2** *and enables fellow researcher to perform similar measurements for their usecases.*

C 2 (comparison). *We present a comparison (***RQ 3***) of three distinct OCI compliant virtualization technologies regarding their isolation capabilities.*

The remainder of this paper is structured as follows. In section 2 we briefly present the fundamentals of this work. This includes eBPF, OCI and a discussion of isolation and its quantification. This is followed by a description of the methodology in section 3 and lays the foundation for the answer to **RQ 1**. The methodology is followed by some important details of the implementation in section 4. It discusses the remaining aspects of answering **RQ 1** and additionally answers **RQ 2**. Section 5 gives a brief overview of the technologies involved in the experimental setup. This setup is used to generate the final results in section 6 which closes **RQ 3**. We finish with a discussion in section 7, a review of related work in section 8 and a final summary in section 9.

2 BACKGROUND

This section explains fundamentals that are essential for the further course of this work.

2.1 eBPF and Instrumentation

This section briefly highlights eBPF and Linux profiling. A more detailed description is available in the previous work of the fellow authors [2, 21].

eBPF enables the execution of verified code within a special VM that runs as part of the Linux kernel. It hereby extends the capabilities of the initially developed Berkeley Packet Filter (BPF) developed at a Berkeley Laboratory [13]. Apart from executing functions when receiving network packets, it can now observe and react to a multitude of event sources as part of the Linux profiling subsystem. Those specifically include Performance Monitoring Counters (PMCs), tracepoints, kernel, and user functions.

While these events are technically not part of eBPF, it still enables an approachable exploitation of them. The typical lifecycle of an eBPF program is visualized in Figure 1 as presented by Gregg [6].

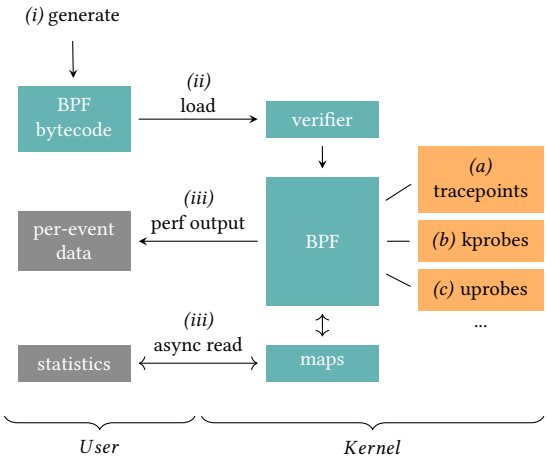


Figure 1: eBPF internals and Linux instrumentation according to [6]

The following paragraphs briefly describe the previously mentioned Linux profiling subsystem instrumentation points that are relevant to this work.

(a) *Tracepoints.* Tracepoints are static kernel instrumentation points[19]. They are defined and implemented by the kernel developers and issue an event once a specific call occurs. They further include counters that are specific to hardware, like CPU instrumentation.

(b) *Kprobes.* Kprobes are similar to tracepoints, yet not statically defined [7]. They allow dynamic hooks into any kernel function call. As this depends on the kernel function name, this is not stable across kernel releases.

(c) *Uprobes.* Similar to kprobes, uprobes can dynamically instrument user space function calls. In practice, this requires available debug symbols [4].

2.2 Open Container Runtime

The "OCI"[1] is part of the "Linux Foundation"[2] that develops open standards for container-based virtualization. These open standards take the form of specifications. In the context of this paper, the "runtime specification" is of particular importance. Figure 2 highlights

[1]https://opencontainers.org/
[2]https://linuxfoundation.org/

the complete container virtualization toolchain from user input to running the actual *(v)* container. Mavridis and Karatza [12] describe this structure and the accompanying technologies in detail.

A *(iv)* runtime that implements the *(iii)* OCI runtime specification can be utilized by *(ii)* container engines that provide a respective interface. Popular technologies that implement this interface include containerd[3], Podman[4] and CRI-O[5]. The engines offer an Application Programming Interface (API) that can be used by *(i)* user-experience-oriented container management and orchestration tools such as Docker[6] or Kubernetes[7].

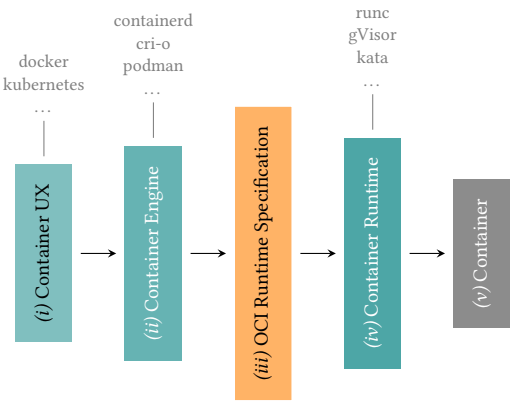


Figure 2: OCI ecosystem

As Figure 2 suggests, there are many combinations of tools within this chain possible, as each segment is individually interchangeable. Specifically, the widely adopted Kubernetes orchestrator created a large amount of the so-called "Kubernetes distributions" that bundle toolchains in an opinionated use case-driven manner. Popular examples are SUSE's[8] k3s[9] and RedHat's[10] OpenShift[11].

Another popular example for OCI compliant technology combinations is the container UX solution *Docker*. While docker initially started out as a full stack container solution including engine and runtime, it open sourced its components (containerd, runc) and now focusses on its role as management and orchestration tool.

2.3 Isolation Terminology

Isolation is a state that occurs when two workloads share and thus compete for a resource. The degree to which they influence each other describes isolation. If they have a strong impact on each other, the isolation is low and vice versa. [9, 11, 24]. This work follows the isolation definition of Krebs et al. [9] who define performance isolation as follows:

Definition 1 (Isolation). *Performance isolation is the ability of a system to ensure that tenants working within their assigned quota (i.e., abiding tenants) will not suffer performance degradation due to other tenants exceeding their quotas (i.e., disruptive tenants).*

In a similar context and especially in cloud computing, related work regularly uses the term "noisy neighbor". This noisy neighbor describes a disruptive tenant that negatively impacts another tenant. According to Longbottom [10], it is defined as follows:

Definition 2 (Noisy Neighbor). *A workload within a shared environment is utilizing one or more resources in a way that it impacts other workloads operating around it.*

2.4 Isolation Quantification

Throughout this work, we assume two distinct workloads W_a and W_d. The workloads themselves enact a certain amount of resource utilization.

W_a describes the abiding, behaving workload that stays within its assigned limits and utilizes a constant amount of resources. W_d on the other hand, defines a disruptive workload that misbehaves in one way or another. It may do so by actively trying to disturb other workloads as a "Noisy Neighbor" or by inadvertently negatively impacting other workloads due to an error.

A simple and natural approach to the quantification of isolation is the calculation of a "performance loss rate" [8, 11, 17, 24]. It describes the amount of performance degradation of W_a when affected by W_d on a fixed amount of workload.

Therefore, the baseline performance of a workload W_{a_1} in an uncontended environment is measured. Subsequently, the same workload plus an additional disrupting workload W_d is started, resulting in workload performance W_{a_2}. Both workloads compete against resources.

The isolation performance loss rate I_{plr} as the rate between the difference of both performance measurements can then be determined as shown in Equation (1). Slightly changing the perspective, I_{ulr} refers in Equation (2) to the utilization loss rate relative to the maximum possible utilization $R_{n_{max}}$ that a resource R_n can achieve.

$$I_{plr} = \frac{|W_{a_1} - W_{a_2}|}{W_{a_1}} \quad (1) \qquad I_{ulr} = \frac{|W_{a_1} - W_{a_2}|}{R_{n_{max}}} \quad (2)$$

In addition to this simplified model, further distinctions can be made. As part of his dissertation, Krebs et al. developed a model that included a graphical representation of isolation characteristics. It incorporates several interesting isolation points of interest along the range of values [9]. On the basis of those, additional metrics can be derived. Their graphical representation is adopted here and is presented in Figure 3.

In the following, we briefly iterate over their most notable metrics in the context of this paper.

Generally, this model also assumes static workload W_a. In contrast to the performance loss rate as described above, the disruptive workload W_d increases its load over time and as a consequence impacts W_a

The x-axis W_d in Figure 3 represents the amount of workload the disruptive tenant causes, whereas the y-axis W_a represents that for the abiding tenant.

[3]https://containerd.io/
[4]https://podman.io
[5]https://cri-o.io/
[6]https://docker.com/
[7]https://kubernetes.io/
[8]https://www.suse.com/
[9]https://k3s.io/
[10]https://www.redhat.com/
[11]https://www.openshift.com/

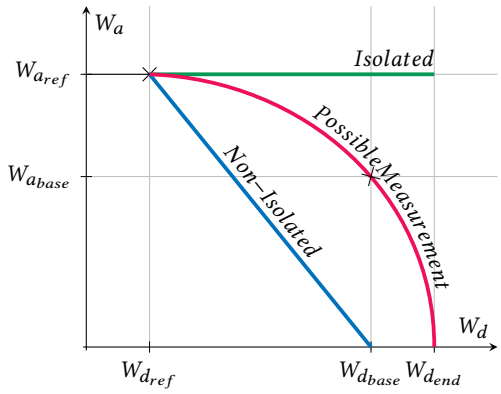


Figure 3: Visualization of theoretical and practical behaviour in isolation scenarios

The green line denoted with "isolated" shows a perfectly isolated W_a, which is not affected by W_d at all. No amount of W_d has an impact on W_a.

In contrast, the blue line indicated with "non-isolated", shows how workload W_a decreases, while workload W_d increases. For this case, no isolation occurs at all and W_d is clearly prioritized.

In reality, the actual graphical representation is represented by the red line denoted by "Possible Measurement". It will lie somewhere between the aforementioned green and blue lines.

With the graphical representation in Figure 3 in mind, various interesting isolation points can be identified. Moreover, these offer the potential to derive additional useful metrics. The following briefly iterates on the said points as defined by Krebs et al. [9].

Reference points. The reference point $W_{d_{ref}}$ marks where the disruptive tenant starts to degrade the abiding tenant. In consequence $W_{a_{ref}}$ defines this point from the perspective of W_a.

Degradation points. $W_{d_{base}}$ highlights the point where W_a is fully degraded and thus reduced to zero if W_a was not isolated. The same is true for $W_{d_{end}}$, marking the point where an arbitrary isolation would cause that degradation to zero. Finally, the cross section of $W_{a_{base}}$ and $W_{d_{base}}$ marks the respective workload for that arbitrary isolation that would otherwise be zero if no isolation had taken place.

Krebs et al. give an example for an isolation metric based on degradation points. They describe the difference between $W_{d_{end}}$ and $W_{d_{base}}$ in relation to $W_{a_{ref}}$. This relation is shown as I_{end} in Equation (3). Its value is zero if no isolation happens, and the higher this value gets, the better the isolation is. As this value tends to ∞, the authors suggest to rather use $W_{a_{ref}}$ as a reference resulting in values between $[0, 1]$. This is in Equation (4).

$$I_{end} = \frac{W_{d_{end}} - W_{d_{base}}}{W_{a_{ref}}} \quad (3) \qquad I_{base} = \frac{W_{a_{base}}}{W_{a_{ref}}} \quad (4)$$

The authors further argue that these metrics are sufficient only for systems that degrade linearly. They therefore propose two additional integral-based metrics. Equation (5) describes the relation between the area under the measured curve and the area under the curve of the non-isolated workload starting from $W_{d_{ref}}$. Equation (6) describes the same relation but starting from $W_{d_{base}}$ to an arbitrary point beyond $W_{d_{base}}$. The latter could be the highest value on $W_{d_{base}}$. Both values range between $[0, 1]$.

$$I_{intBase} = \frac{\left(\int_{W_{d_{ref}}}^{W_{d_{base}}} f_m(W_d)dW_d\right) - W_{a_{ref}}^2/2}{W_{a_{ref}}^2/2} \quad (5)$$

$$I_{intFree} = \frac{\left(\int_{W_{d_{ref}}}^{p_{end}} f_m(W_d)dW_d\right) - W_{a_{ref}}^2/2}{W_{a_{ref}}^2 \cdot \left(p_{end} - W_{d_{ref}}\right) - W_{a_{ref}}^2/2} \quad (6)$$

Not every metric described here will ultimately be useful for this work. Although we will elaborate on their applicability in section 4.

3 METHOD

This section presents the underlying method applied to gather isolation metrics. It starts by briefly summarizing this work's goal and follows by discussing the applied scenarios, instrumentation, and isolation quantification methods.

3.1 Goals

To reiterate our research questions, our aim is to measure the isolation of certain OCI based virtualization technologies. Therefore, we isolate them against each other in specific scenarios that are presented in the upcoming section. To achieve this, we instrument them directly on the host as in outside their virtualization environment. This enables us to collect high-resolution performance metrics of any involved process. This instrumentation is possible by leveraging eBPF. The actual isolation quantification follows related work. We decide to determine different metrics in order to compare and discuss them in conclusion.

3.2 Scenarios

We analyze the characteristics of isolation among tenants in four distinct scenarios. Each scenario consists of an abiding and a disruptive workload according to the model presented by Krebs et al. [9]. The abiding workload statically utilizes a resource and is contended by a disruptive workload. This disruptive workload continuously increases its workload until it reaches its final utilization. An overview of all scenarios is presented in Table 1. In this table, l_a and l_d describe the resource limits for w_a and w_d compared to the total available resources. w_a and w_d describe the relative utilization of the respective workloads within these limits.

For each scenario, the abiding workload utilizes a static amount of resources, whereas the disruptive workload increases linearly over time. This enables us to measure every possible combination of workloads between a static abiding and a linear disruptive workload. Plotting both on distinct axes results in a graph similar to Figure 3.

The runtime of each scenario is sensibly chosen depending on the maximum degree of utilization, its maximum capacity, and the time it takes to utilize it. The linear increasing disruptive workload

Table 1: Isolation scenarios with workload and their limits

name	w_a	l_a	w_d	l_d
harmony	100 %	50 %	100 %	50 %
escape	100 %	50 %	150 %	50 %
overcommitting	100 %	50 %	100 %	75 %
steal	50 %	50 %	100 %	75 %

evenly utilizes its resource over the experiment runtime and thus subdivides into fitting steps whose size and degree of isolation depend on the same characteristics as the total runtime. The experiments themselves are each repeated at least 10 times and the physical server involved is fully reset in between to mitigate any residue and thus impact from previous experiment runs [18].

harmony. In this scenario, two workloads fully utilize their assigned resources. The assigned resources are imposed through limits and evenly distributed throughout the resource as a whole, resulting in 50% for both. In practice, this means that the capacity planning performed previously adhered strictly to the combined available resources. This is a typical use-case for scenarios where no overcommitting or dynamic resource sharing happens. From a theoretical point of view, no resource contention should occur, and thus both workloads should not interfere with each other.

escape. This scenario is similar to the harmony one with only one exception. Here, the disruptive workload tries to escape its own imposed limit. Such scenarios occur when a workload accidentally or on purpose tries to exceed its allocated resources. In consequence, we can see two different things. One being whether the limit can actually be imposed and the workload is not able to exceed its limit, and the other being whether it is able to have an impact on the abiding workload in either case. In an ideal case, there should be no impact whatsoever.

overcommitting. Overcommitting is something that typically happens in cloud scenarios. As briefly mentioned in section 1 this may be part of their business model. In this scenario, we set a higher total limit for a resource than is physically available. To consider the worst case, both workloads use 100% of resources within their own limit. Again, considering the cloud computing use case, this most likely leads to a violation of customers Service Level Agreement (SLA). As both workloads stay within their limits, it is not deterministic how the workloads will behave. Nevertheless, it is still interesting how degradation occurs and whether different virtualization technologies behave differently.

steal. This is an extension of the "overcommitting" scenario. Although the limits still exceed the total available resource, workloads no longer use them fully. The abiding workload purposefully utilizes only half of its granted resources, while the disruptive still fully uses the resource until its limit. The combined workloads fully utilize all available resources, however. Essentially, this scenario tests whether the disruptive workload can steal free available resources from the abiding one. Some virtualization technologies may allow this, whereas others may strictly assign resources and block them.

3.3 Instrumentation

The method for resource instrumentation is based on the principles designed by [21]. In summary, this implies that the instrumentation must follow two essential functions. It *(i)* must occur outside of the virtualization technology to get a holistic view of the unobstructed resources. Furthermore, *(ii)* needs to be independent of virtualization technology to allow a fair comparison. This means that the instrumentation points need to be reasonably similar. Therefore, the uprobes as outlined in section 2.1 are not applicable.

eBPF has made it possible to access any kind of Linux instrumentation while promising low instrumentation overhead. The overhead can be kept low, as it can be performed within the kernel, reducing the amount of overhead induced by frequent kernel userspace interactions significantly.

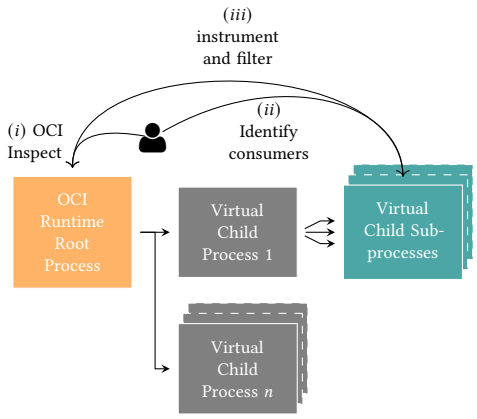


Figure 4: Instrumentation of processes controlled by OCI runtimes

As the instrumentation happens outside the virtualization technology, the following issues need to be solved. The process to do is highlighted in Figure 4.

We first need to *(i)* identify and distinguish virtualization technology instances that compete for resources. We further need to *(ii)* identify the actual process that consumes the resources. Finally, we need to *(iii)* find an appropriate instrumentation point that correctly profiles the resource in question.

When it comes to network resource profiling, the eXpress Data Path (XDP) feature of eBPF offers an efficient way of instrumenting. XDP provides an API to implement functions that are attached directly to network interfaces and allow stateless processing of incoming packets. It promises fast networking functionality, as it allows bypassing the Linux netfilter stack. State handling, including, but not limited to, packet counting and stateful connection tracking, can be achieved by leveraging eBPF maps that can be accessed by the named XDP function, as well as by user-space applications.

3.4 Isolation Quantification

In section 2.4 we presented and briefly discussed several useful metrics to quantify isolation. However, not all are desirable or even applicable to our scenarios as shown in section 3.2.

For example, I_{plr} in Equation (1) is widely applied in scientific work, but that may also be due to it being straightforward and comparatively simple. For these reasons and to compare it with another metric, we still calculate it. In order to do so, we need to pick a performance degradation point in an isolation diagram like Figure 3. Here, we choose the performance degradation where the disruptive workload is at its maximum. Another viable option would be the highest degradation of the abiding workload observed. However, we did not observe a significant difference between these possible points and therefore neglected them.

In section 2.4 we cite Krebs et al. [9] who argues that metrics like I_{plr} are only sufficient for linearly degrading resources. However, this is not always the case, as we can see in our results in section 6. They therefore suggested integral-based ones. However, some metrics assume that we have a disruptive workload that is capable of fully degrading the abiding one. Our scenarios do not force that, and depending on the resource, the respective resource scheduler might not allow this. As a consequence, we decide to use the $I_{int_{free}}$ metric in Equation (6). This metrics calculation only regards the first point of degradation and the highest applied disruptive workload.

In section 7 we reiterate the correlations for these metrics and discuss their respective applicability retrospectively. We therefore introduce a simple isolation similarity metric S_I between $I_{int_{free}}$ and I_{plr} that describes how similar they are to each other on a scale from $[0, 1]$ where 1 is exactly the same and 0 very different.

4 IMPLEMENTATION

The evaluation process is based on the evaluation framework presented in [21]. Some aspects are extended to enable answering the research questions. In particular, this involves updates to the load generation and instrumentation details.

Compared to the initial work, the three notable changes are *(i)* new scenarios as presented in section 3.2, *(ii)* new isolation metrics as described in section 3.4 and new technologies as shown in section 5.2.

4.1 Workflow

The whole process of load generation and data acquisition for every possible combination of scenario and technology relies on the evaluation framework. The abstract process of a conducted experiment is briefly highlighted below.

It follows the workflow highlighted in Figure 5. Here, the process begins with the *(i)* spawning of a virtualization technology process. Within this *(ii)* load is generated by the respective load generation tools. Afterwards, the *(iii)* profiling process on the host system is started in parallel. This profiling supervises and profiles the virtualization technology process. Upon success, data is *(iv)* acquired and *(v)* stored on external storage.

4.2 Load generation

The original load generation process needs several changes in order to enact the previously mentioned scenarios. Specifically, the static load generation at several distinct interesting points is changed to a linear load generation phase. This improvement still includes all previous configurations and extends them with a configurable number of points.

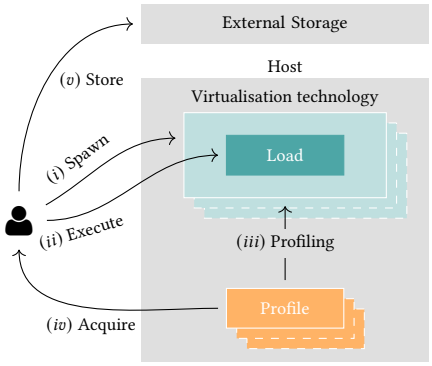


Figure 5: Flow of an abstract measurement

Benchmarking and load generation tools that are scientifically trusted do not commonly offer the possibility to gradually increase the load over given amount of. We therefore separate the load generation into multiple intervals with configurable resolution. By doing so, we can achieve a nearly linear behavior of load generation, as highlighted in Figure 6.

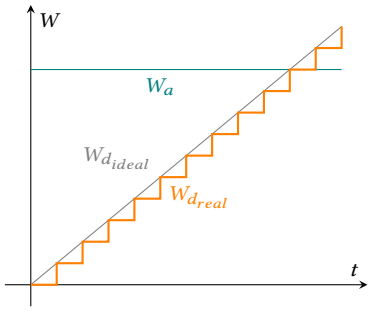


Figure 6: Visualization of linear load generation highlighting ideal and real W_d

Here, the axis describes the workload over time showing the abiding constant workload W_a as well as the theoretically ideal disruptive workload $W_{d_{ideal}}$ and the actual real disruptive workload $W_{d_{real}}$.

4.3 Instrumentation

In practice, the process structures of distinct virtualization technologies are very different. Adding a new technology to this workflow engine would involve a small implementation effort based on process structure investigations. To give an example, the process structure of loosely isolated container technology like docker with runc is visible in detail, whereas it is mostly abstracted for hypervisor-based virtualization like KVM. In the latter case, we cannot see processes running within the virtual machine from the outside.

The following Table 2 presents all the instrumentation points for the technologies observed in this work and the process filter necessary to sort the process trees of competing tenants. We specifically avoided kprobes, as they are significantly less stable than

instrumentation	resource	description
hardware:cycles	CPU	Hardware counter reporting CPU cycles
tracepoint:kmem: rss_stat	Memory	Tracepoint called when Resident Set Size (RSS) counters change
tracepoint:block: block_io_start	Disk	Tracepoint called when block operation request is queued for execution
XDP	Network	Network interface specific functions executed upon incoming network packet

Table 2: Instrumentation points list

name	version	note
Fedora CoreOS	39	Operating system version
Linux Kernel	6.5.6	Kernel used by the operating system Fedora CoreOS
k3s	v1.28.4	Rancher Kubernetes Distribution
Argo Workflow	v3.5.0	The workflow engine to orchestrate experiments and scenarios
bpftrace	v0.19.0	Profiling tool based on eBPF and bcc
stress-ng	0.13.05	Load generator for CPU and memory
fio	3.28	Load generator for disk I/O
iperf3	3.10.1	Load generator for network I/O
podman	4.7.0	
gvisor	20231023	with the systap platform
kata	3.2.0	with the virtio-fs storage driver

Table 3: Software version list

tracepoints. This is no hard requirement, though, and extensions of this framework might make their usage necessary.

To measure the throughput on a network interface, we implement an XDP function that increments time-based buckets in an eBPF-map. After a finished run, all buckets are then extracted by a user-space application for further analysis. The OCI specification dictates the use of veth-pairs to be OCI compliant. This allows us to perform black-box measurements of network isolation capabilities, independent of the OCI runtime under test.

5 EXPERIMENTAL SET-UP

5.1 Physical nodes

The experimental setup consists of 7 physical servers. They are arranged symmetrically and consist of identical components. The CPUs are two Intel CPUs of the model "Intel(R) Xeon(R) CPU E5-2630 v3" with a basic clock frequency of 2.40 GHz and a maximum clock frequency of 3.20 GHz. The memory attached to those CPUs have a total of $16 \cdot 16 = 256$ GiB DDR4 memory clocked at 2133 MHz available. The disk involved at the Input Output Operations Per Second (IOPS) isolation tests is a Samsung SM843TN, rated with 15000 IOPS "random write" performance. The server types involved are six experiment nodes for parallel execution and one control node that provides bare metal provisioning for the workflow control engine.

The networking between all involved nodes is realized by Mellanox Technologies Network Interface Card (NIC) of the "MT27800 ConnectX-"5 family. These are capable of a network throughput rate of $50 Gbit/s$. Nevertheless, as described during the results in section 6.4 they are not used for the actual network resource experiments. Here we use on-board network cards that only provide a maximum throughput of $1 Gbit/s$, as otherwise there would be no resource contention for hypervisors that do not offer bandwidths beyond a few $Gbit/s$

Notable other software components involved are listed in Table 3. All of these are part of the automated experiment workflow engine as described in section 4.1.

5.2 Selected Virtualization Technologies

Based on the components deconstructed of virtualization technologies, we can roughly separate them into categories [21]. Although there are not necessarily strict categories for their isolation capabilities, they are sufficient for a rough starting point. Thus, for this work we select three popular engines including OCI compatible container runtimes that each fit into those proposed categories. More specifically, the technologies are Podman[12], gVisor[13] and Kata[14]. Figure 7 shows a hierarchical representation of those choices.

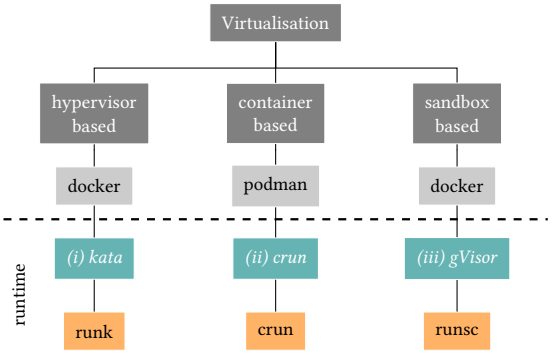


Figure 7: Virtualisation Classification Overview

Naturally, these technologies offer many possible configurations. In this paper, we use their most recent releases as of date, as well as the upstream default configuration. Table 3 gives an overview over these details in the bottom part. Here, we distinguish between the container runtime as defined in Figure 2 and the container runtime's binary name for reference.

crun. This container runtime, developed by RedHat, adheres to a more conventional approach. It leverages namespaces and cgroups for isolation, offering an alternative to runc, often bundled with containerd. Notably, it does not differentiate between the runtime and binary name, hence also referred to as "crun".

[12] https://podman.io/
[13] https://gvisor.dev/
[14] https://katacontainers.io/

gVisor. Originated at Google[15] gVisor promises a stricter isolation among workloads. Although this technology also builds on namespaces and cgroups, they further improve isolation by filtering Linux system calls [26]. This "sandbox" approach reimplements fundamental Linux capabilities within the user space to gain more control and thus improve isolation [22]. The runtime "runsc" is bundled with gVisor itself.

Kata. This independent OpenSource engine leverages hypervisor based virtualization to achieve isolation. Moreover, it also uses cgroups and namespaces where applicable. Similarly to the technologies mentioned above, Kata also bundles its container runtime "runk".

6 RESULTS

This section presents and discusses the results of the isolation measurements. Therefore, we present an overview in Table 4 that contains all isolation metrics determined based on the measurements we performed.

Table 4 consists of two multi-indexes. The vertical ones describe all permutations of virtualization technologies as described in section 5.2 and the scenarios we presented in section 3.2. Horizontal indices describe all permutations of resources that are instrumented as part of the workflow in section 4.1 and all isolation metrics discussed in section 3.4.

The number of results is too numerous to discuss every isolation characteristic in detail. As a consequence, we select interesting aspects for every resource and discuss them in the following sections. One thing all technologies have in common is that they allow granting unused resources to other workloads. In every steal scenario, the disruptive workload is able to allocate resources that could have been exclusively granted to the abiding one, without negatively impacting it. For this reason, we do not discuss this scenario further in the detail sections below.

The figures "Isolation" and "Timeline" presented throughout the remainder of this section are aligned to Figure 3 and Figure 6, respectively. The isolation figures furthermore highlight the changepoint discussed in section 3.4 with an orange circle and highlight the area under the abiding but disturbed workload. This tries to give a better idea of what the $I_{int_{free}}$ metric will look like, since this is part of Equation (6).

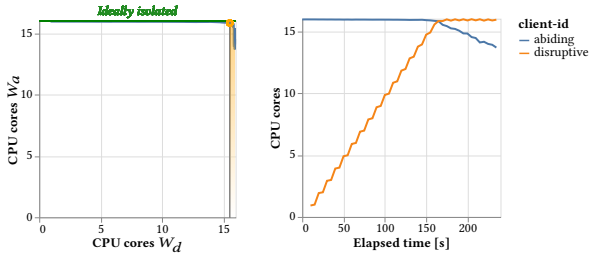
6.1 CPU

Reviewing the isolation metrics in the CPU column of Table 4, it is evident that CPU isolation is comparatively good and stable across the different virtualization technologies.

Considering a typical scenario like the escape one for the gVisor CPU, we can see that there is a slight degradation visible. However, S_I indicates that our two isolation models slightly disagree. To understand the reasoning behind this, we investigate the time and isolation charts.

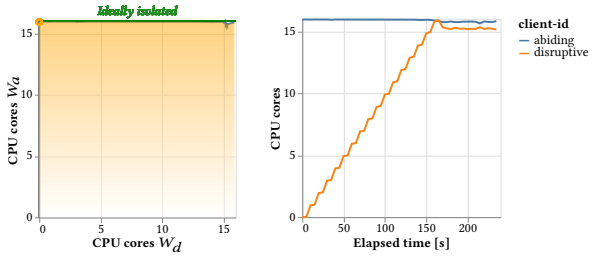
According to the escape scenario, both workloads fully utilize the CPU together. In fig. 8b we can see the steadily increasing workload W of the disruptive workload until it tries to escape its limit. We can see that this is not possible. However, as soon as the disruptive

[15] https://www.google.com/



(a) Isolation (b) Timeline

Figure 8: gVisor CPU Escape Scenario



(a) Isolation (b) Timeline

Figure 9: Kata CPU Escape Scenario

workload reaches its limit, the abiding workload starts to degrade. This might be due to the fact that the CPU scheduler needs to take efforts to keep the disruptive workload from exceeding its limit, implying less available CPU time for the abiding workload.

In contrast, the same scenario for the Kata CPU isolation looks slightly different. In fig. 9b we cannot observe a significant degradation over the course of the experiment. The fact that there is no change point above our threshold detected in fig. 9a, has a huge effect on the areas under the graphs of the calculated $I_{int_{free}}$. This can be neglected, though, as these cut each other out by building a ratio as highlighted in Equation (6).

6.2 Disk

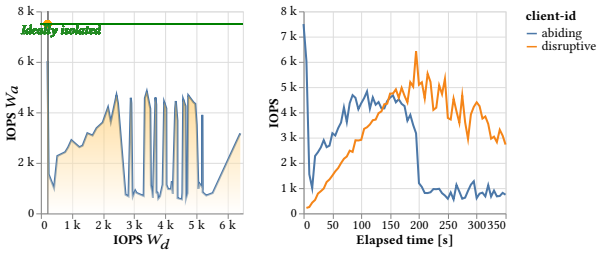
As visible in Table 4 it is imminent that disk isolation has problems for our scenarios. An immediately visible aspect is the fact that disk isolation for gVisor is not present. This is because it, at the time of writing, does not support directly passing block devices into its container. This is a mandatory requirement as we generate and measure direct block operations.

Apart from that observation, the disk isolation for the remaining technologies is arguably bad. Considering Podman, for example, yields bad results in almost every scenario with $I_{int_{free}} < 0.5$ and I_{plr} even worse.

This is clearly reflected in the respective time and isolation charts in Figure 10, taking the overcommitting scenario as an example. Here we can see that as soon as the disruptive workload starts, the abiding is almost reduced to zero. Interestingly, it starts to regain its workload over time, which will quickly be impacted by the ever-increasing disruptive workload, though. This behavior is a good example of how changing or specifically increasing the workload

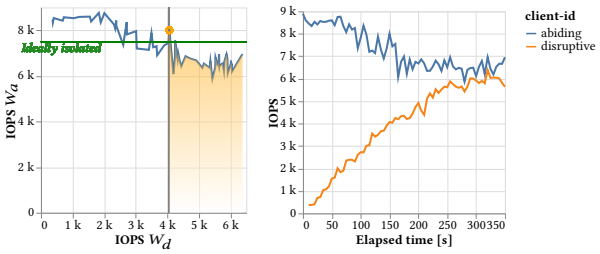
Table 4: Isolation metrics comparison

Technology	Resource	cpu			disk			memory			network		
	Metric	$I_{intFree}$	I_{plr}	S_I	$I_{intFree}$	I_{plr}	S_I	$I_{intFree}$	I_{plr}	S_I	$I_{intFree}$	I_{plr}	S_I
	Scenario												
gvisor	escape	0.89	0.85	0.96	n/a	n/a	n/a	0.99	0.99	1.00	n/a	n/a	n/a
	harmony	0.99	0.99	1.00	n/a	n/a	n/a	0.99	0.99	1.00	1.00	1.00	1.00
	overcommit	0.94	0.89	0.95	n/a	n/a	n/a	n/a	n/a	n/a	0.76	0.87	0.87
	steal	1.00	0.99	1.00	n/a	n/a	n/a	1.03	1.00	0.97	0.95	1.00	0.95
kata	escape	0.95	0.99	0.96	0.86	0.95	0.90	0.99	1.02	0.97	n/a	n/a	n/a
	harmony	0.99	0.99	1.00	0.60	0.93	0.64	1.00	1.02	0.99	1.00	1.00	1.00
	overcommit	0.88	0.80	0.92	0.44	1.04	0.43	n/a	n/a	n/a	0.96	0.94	0.98
	steal	1.00	1.00	1.00	1.42	1.31	0.92	1.10	1.16	0.94	1.00	0.99	0.99
podman	escape	0.94	0.84	0.89	0.41	0.13	0.32	0.93	0.91	0.97	n/a	n/a	n/a
	harmony	1.00	0.99	1.00	0.47	0.15	0.32	0.93	0.92	0.98	1.00	1.00	1.00
	overcommit	0.94	0.90	0.96	0.39	0.10	0.25	n/a	n/a	n/a	0.93	0.86	0.93
	steal	1.00	1.00	1.00	0.41	0.95	0.43	1.06	1.00	0.95	1.00	1.00	1.00



(a) Isolation (b) Timeline

Figure 10: Podman disk overcommit Scenario



(a) Isolation (b) Timeline

Figure 11: Kata disk harmony Scenario

has a different impact than two static competing workloads. This also relates to the comparative low S_I scores. As mentioned before, the less linear the degradation process is, the less applicable the I_{plr} metric becomes. For this metric, we only consider the highest applied disruptive workload that is not stable, leading to a I_{plr} with very low expressiveness.

However, in general, we can see that the contended resource is not actually the observed and limited IOPS but a related resource that is saturated. This naturally depends on many factors such as the physical type of the disk (e.g. HDD, SSD, etc.), the bus it is attached to (e.g. PCIE, SATA, SAS) or hardware specific details like the installed disk controller. The analysis of what exactly happens here is beyond the scope of this work and is left for future investigations.

The Kata disk isolation issue is very different. Although it is capable of effectively limiting disk IOPS for processes running inside the virtual machine, this does not include the disk IOPS performed by actual virtualization technology. The kind of hypervisor Kata uses does not offer the possibility to instrument the individual processes inside it through eBPF. Instrumenting the hypervisor in consequence adds IOPS it executes for its own overhead. This behavior can be seen in Figure 11 for the harmony scenario.

Although this limits the significance of these specific results, we were still able to observe performance degradation between the abiding and disruptive workload. However, we only consider workload below the actual targeted utilization as described in the scenarios in Table 1, to ensure that an actual degradation of the workload inside occurs.

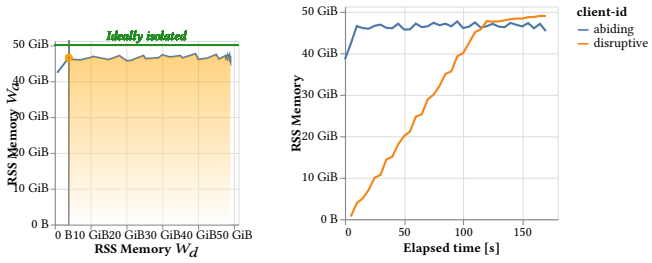
6.3 Memory

The isolation results for the memory metrics come with a small limitation. During the benchmarks, we determined that the Kata runtime has issues with the Non-Uniform Memory Access (NUMA) architecture of our physical servers and was not properly accessing memory across the CPU boundaries. Therefore, we limited virtualization technology access to a specific node and performed the experiments there while only allocating the memory (half of total) attached to it.

Moreover, we did not conduct any overcommit scenarios, as overcommitting memory was not possible in general. The Out Of Memory (OOM) killer would quickly kill the processes involved in our experiment, rendering the results useless. overcommitting the actual allocated memory (as in Resident Set Size (RSS)) is not possible in our configuration.

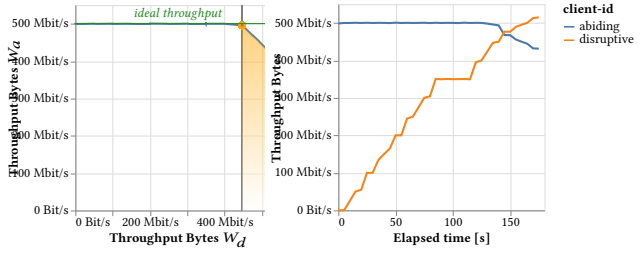
As memory isolation is very similar across technologies, we briefly discuss Podmans' escape scenario as an example in Figure 12.

One notable aspect is the observation that the actual workload never fully utilizes its designated utilization. This is due to the fact



(a) Isolation (b) Timeline

Figure 12: Podman memory Escape Scenario



(a) Isolation (b) Timeline

Figure 13: Podman network overcommit Scenario

that the load generator tries to allocate RSS memory with many workers in parallel and thus eventually reaches the limit. Once that happens, a process within the container gets OOM killed. One consideration here is that due to this circumstance, $I_{int_{free}}$ is lower than expected.

From the Figure 12 it is clearly visible that the imposed limitation for disruptive workloads works very well and cannot disturb the prevailing workload in any way.

6.4 Network

The virtualization technologies studied in this work do not offer the possibility of limiting the network bandwidth or IOPS of a NIC. This is the reason we do not perform an escape scenario for this resource, as it would merely mimic the overcommit scenario. However, technologies still possibly implement different strategies to enable network virtualization.

One significant difference is the Kata-induced limitation. Here, the maximum possible bandwidth achievable by the virtual NICs it creates is $5Gbit/s$. We therefore performed all our tests on an exclusive $1Gbit/s$ card specific for these experiments.

Apart from this consideration, the isolation metrics across all virtualization technologies are very similar. We see that as soon as both workloads fully utilize the NIC, performance degradation occurs. This behavior is clearly visible from any overcommit scenario as shown in Figure 13.

7 DISCUSSION

Although we can quantify the isolation capabilities of OCI runtimes by applying two different approaches to acquire isolation metrics, the results need critical reflection. This especially applies to the meaningfulness of the quantification methods in regard to the examined technology, scenario and resource, as well as the deviation of some results exceeding the predicted numeric range with an upper bound of 1.

As the respective S_I columns in Table 4 show, the deviation of the retrieved isolation metrics can be rather high across the technologies, applied scenarios, but also resources of interest. In section 3.4 we noted that I_{plr} is only sufficient for systems that degrade linearly. Looking at the corresponding graphs, it is evident that a linear degradation behavior does not apply to disk measurements. For some CPU measurements a near-linear behavior can be observed.

On the other hand, the integral-based metrics can also be misleading. As we calculate $I_{int_{free}}$ with the highest value of W_d, the applied scenario and therefore the maximum resource consumption of the disruptive tenant has a huge impact on the result of this metric.

In summary, depending on the selected configuration, the method to derive the isolation metric has to be selected carefully, the most expressiveness regarding the isolation capabilities is given by the isolation graphs.

Some calculated metrics show values higher than the theoretical upper bound of 1. For runtimes that hide information about running processes, such as gVisor, the retrieved values include the overhead produced by the runtime itself. A good example is the memory consumption measured for Kata containers. In each scenario, the retrieved values lie slightly above the imposed limits. Although Kata seems to apply the correct limits to the processes running inside, our method of measuring the consumption does not resemble the additional overhead of runtime. Subtracting overhead, to acquire more reasonable results, requires further investigation.

Generally speaking, the results that we present in this work are naturally very specific to the hardware, system configurations, and scenarios used. Isolation metrics cannot be easily compared between resources and scenarios. However, they can be compared within scenarios.

8 RELATED WORK

A common approach to quantify the isolation capacity of a virtualization technology is to determine the I_{plr} similar to Equation (1). Therefore, related work typically first measures a resource from within the virtualized environment[11, 17, 24, 25]. Combined with a subsequent measurement of the same workload under the influence of a disrupting contending workload, this ratio can be calculated. This approach has several constraints. One *(i)* is the measurement from within the virtualized environment. This neglects unforeseen impact on the host system. Another *(ii)* one is the dependability on the load generator specific to the stressed resource. Moreover, applying *(iii)* static workloads neglects effects induced by the variability of stress. Calculating only a *(iv)* single metric disregards the time in which a possible equilibrium of workloads is reached. In our work, we instrument the virtual environments from the hosting system through eBPF and thus decouple ourselves. Additionally, we induce a variable stress based on multiple scenarios and determine an isolation metric that takes the timeline into account.

This work focuses on the isolation of compute resources. Another aspect to consider is the impact that virtualization technology has on kernel resources and to classify isolation based on that. Wang et al. [23] follow very interesting approach to determine a misbehaving workload and further presents a tool to improve this isolation. Similarly Anjali et al. [1] assess virtualization technology categories along the kind and amount of system calls, and thus kernel stress.

9 CONCLUSION

Throughout this work, we presented a workflow engine and implementation details on isolation characteristics instrumentation for OCI compatible virtualization runtimes and give a rough estimate of the capabilities state-of-the-art virtualization technologies bring. Our results can be used in a decision process to pick a fit-for-purpose technology. Moreover, the framework itself can be extended for custom changes and executed on custom platforms.

We discovered some limitations in instrumentation and isolation for certain scenarios and technology combinations. These findings can be used to decide against a certain technology or to implement improvements.

One future direction that we intend to pursue with our work is to create a system of continuous profiling of isolation characteristics. Similarly to the work of Wang et al. [23], a classification of abiding and disruptive workloads could be based on a combination of performance metrics, derived isolation metrics, and kernel resource utilization. Furthermore, the characterization of workloads based on these metrics could be improved by integrating Quality of Service (QoS) metrics. Experiments may show a possible relation between resource isolation degradation and, for example, application response time.

REFERENCES

[1] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending Containers and Virtual Machines: A Study of Firecracker and gVisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/3381052.3381315

[2] Jörg Domaschka, Simon Volpert, Kevin Maier, Georg Eisenhart, and Daniel Seybold. 2023. Using eBPF for Database Workload Tracing: An Explorative Study. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. ACM, Coimbra Portugal, 311–317. https://doi.org/10.1145/3578245.3584313

[3] Jörg Domaschka, Simon Volpert, and Daniel Seybold. 2020. Hathi: An MCDM-based Approach to Capacity Planning for Cloud-hosted DBMS. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 143–154. https://doi.org/10.1109/UCC48980.2020.00033

[4] Srikar Dronamraju. [n. d.]. *Uprobe-tracer: Uprobe-based Event Tracing — The Linux Kernel documentation.* https://docs.kernel.org/trace/uprobetracer.html Accessed: 2023-02-27.

[5] Robert P. Goldberg. 1973. *Architectural Principles for Virtual Computer Systems.* Technical Report. HARVARD UNIV CAMBRIDGE MA DIV OF ENGINEERING AND APPLIED PHYSICS. https://apps.dtic.mil/sti/citations/AD0772809 Section: Technical Reports.

[6] Brendan Gregg. 2017. Linux eBPF Tracing Tools. https://www.brendangregg.com/ebpf.html.

[7] Masami Hiramatsu. [n. d.]. Kprobe-based Event Tracing — The Linux Kernel documentation. https://docs.kernel.org/trace/kprobetrace.html Accessed: 2023-02-27.

[8] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. 2020. *Systems Benchmarking: For Scientists and Engineers.* Springer International Publishing, Cham. https://doi.org/10.1007/978-3-030-41705-5

[9] Rouven Krebs, Christof Momm, and Samuel Kounev. 2012. Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA '12)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/2304696.2304713

[10] Clive Longbottom. 2017. *The Evolution of Cloud Computing: How to Plan for Change.* BCS Learning & Development Ltd, Swindon, UK.

[11] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. 2007. Quantifying the Performance Isolation Properties of Virtualization Systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science - ExpCS '07*. ACM Press, San Diego, California, 6–es. https://doi.org/10.1145/1281700.1281706

[12] Ilias Mavridis and Helen Karatza. 2023. Orchestrated Sandboxed Containers, Unikernels, and Virtual Machines for Isolation-Enhanced Multitenant Workloads and Serverless Computing in Cloud. *Concurrency and Computation: Practice and Experience* 35, 11 (2023), e6365. https://doi.org/10.1002/cpe.6365

[13] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. USENIX Association, USA, 2.

[14] Marek Moravcik, Martin Kontsek, Pavel Segec, and David Cymbalak. 2022. Kubernetes - Evolution of Virtualization. In *2022 20th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. 454–459. https://doi.org/10.1109/ICETA57911.2022.9974681

[15] David Schrammel, Samuel Weiser, Stefan Mangard, and Richard Sadek. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. (2022), 18.

[16] Martin Straesser, André Bauer, Robert Leppich, Nikolas Herbst, Kyle Chard, Ian Foster, and Samuel Kounev. 2023. *An Empirical Study of Container Image Configurations and Their Impact on Start Times.* https://doi.org/10.1109/CCGrid57682.2023.00019

[17] Xuehai Tang, Zhang Zhang, Min Wang, Yifang Wang, Qingqing Feng, and Jizhong Han. 2014. Performance Evaluation of Light-Weighted Virtualization for PaaS in Clouds. In *Algorithms and Architectures for Parallel Processing (Lecture Notes in Computer Science)*, Xian-he Sun, Wenyu Qu, Ivan Stojmenovic, Wanlei Zhou, Zhiyang Li, Hua Guo, Geyong Min, Tingting Yang, Yulei Wu, and Lei Liu (Eds.). Springer International Publishing, Cham, 415–428. https://doi.org/10.1007/978-3-319-11197-1_32

[18] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. 2008. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage* 4, 2 (May 2008), 1–56. https://doi.org/10.1145/1367829.1367831

[19] Theodore Ts'o. [n. d.]. *Event Tracing — The Linux Kernel documentation.* https://docs.kernel.org/trace/events.html Accessed: 2023-02-27.

[20] Simon Volpert. 2024. Workflows: An Empirical Analysis of Common OCI Runtimes' Performance Isolation Capabilities. https://doi.org/10.5281/zenodo.10612048

[21] Simon Volpert, Benjamin Erb, Georg Eisenhart, Daniel Seybold, Stefan Wesner, and Jörg Domaschka. 2023. A Methodology and Framework to Determine the Isolation Capabilities of Virtualisation Technologies. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. ACM, Coimbra Portugal, 149–160. https://doi.org/10.1145/3578244.3583728

[22] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2019. Practical and Effective Sandboxing for Linux Containers. (2019), 41. https://doi.org/10.1007/s10664-019-09737-2

[23] Kun Wang, Song Wu, Kun Suo, Yijie Liu, Hang Huang, Zhuo Huang, and Hai Jin. 2023. Characterizing and Optimizing Kernel Resource Isolation for Containers. *Future Generation Computer Systems* 141 (April 2023), 218–229. https://doi.org/10.1016/j.future.2022.11.018

[24] Xingyu Wang, Junzhao Du, and Hui Liu. 2022. Performance and Isolation Analysis of RunC, gVisor and Kata Containers Runtimes. *Cluster Computing* (Jan. 2022). https://doi.org/10.1007/s10586-021-03517-8

[25] Miguel G. Xavier, Israel C. De Oliveira, Fabio D. Rossi, Robson D. Dos Passos, Kassiano J. Matteussi, and Cesar A.F. De Rose. 2015. A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 253–260. https://doi.org/10.1109/PDP.2015.67

[26] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. (2019), 6.

An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory

Lukas Alt
RWTH Aachen University
Aachen, Germany
lukas.alt@rwth-aachen.de

Anara Kozhokanova
IT Center
RWTH Aachen University
Aachen, Germany
kozhokanova@itc.rwth-aachen.de

Thomas Ilsche
CIDS, Information Services and High Performance Computing (ZIH)
TU Dresden
Dresden, Germany
thomas.ilsche@tu-dresden.de

Christian Terboven
IT Center
RWTH Aachen University
Aachen, Germany
terboven@itc.rwth-aachen.de

Matthias S. Mueller
IT Center
RWTH Aachen University
Aachen, Germany
mueller@itc.rwth-aachen.de

ABSTRACT

Power consumption of the main memory in modern heterogeneous high-performance computing (HPC) constitutes a significant part of the total power consumption of a node. This motivates energy-efficient solutions targeting the memory domain as well. Practitioners need reliable energy measurement techniques for analyzing energy and power consumption of applications and performance optimizations. Running Average Power Limit (RAPL) is a common choice, as it provides uncomplicated access to the energy measurements. While RAPL's accuracy has been studied and validated on homogeneous memory platforms, no work we are aware of investigated its accuracy on heterogeneous memory platforms, specifically with high-capacity memory (HCM).

This paper describes the process of measuring the memory power consumption externally using riser cards in detail. We validate RAPL's accuracy by comparing results obtained from Intel's Ice Lake-SP system equipped with DDR4 DRAM and Intel Optane Persistent Memory Modules (PMM). In addition, we verify the accuracy of our instrumentation setup by comparing the results from an older Broadwell system with the results in the literature.

We show that the RAPL values on a heterogeneous memory system report a higher offset from the reference measurements. The difference is more pronounced at lower memory load for all memory types. Also, we find that RAPL readings are inconsistent between multiple sockets and over time. Based on the evaluated scenarios, we conclude that RAPL overestimates the actual power consumption on heterogeneous memory systems and provide a discussion on the possible causes of this effect.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645052

CCS CONCEPTS

• **Hardware → Power estimation and optimization**; *Memory and dense storage*; • **Computer systems organization → Heterogeneous (hybrid) systems**; Multicore architectures; • **General and reference → General conference proceedings**; **Validation**.

KEYWORDS

HPC, Heterogeneous Memory, Running Average Power Limit, Energy Efficiency, Intel Optane Persistent Memory

ACM Reference Format:
Lukas Alt, Anara Kozhokanova, Thomas Ilsche, Christian Terboven, and Matthias S. Mueller. 2024. An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645052

1 INTRODUCTION

Continuously increasing complexity and data intensity of modern computing applications promote the development of heterogeneous architectures even further. In addition to the variety of special-purpose compute cores, the latest systems now feature a new class of high-capacity memory (HCM) and a range of high-bandwidth memory (HBM) devices. HCM provides higher memory capacities, often with persistence features, and is slower than traditional DRAM. While HBM delivers higher bandwidth, it usually offers a lower capacity.

In the past, node-level power optimizations mainly focused on CPU and GPU power measurements, and the memory subsystem was often neglected due to its minor contribution to total power consumption. However, these days, in heterogeneous memory systems, particularly with HCM, the memory domain can consume a non-negligible amount of electric power[1]. With energy consumption being a limiting factor in HPC, especially in terms of costs, carbon footprint, and power availability, the increasing energy consumption motivates the investigation of energy-efficient solutions

[1]The memory power consumption during the STREAM benchmark execution on PMM takes up to 51% of CPU TDP on Intel Ice Lake and up to 72% on Cascade Lake [20].

for the memory domain as well. This requires an accurate and reliable method for collecting power consumption data from memory technologies at high temporal resolution.

A well-established proxy that provides energy measurements is the Intel Running Average Power Limit (RAPL) - a software interface for monitoring and limiting the power consumption on Intel systems. Over the last 10 years and different Intel microarchitecture generations, RAPL evolved and was found to be a reliable source for nonintrusive energy readings since the Haswell-EP architecture [1]. However, it is unknown whether RAPL is as reliant on the latest Intel server platforms such as Ice Lake or Sapphire Rapids equipped with heterogeneous memory.

In this paper, we validate RAPL on the latest Intel Ice Lake-SP platform with heterogeneous memory, namely DDR4 DRAM and Intel Optane Persistent Memory Modules (PMM). In particular, we describe in detail a methodology for measuring the power consumption of the memory domain by using two different DDR4 DRAM modules and 200-series PMM memory devices. The instrumentation setup is based on the DDR4 DIMM extenders by Adex Electronics that are compatible with both memory types. While this power measurement method has been practiced before [1, 5, 19], this work provides novel insight into RAPL's memory power measurement mechanisms on a heterogeneous memory platform. Additionally, we verify the accuracy of the reference measurements against a previous study utilizing a Broadwell-EP system. The results show that RAPL measurements for the memory domain differ from the actual power consumption by an offset that increases with lower memory load, both on DRAM and PMM modules.

The following are the main contributions of this paper:

(1) We present a comprehensive measurement setup to physically monitor the power consumption of DDR4 DIMMs.
(2) We evaluate the accuracy of RAPL on a modern heterogeneous memory system - Intel Ice Lake-SP equipped with Optane Persistent Memory and DRAM modules.
(3) We investigate the granularity of RAPL counters (update intervals) of the memory domain on Ice Lake-SP and Broadwell-EP platforms.
(4) We highlight RAPL's inconsistencies: temporal and inter-socket and provide a discussion on possible causes of these effects.

The rest of the paper is structured as follows: After introducing the extensive background of this work in Section 2, Section 3 discusses related work. In Section 4, the experimental setup by instrumenting DIMMs using riser cards is introduced. Section 5 contains the results of the experiments along with a discussion on the accuracy and temporal resolution of the RAPL memory domain. The key findings are concluded in Section 6.

2 BACKGROUND

Before presenting the methodology of this paper, in this chapter, we cover several necessary aspects related to the power consumption measurements in our setup. First, we explain the basic characteristics of the PMM memory. Then, a brief background of the RAPL is provided. Next, we describe common power measurement methods and present an overview of the power delivery scheme to the main memory.

2.1 Intel Optane Persistent Memory Modules

Intel Optane Persistent Memory Modules (PMM) were introduced by Intel as a novel non-volatile memory tier based on the 3D XPoint technology. Both PMMs, that ship in DIMM form factor, and Intel Optane SSDs are based on this technology. PMMs were first supported by systems starting Intel's 2nd generation Xeon Scalable processors (Cascade Lake architecture). Currently, three PMM generations - 100, 200, and 300 series - have been released, each compatible with the 2nd, 3rd, and 4th generations of Xeon Scalable processors, respectively.

PMMs operate in *Memory* and *AppDirect* modes. The *Memory* mode exposes the DRAM as a write-back cache and the PMMs as a directly addressable volatile HCM. In *AppDirect* mode, DRAM and PMM are both byte-addressable, with PMM being non-volatile in addition. PMMs can be configured as *interleaved* and *non-interleaved* in this mode. Moreover, PMM can be configured in a so-called *Hybrid*[2] mode where one part is allocated and configured in *Memory* mode and the other in *AppDirect* mode. Each memory type can be exposed as a separate NUMA node with Linux kernel v5.15 or later. In addition, `ipmctl`, `daxctl`, `ndctl` Linux packages are required for managing and configuring the PMMs.

Memory allocations on target memory types can be done in several ways. In cases when the source code cannot be modified, `numactl` [22] can be used to query NUMA settings and manage NUMA policies for memory allocations. When the source code is modifiable, `libnuma` [21], `libvmmalloc` [29], and `memkind` [30] libraries provide custom memory allocators and allow for a more granular memory allocation control.

Depending on the configuration modes, PMM exhibits distinct performance characteristics. Reading from the Optane PMM media is about three times faster than writing, however, it is still two to three times slower compared to DRAM [39]. Read latency is also impacted by whether the access is sequential or random whereas DRAM is almost insensitive to that. *Memory* mode is generally more performant than *AppDirect* mode [28]. Also, remote and highly concurrent accesses significantly impact the PMM performance. This is known to be caused by the contention at the internal PMM media buffers and at the integrated memory controller level [39].

2.2 RAPL

Running Average Power Limit (RAPL) is an interface for managing the energy consumption of a system and was first introduced by Intel for the Sandy Bridge architecture. RAPL is a well-known tool for measuring and modeling a system's energy consumption and has been widely used in numerous applications. It exposes a set of model-specific registers (MSR) containing information on energy counters and power limits, among others, for certain system component domains, also known as power planes. The availability of power planes varies depending on the architecture and between client and server platforms. On Intel servers, *Package* and *DRAM* power planes are mostly present. The high-level access to the RAPL interface is facilitated by the performance monitoring tools such as PAPI [25], LIKWID [36], and perf [23].

RAPL energy counters should be updated roughly every 1 ms [11, Vol. 3, Sec. 15.10.3-5], however, the span of the timing gaps varies on

[2]Note that this mode is supported only on 100-series PMM.

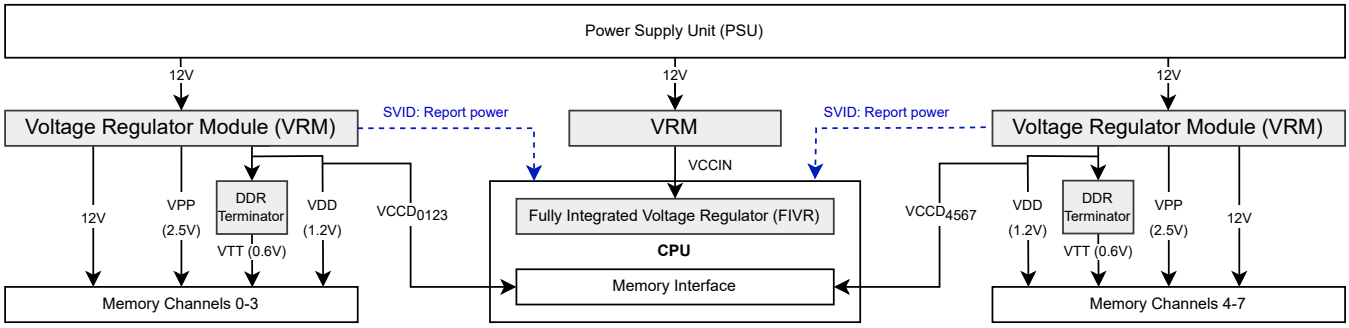


Figure 1: Power delivery to the memory of one socket on Intel Ice Lake-SP systems [2, 10, 13]

different architectures [18] as we also show in Section 5.1. In recent Intel architecture generations, RAPL's accuracy has been improved so that the current systems report RAPL values that are based on the actual measurements by default rather than estimations [1].

RAPL energy readouts were successfully used for side-channel attacks, e.g., monitoring the control flow of applications or accessing cryptographic keys, in previous work [24]. To mitigate this, Intel implemented an *energy filtering* mechanism, which can be explicitly enabled by setting an MSR or enabling Software Guard Extensions (SGX). If enabled, random noise is added to the energy readouts and the update interval is varied for the RAPL *Package* and *PP0* domains. Unfiltered energy measurements are still used for power management. The hereby introduced offset in energy measurements decreases with lower sampling rates [14]. All systems used in this paper have *energy filtering* and SGX disabled.

2.3 Direct Power Measurement Methods

Physical power measurement techniques can be classified into Hall-effect-based and shunt-based methods. Hall-effect sensors provide non-intrusive and contactless current sensing by outputting a voltage proportional to the magnetic field induced by the current through a conductor [1]. Alternatively, a current-sense resistor, usually with resistance in the order of milliohm and often called *shunt*, can be placed in the circuit. This shunt can be arranged either before or after the load in the circuit, which is called high-side and low-side current sensing, respectively [32]. Shunt-based methods are overall more intrusive than Hall-effect-based methods.

Employing these techniques for power measurements of components of an HPC node requires instrumenting the system with such power measurement sensors. Several measurement points and techniques have been utilized for this in the past. For example, some studies used wall-socket power measurement devices, e.g., the WattsUpPro [1], or power analyzers, such as the LMG450 [3], LMG670 [4, 33], and Plugwise [18], to measure the total power consumption of a node. On the other hand, specialized measurement techniques for measuring the power consumption of specific components or subsystems have been developed. Desrochers et al. [1] used a modified Molex connector cable, which connects the power supply unit (PSU) to the P4 ATX connector of the mainboard, with Hall-effect sensors to measure the power consumption of CPUs on desktop systems. The authors also utilized DIMM riser cards with integrated shunts for per-DIMM power measurements of the memory. Ilsche [4] instrumented the voltage regulators that power the memory and CPU for power measurements using the LMG670 power analyzer. In addition, the author used custom Molex connectors with integrated shunts and utilized them for measuring the power consumption of different DC consumers from the PSU.

2.4 DIMM Power Delivery

Figure 1 schematically visualizes how the AC power input on an Intel Ice Lake-SP platform is delivered to the memory DIMMs on the mainboard. The AC input is first converted into a fixed DC voltage, usually 12 V, by the power supply unit (PSU). This voltage is used to power a voltage regulator (VR), commonly implemented as a multiphase buck converter, that converts the DC input into a set of fixed voltages to power the DIMMs [2], i.e., VDD (1.2 V), VPP (2.5 V), and 12V. On the Ice Lake-SP platform, the memory modules of channels 0–3 and 4–7 of each socket are powered independently [10, 13]. According to the electrical specification [10], the mainboard provides two 1.2 V lines to the CPU as a "power supply for the processor system memory interface". While not specified explicitly, it can be expected that this supply voltage is also provided by the same voltage regulators that power the memory modules.

The CPU can communicate with the voltage regulator over the Serial Voltage Identification (SVID) interface, which is used for several purposes. Most importantly, it allows the CPU to digitally configure the output voltages of the different channels of the VR. Furthermore, the VR reports power consumption to the CPU, which forms the base for RAPL power measurements [2].

3 RELATED WORK

While most of the previous works focused on validating RAPL's power consumption readouts of the package or the total system domain, only a few studies validated RAPL's memory power measurements separately.

Desrochers et al. [1] describe the details of external power sensing approaches that utilize Hall-effect current sensors and DIMM extenders (riser cards) for measuring the power drawn by CPU and memory domains, respectively. They conduct experiments on various Intel Haswell client and server platforms equipped with DDR3 and DDR4 DRAM modules to collect and compare the power readings from RAPL and reference measurements. The authors use a single DIMM extender per each experiment. Their results show that on a system where RAPL reports actual power measurements

(Haswell-EP), the measured values match more closely or offset by 0.3 W compared to the RAPL values based on the estimation in desktop machines, where the error is higher. Memory capacity is also shown to be a factor: the larger capacity 16 GB memory module matches the measurements more closely within 5% than the 8 GB module with 20%. The most significant divergence is shown to be at idle state, which is also impacted by the capacity of a memory module, with more than 75% error for the 8 GB module vs. 38% error with the 16 GB module. The authors conclude that RAPL is generally accurate on Haswell platforms.

Hackenberg et al. [3] evaluate RAPL's total power consumption from Package and DRAM domains on Intel Sandy Bridge-EP and Haswell-EP systems. The authors use an LMG450 ZES power meter for reference measurements of the total system power consumption (AC). Their results show that RAPL reports a stronger correlation on Haswell-EP compared to the Sandy Bridge-EP system on a quadratic fit. Our work, in turn, provides a detailed look at the memory domain on a more recent architecture.

Ilsche et al. [5] compare different power measurement techniques by instrumenting socket components such as CPU, northbridge, and RAM on Intel Sandy Bridge-EP and AMD Opteron systems. The authors compare the results from the LMG450 power meter, Hall-effect, and shunt sensors, as well as PCIe and 1×DDR3 riser card, and provide a discussion on the trade-offs of each power measurement technique.

Ilsche [4] matches RAPL energy readouts at package and DRAM domains on a Sandy Bridge-EP and a Skylake-SP system with measurements of an LMG670 power meter. This work instruments the 12 V inputs of VRs on mainboards with separate power pins for memory and CPU sockets. The results for the package power domain are consistent with previous work, showing discrepancies on the older Sandy Bridge architecture and a good fit for the Skylake system. For DRAM power on Sandy Bridge, this work demonstrates a close correlation between RAPL and measured power for configurations with high power consumption. However, at several different low-power workloads, that vary in their physically measured value, RAPL reported a constant power consumption. In contrast, the RAPL DRAM values on the Skylake system were mostly consistent with the measurements at the respective 12 V input, exhibiting a maximum relative discrepancy of 7.1% under an assumed VR-efficiency, excluding idle. However, the author remarks that the remaining discrepancies are even smaller for the sum of package and DRAM, which may indicate a slight mismatch of the RAPL domains and the respective 12 V pins.

The experimental evaluation in our work shows that RAPL on a newer architecture reports higher divergence and variability for the memory subsystem than observed previously by [1]. Due to the scarcity and complexity of such work, we believe this paper will bring better clarity on the subject in addition to the prior work.

4 EXPERIMENTAL SETUP

The experiments were conducted on two different Ice Lake-SP and Broadwell-EP systems, as shown in Table 1. While the Ice Lake-SP system is a state-of-the-art heterogeneous memory system, the Broadwell-EP system is a homogeneous memory server. In particular, the Broadwell-EP microarchitecture is similar to the Haswell-EP architecture, for which previous work evaluated the temporal resolution [18] and the accuracy of the RAPL memory domain [1]. For the following experiments, the memory population of the systems from Table 1 was modified by physically exchanging the memory modules. We explicitly specify the exact memory modules from Table 2 for each experiment. The PMMs were configured in AppDirect mode throughout the experiments.

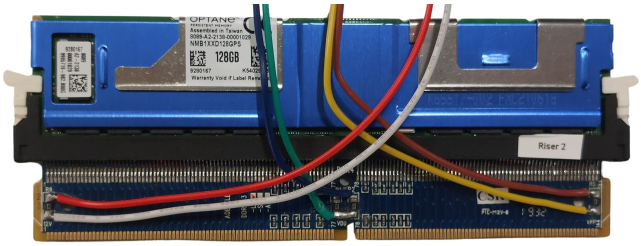


Figure 2: DDR4-compatible riser card with current-sense resistors at 12V, VDD, and VPP and probing wires. A PMM module is installed in the riser.

Intel architecture	**Ice Lake-SP**	**Broadwell-EP**
Mainboard	Intel M50CYP2SB Engineering Sample	Supermicro X10DRTH
No. sockets	2	2
CPU model	Xeon Gold 6338	Xeon E5-2650 v4
Cores	32	12
Base frequency	2.0 GHz	2.2 GHz
Main memory	16x32GB Samsung DDR4 3200 MT/s	8x16GB SK hynix DDR4 2400 MT/s
Other memory	16x128GB Intel® Optane™ PMM	-
Power governor	performance	
OS	Rocky Linux 8.8 (Green Obsidian)	
Kernel	6.4.3-1.el8.elrepo	

Table 1: Default hardware configuration of the Ice Lake-SP and Broadwell-EP system

Vendor	**Name**	**Capacity**	**Transfer Rate**	**Part Number**
Samsung	DDR4 SDRAM ECC RDIMM	32GB	3200 MT/s	M393A4K40DB3-CWE
SK hynix	DDR4 SDRAM RDIMM	16GB	2400 MT/s	HMA42GR7AFR4N
Intel	Optane™ PMem 200 Series	128GB	3200 MT/s	NMB1XXD128GPSU4

Table 2: Memory modules used for experiments

4.1 Hardware Instrumentation with DIMM Riser Cards

For the reference measurements, we utilized DDR4 compatible DIMM riser cards from Adex Electronics as shown in Figure 2. These riser cards are compatible with DDR4 registered DIMMs

and 200 series PMM and feature 5 mΩ, 1% tolerance current-sense resistors (also called shunts) with a power rating of 0.5 W at all power-delivering pins of the DIMM[3], which enables high-side current sensing.

To measure the power consumption at the different pins, we measured the voltage drop across these shunts. For DRAM, we measured the 1.2 V VDD and 2.5 V VPP pins, while the 12 V and 1.2 V VDD rails are measured for PMM. According to our measurements, the power consumption of VPP is less than 10 mW for PMM and can be neglected for our purposes. All other pins also only show a minor contribution to the power consumption of the DIMM and are neglected here [1].

According to Ohm's law, the voltage drop is proportional to the current, i.e., $I = \frac{\Delta U}{R}$ with ΔU being the voltage drop and $R = 5\,\text{m}\Omega$ the known resistance of the shunt. As this voltage drop is small and cannot directly be measured by most measurement devices, we amplify the voltage drop by the factor of 100 for VDD and 200 for 12V and VPP using *INA2180* [9] current-sense amplifiers [4]. These integrated circuits take two voltages as an input and output a single-ended voltage that corresponds to the difference between the inputs multiplied by a fixed gain on two independent channels. In our case, the input pins were connected to both sides of the shunts. For this work, evaluation boards for these amplifiers with preinstalled bypassing capacitors were utilized and powered by a 5 V supply voltage from a Raspberry Pi.

The amplified voltage drop and the bus voltage, probed on the high side of the resistor for each considered pin of a DIMM, are then measured using an MCC128 [26] data-acquisition board. This device was installed as a HAT on a Raspberry Pi and is capable of measuring the voltage on 8 single-ended input channels at an aggregated sampling rate of 100 kHz across all channels. We configured it to measure input voltages between 0 V and 5 V and stepped the 12 V bus voltage for PMM down to around 2.8 V using a custom voltage divider realized with an 815 Ω and a 2680 Ω resistor with 0.1 % tolerance. The sampling rate was 1 kHz for each channel[5].

4.2 RAPL Measurements

perf_event is a commonly used tool for accessing RAPL measurements, which was used by Desrochers et al. [1] in a previous evaluation of RAPL. As sampling RAPL counters with *perf* at high sampling rates introduces significant overhead and the start of the measurements takes too long, a more lightweight solution for conducting RAPL measurements was required. We adopted a C code snippet by Weaver [37] to directly read RAPL MSRs at a configurable sampling rate and write the results into a text file. Furthermore, we integrated support for the Intel Ice Lake-SP platform and added the option to supply an application that should be monitored as a child process.

While RAPL advertises a sampling interval of around 1 ms, our results revealed that the update interval can be subject to a significant jitter (see Section 5.1). Thus, we decided to sample RAPL counters only every 5 ms.

4.3 Measurement Synchronization

While the reference measurements are conducted by the Raspberry Pi, RAPL measurements have to be conducted by the system under test itself. This introduces the challenge of synchronizing the results between both devices.

Previous work employed different techniques for this. Desrochers et al. [1] utilized a serial interface cable for this purpose and modified the *perf* tool to output a high signal to one of the pins of the serial cable when measurements start and to set it back to low when measurements end. Ilsche [4] added correlation sequences to the power measurement signal and detected the pattern of this sequence in the resulting measurements to line up the measurements with the execution of the application. For this work, we utilized a *MCP2221* [27], which plugs into the system under test via USB and features a serial UART interface and several GPIO pins. One of these GPIO pins was connected to the digital trigger pin of the data-acquisition card. We configured it to start the reference measurements when the GPIO is set to high. The GPIO is connected to another GPIO of the Pi, which is used to stop the measurements when a falling edge is detected. While the measurement start is synchronized tightly, a higher delay can be expected for stopping measurements.

When the RAPL measurements are started on the system under test, the GPIO is set to high by our RAPL measurement tool. Once the monitored application exits, the reference measurements are stopped by setting the GPIO back to low.

The serial UART interface of the *MCP2221* is also connected to the serial interface pins of the Pi to establish a communication link between the system under test and the Pi. This interface is utilized to communicate benchmark information between both devices. In addition, the ground pin of the *MCP2221* is connected to the analog ground of the MCC128 to establish a common reference ground between the data-acquisition card and the system under test[6].

4.4 Data Post-processing

For each experiment, a CSV file with the raw voltages measured by the MCC128 is stored on the Pi. The voltage drop is divided by the gain and the shunt resistance, i.e., 2.5 mΩ for VDD and 5 mΩ for the rest, and multiplied by the measured bus voltage to compute the power, e.g.:

$$P_{VDD} = U_{VDD} \times \frac{\Delta U_{VDD}}{100 \times 2.5\,\text{m}\Omega}$$

The RAPL measurements produce a text file with the energy consumption since the last sample and the computed average power

[3] At the VDD pin, which is the primary 1.2 V power supply of DDR4 DRAM, two of these shunts are installed in parallel to double the maximum current rating. This is required for faster and larger-capacity DDR4 modules. The previous revision of the risers cards used by Desrochers et al. [1] only has a single shunt at the VDD, which limits maximum power drawn by the memory module.

[4] Initially, we used *INA122* instrumentation amplifiers on a breadboard like Desrochers et al. [1]. These amplifiers, however, feature insufficient bandwidth, which resulted in decreased amplification gain and an underestimation of the power consumption as a result. As the breadboard introduced significant contact resistance and instability, we soldered all connections or used screw terminals to maintain strong and low resistance connections.

[5] The MCC128 samples all configured channels within one scan clock cycle [26]. For DRAM at idle, energy measurements with the maximum supported sampling rate of 25 kHz per channel compared to 1 kHz only differed slightly (1.8575 W vs. 1.8431 W). If higher accuracy is required or if signs of strong aliasing occur, an increased sampling rate or low-pass filtering is preferred.

[6] Although both devices are inside the same building, a substantial difference between the ground level of both devices cannot be ruled out.

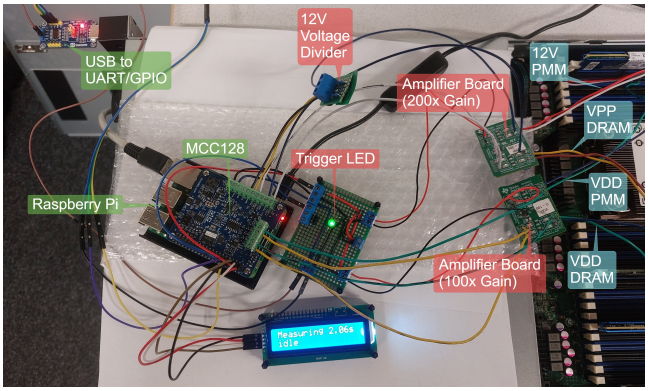


Figure 3: Hardware for amplification and processing of power measurements on the Ice Lake-SP system

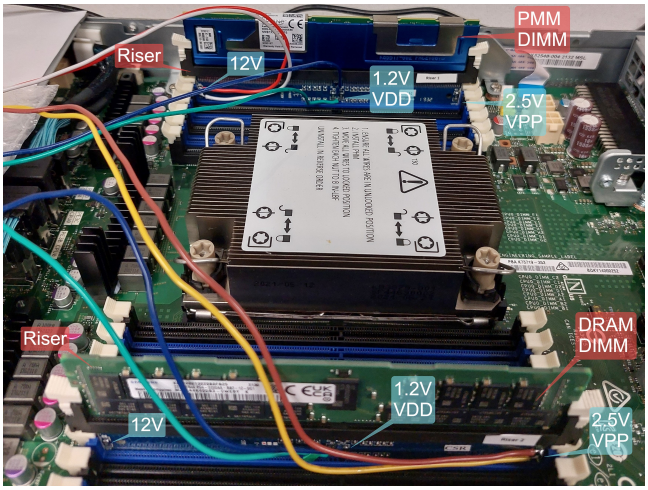


Figure 4: DRAM and PMM modules instrumented with riser cards on the Ice Lake-SP system

consumption during that interval for each RAPL domain on both sockets.

As both measurements are sampled at different rates, we had to resample them to a common sampling interval to directly compare RAPL with reference measurements. For this, the values for both measurements are averaged within 200 ms intervals[7].

4.5 Experimental Validation

The 0.5 W power rating of the preinstalled shunts at the riser cards limits the power consumption of the installed memory modules. For example, the power dissipation at the 12 V pin is $P = U \times I = I^2 \times R = I^2 \cdot 5\,\text{m}\Omega$. In this particular case, the maximum current rating is 10 A, which would correspond to a power consumption of 120 W, which is well above the maximum burst power consumption of 18 W for PMM.

[7]The artifacts archive containing our measurement data, the sourcecode, and further information on reproducibility of our experiments is available at https://doi.org/10.5281/zenodo.10783502.

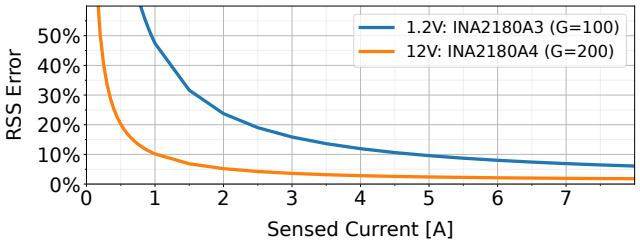


Figure 5: Theoretical residual sum of squares error of INA2180 in high-side current sensing at 25°C. V_s = 5.1 V, Shunt Tolerance: 1 %. Comparing V_{CM} = 12 V , R_{shunt} = 5 mΩ, Gain = 200 and V_{CM} = 1.2 V , R_{shunt} = 2.5 mΩ, Gain = 100

However, the VDD pin on registered DDR4 DIMMs has a maximum expected current of 11.7 A [16], which exceeds the 10 A rating of a single shunt. As two shunts are installed in parallel for VDD, the maximum current rating is doubled, so RDIMMs can be operated safely. PMM is primarily powered via 12 V and can be operated safely as well due to the higher voltage and lower current.

We also computed the maximum expected voltage drop at the shunts, which is highest for VDD with a value of 29 mV. If we assume a typical voltage of 1.21 V [16] at VDD, the voltage behind the shunt can drop to 1.181 V, which is still above the minimum voltage specification of 1.16 V for VDD [16].

Furthermore, we measured the memory access latency overhead introduced by the riser by running the Intel Memory Latency Checker (MLC) [12] with and without riser cards. The use of the risers and the open sever case did not introduce a statistically significant latency overhead, which is in line with previous results [1].

The accuracy of the reference measurements is mainly influenced by the shunt tolerances, the gain error at the current-sense amplifier, and the analog-digital conversion using the MCC128. Figure 5 shows the RSS error estimation for current sensing of VDD and 12V according to a calculator provided by Texas Instruments [8], which covers the first two error sources. The figure shows that the error strongly depends on the current, decreases with increasing current, and is significantly lower for 12 V compared to 1.2 V. In this case, the minimum expected current is 1.5 A for 1.2 V and 0.25 A for 12 V, which corresponds to an RSS error around 30 % for 1.2 V and 40 % for 12 V and that is mainly caused by a relatively high bias current, which is the current that flows through the input pins of the amplifier. Our practical experience, however, showed that this high error is more of a theoretical nature. For example, the 16GB DDR4 modules by SK hynix should consume 1.7 W during *active standby* according to its datasheet [34]. Our reference measurements report between 1.65 W and 1.85 W at idle and closely match the specification from the datasheet [8]. The analog-digital conversion introduces a maximum absolute error of 3 mV at 25°C [26][9]. At a load current of 2 A, the 5 mV voltage drop at the VDD shunts is amplified to 1 V, resulting in a relative error of 0.3 %. An additional error of at most

[8]Experiments in which we used another current-sense amplifier with a significantly lower error (e.g., the INA296) gave us overall almost identical results.

[9]The MCC128 was placed outside the server case with a sufficiently stable ambient temperature of 22 °C to 24 °C.

0.154 % is introduced by the voltage divider that steps down the 12 V bus voltage.

4.6 Benchmarks

The described methodology was used to compare RAPL and reference measurements for DRAM and PMM on the Ice Lake-SP and Broadwell-EP systems during the execution of different workloads:

- Idle workloads: `sleep`, busy-waiting
- Memory intensive: different kernels of the STREAM benchmark at varying thread count, memory writes, memory reads, dot product
- CPU intensive: matrix multiplication (DGEMM), packed add, packed multiplication

When allocating on DRAM, we spawned between 1 and 32 threads, while we limited the number of threads to 4 for PMM. OpenMP threads were pinned to the CPU cores using the environment variables `OMP_PLACES=cores` and `OMP_PROC_BIND=true` for all experiments. For DRAM measurements, `numactl -m 0 -N 0` was employed to allocate memory on the local DRAM of socket 0 and to bind threads to socket 0. `numactl -m 2 -N 0` was used for PMM. The GCC 8.5.0 compiler was used throughout all experiments. Due to the limited availability of DIMM risers and data-acquisition channels, only memory populations with up to two DIMMs per socket were considered[10].

5 RESULTS

The following section presents the results from the validation experiments of the RAPL memory domain on the Ice Lake-SP and Broadwell-EP systems, as described in the previous section.

5.1 Temporal Resolution of RAPL

First, we empirically determined the update interval of the RAPL energy counters on both systems with default memory configuration as described in Table 1. For this, we used the code by Schöne et al. [33], which busy waits for an update of a RAPL MSR and measures the number of CPU cycles between consecutive updates using the `rdtscp` instruction. The code measures this time gap for 10 000 consecutive updates on socket 0 of the system for RAPL's package and memory domains. The difference in clock cycles is then multiplied by the clock speed to compute the update interval in milliseconds.

We executed these experiments five times on each system and show a representative excerpt of the update interval for 1000 consecutive updates of the RAPL memory domain for both systems in Figure 6. The results reveal that the update interval is subject to a substantially higher variation on the Ice Lake-SP system compared to Broadwell-EP. On the Broadwell-EP system, most counter updates are observed after a gap between 0.95 ms and 1.05 ms, averaging to 1.01 ± 0.11 ms. In contrast, the update rate on the Ice Lake-SP System is spread between 0.7 ms and 1.6 ms in an apparently bimodal distribution, averaging to 0.99 ± 0.26 ms. For the Broadwell-EP system, the outliers close to 2 ms strike out. The update interval distribution of the Package domain is overall similar to the memory domain on both systems and is omitted here. However, the outliers

[10]This approach can be extended to up to 16 DIMMs by stacking 8 MCC128 devices and using 16 risers.

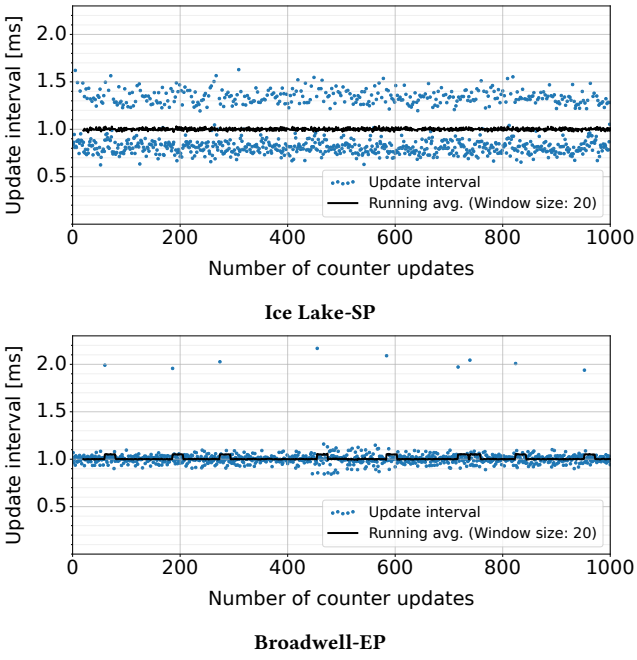


Figure 6: RAPL memory domain update interval variation.

close to 2 ms were not observed for the package domain on the Broadwell-EP system.

Our results indicate that RAPL energy counters either need to be oversampled at a substantially higher rate greater than 2 kHz or a lower rate is required to average out update rate variations.

5.2 Ice Lake-SP

Figure 7 shows the correlation between RAPL and our reference measurements on socket 0 of the Ice Lake-SP system for different memory populations. To facilitate the visualization, only up to 20 random samples per workload for each memory configuration are shown, and we do not visually indicate the different workloads. Interested readers can find the workload-specific visualization for each memory configuration in Figures 11 to 15 in the appendices.

The results indicate that RAPL reports a significantly higher power consumption than the reference measurements at the DIMM. If the memory is idle, RAPL measurements can be subject to a relative offset as high as 120%, for example, at idle, reference measurements report around 1.9 W for 32GB DDR4 memory, while RAPL reports 3.8 W to 4.2 W. At higher load on the memory, this absolute offset decreases slightly and the relative offset decreases significantly to around 10 % at maximum load for DRAM. PMM shows an overall different fit here and is subject to an even higher absolute offset than DRAM, even when PMM is idle and memory is only allocated on DRAM. While the different workloads show a wider spread of the power measurements on DRAM, PMM consumes around 11.5 W to 13 W under load.

During the experiments, a substantial inconsistency between RAPL measurements for the memory on both sockets was observed, which is shown in Figure 8. The reference measurements report

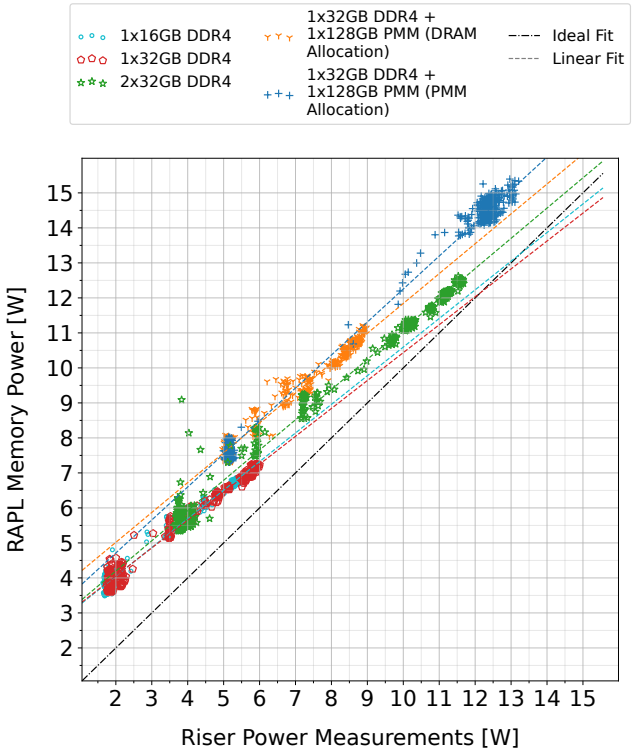


Figure 7: RAPL vs. reference power measurements for different memory populations for socket 0 on the Ice Lake-SP system

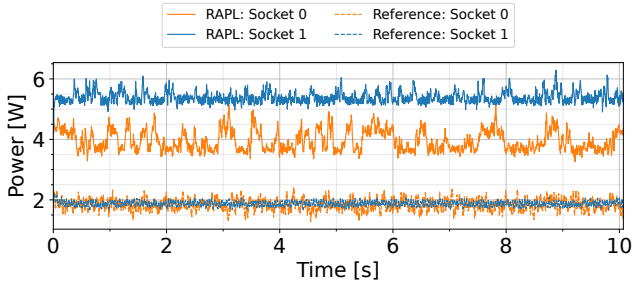


Figure 8: RAPL vs. reference measurements for the memory of both sockets on the Ice Lake-SP system at idle with 1x32GB DDR4 per socket.

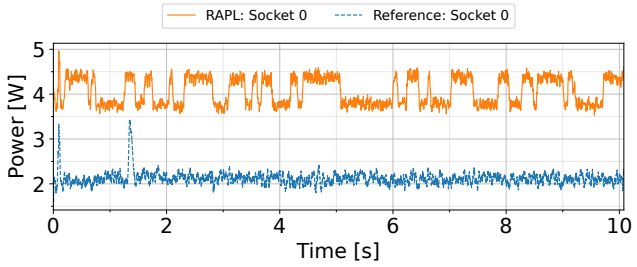


Figure 9: RAPL vs. reference measurements for the memory of socket 0 on the Ice Lake-SP system for the *memory copy* kernel with 32 threads and 1x32GB DDR4 per socket.

roughly the same power consumption for both sockets. Measurements on socket 0 are subject to a higher variation than on socket 1, which can be explained by the RAPL measurements being performed by socket 0 as well. However, while reference measurements coincide mostly for both sockets, the respective RAPL values diverge. This is reinforced on the same system with full DIMM population (see Table 1), for which RAPL reports 35.4 W for the memory domain of socket 0 and 41.9 W for socket 1 in idle state[11].

Furthermore, our results show that RAPL measurements are subject to a temporal inconsistency. This effect can already be seen in Figure 8, but is particularly prominent during the execution of the *memory copy* kernel, as visualized in Figure 9. While reference measurements are subject to a minor variation and mostly constant around 2.1 W throughout the measurements, corresponding RAPL measurements alternate between 3.5 W and 4.5 W. These inconsistencies can be observed for all workloads with low load on the memory, i.e., all single-threaded workloads, but not for the STREAM benchmark at a higher number of threads. In particular, it can be observed for busy waiting but not during `sleep`.

5.3 Broadwell-EP

Using the same experimental setup, we conducted RAPL memory measurements for the Broadwell-EP system with 1x16 DDR4 memory installed on each socket. Figure 10 shows the resulting correlation between RAPL and reference measurements. While RAPL reports significantly higher power than the reference measurements on the Ice Lake-SP system, RAPL and reference measurements match more closely on the Broadwell-EP system. At idle, RAPL slightly overestimates, and for a reference power consumption greater than 3.8 W, RAPL slightly underestimates. For workloads with a medium load on the memory, for example, the *dot product* kernel or the *memory read* kernel, RAPL, and reference measurements agree well. However, more samples deviate strongly from a linear fit.

5.4 RAPL Error Analysis

As demonstrated by our experimental results, RAPL significantly overestimates the reference power consumption on the Ice Lake-SP system, while measurements match more closely with a relative error of at most 10% when not idle on the Broadwell-EP system. In addition, our results show that the offset depends on whether PMM is installed. An extensive study of the memory power delivery and implementation of RAPL identified several potential causes for this discrepancy, which we will discuss in the following.

5.4.1 Sensing Point at the Voltage Regulator. As described in Figure 1, the memory DIMMs are primarily powered by one or multiple voltage regulators (VRs) on the mainboard, also on systems with fully integrated voltage regulators (FIVRs).

[11] Measured repeatedly using `perf stat -e power/energy-ram/ -I10000 -per-socket`, i.e., averaging over 10 seconds for each socket.

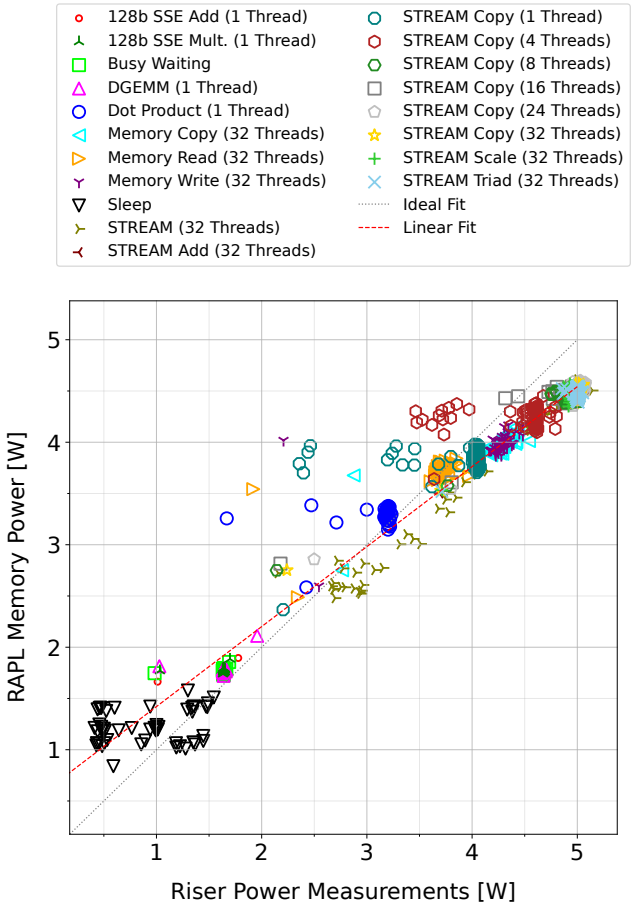


Figure 10: RAPL vs. reference power measurements for socket 0 for different workloads on the Broadwell-EP system with 1x16GB DDR4 per socket.

Such a VR internally measures the current and voltage and communicates it via SVID to the CPU, which utilizes this data for RAPL. This feature is called IMON and usually refers to the measurement of the output current of the VR. In this case, the measurement point for RAPL would be expected to be close to our reference measurements, which are based on measurements directly at the DIMM pins. The exact specification of the demands of voltage regulators and their current sensing capabilities are defined by Intel in the non-disclosed *VR13* / *VR13.HC* standard for the Ice Lake-SP platform. The latest published version of the VR specification is available for the *VR11.1* standard [15] from 2009. According to this document, IMON explicitly measures the "output load current" of the voltage regulator.

An inspection of the Ice Lake-SP system identified the VR controller IC *PXE1410CDM-G003* as a candidate for controlling power delivery to the memory. Datasheets are only available for other variants of this chip with different numbers of phases [7]. Apart from monitoring the output current of each channel, these chips also measure the input current and voltage for the input to the VR. The supported input power monitoring suggests that RAPL memory measurements could be based on input instead of output power monitoring on the Ice Lake system. Other datasheets for VRs typically used in server systems support this hypothesis. For example, the *TPS544C26* [35], which is designed for powering DDR5 memory, features "input power monitoring for DDR5 memory RAPL". Similarly, the *ISL69133* [31] supports "input current sensing required for NVDIMM". Both PMM and NVDIMM have in common that they are primarily powered via 12 V.

If RAPL was based on input power monitoring of the VR, it would mean that RAPL measurements also include the VR power losses. The efficiency of the VRs strongly depends on the output current and typically peaks around 90 % at a certain load current. For lower output current, the efficiency drops significantly and can be lower than 50 %. This matches our observation that the positive offset of RAPL against DIMM measurements at low current is reduced at a higher load on the memory. This potentially also explains the higher offset if PMM is installed. As PMM is primarily powered over 12 V compared to 1.2 V for DRAM, the output current is 10× lower at identical power consumption.

By only installing two DIMMs per socket, the voltage regulators operate well below the efficiency range they were designed for, which reinforces this effect. For full DIMM populations, RAPL is still expected to overestimate the actual memory power consumption. For example, if the voltage regulator has a peak efficiency of 92 %, RAPL can be expected to overestimate by at least 8 %.

The changed measurement point for RAPL could also explain the temporal inconsistencies demonstrated by Figure 8 as the VRs are typically implemented using multiphase buck converters. To improve the energy efficiency at low output current, *phase shedding* is often implemented, which shuts down certain phases at low output current to improve energy efficiency [2]. This way, small changes in the DIMM power consumption can result in a phase being switched on or off at the VR, resulting in a jump in input power consumption while the output power only changes slightly.

5.4.2 Different components included. Another potential reason for the difference between RAPL and reference measurements is that RAPL covers the power consumption of other components as well. For example, it is likely that the VR that powers the memory also provides $VCCD_{0123}$ and $VCCD_{4567}$ to the CPU for powering the memory interface (see Figure 1). In addition, our reference measurements are based on power measurements at only two pins which contribute to the total power consumption the most. The power consumption of the other pins and the heat dissipation at the current-sense shunts are not covered by our measurements but might be included in RAPL. However, this alone cannot explain the large offset of RAPL according to our measurements.

5.4.3 Accuracy of current sensing. Another error source is the actual accuracy of the current sensing at the VR (IMON). As already stated by Gough et al. in 2016 [2], the current sensing feature at VRs is typically optimized for the highest current it can deliver and is subject to an almost constant offset error. However, if the load current reduces, the relative error of the current sensing will increase significantly, resulting in a substantial inaccuracy of RAPL measurements for the memory if systems are populated with significantly lower memory capacity than they have been designed

for [2, p. 63]. The *VR11.1 Design Guidelines* give concrete values for the maximum current monitoring error. At a load of more than 50%, the error needs to stay below 15%, at a load up to 30%, an error of 24% is allowed, and at a load of 15%, an error of up to 50% is recommended [15]. Such requirements are not publicly documented by Intel for modern systems, but some device vendors provide typical accuracy values of their current sensing circuits in the datasheets. For example, the *MP86901C* [38], which is advertised as very "reliable and accurate", has an accuracy of ±2% at 30A load, ±4.0% at 10A load, and ±7% at 5*A* load. However, if only a single DRAM DIMM is used, the current at VDD is usually less than 2*A* and the error is increased. Infineon's *TDA38640* [6], for example, has a typical accuracy of ±6% for output current monitoring (IMON) at maximum load — the accuracy is expected to be lower at light load.

5.5 Implication on the Use of RAPL Memory Measurements

Overall, our results indicate that RAPL significantly overestimates the actual power consumption of the memory module itself. A possible reason is that the RAPL implementation of our Ice Lake-SP system includes power losses at the voltage regulator level and the power consumption of components outside the DIMM. As a consequence, the offset between RAPL and reference measurements strongly depends on the voltage regulators used on the mainboard. This is also shown by the differences to previous architectures, for example, Broadwell-EP. The inter-socket inconsistency (see Figure 8) of RAPL can also be explained by some components being only powered by the VR of socket 1.

The different measurement points — *narrow* covering only DIMM versus *wider* covering input power of the larger memory subsystem including VR losses — both offer unique insight. For example, the *wider* measurements are more suitable to account for all power consumed within a system right after the power supply unit. Therefore, estimating the total system power consumption based on RAPL package and memory measurements would be easier in the *wider* measurement point we assume for our Ice Lake-SP system. The *wider* measurement point is also more suited to gauge the overall impact of an optimization. However, modeling and analyzing the power consumed by memory for certain workloads can be done more accurately with *narrow* only covering the DIMMs. Using the *wider* RAPL measurements would include the confounding factors of VR efficiency as well as inconsistencies we observed across sockets (see Figure 8) and over time (see Figure 9). Our results also show that a focus on high-power configurations (memory under load, full capacity installed) can help limit the impact of these discrepancies when utilizing RAPL for memory power modeling.

While we executed the experiments only on a single Intel Ice Lake-SP system, we expect that our findings also apply to other heterogeneous memory systems, in particular, those that provide 12 V to the memory modules. For DDR5 registered DIMMs, which are primarily powered over 12 V and feature an on-DIMM power management integrated circuit (PMIC) that converts 12 V to a set of different voltages required by the memory [17], RAPL is also expected to include power losses at the VR. This is confirmed by a datasheet for a DDR5-compatible VR controller for Intel systems [35].

6 CONCLUSION

This study provides an extensive overview of the power measurement methodology using DDR4 DIMM extenders. We described our hardware instrumentation setup for collecting power consumption measurements for heterogeneous memory that can serve as a blueprint for conducting similar experiments. To verify the accuracy of this measurement approach, we validated it on a Broadwell-EP system and observed overall agreement with the results from a similar system in prior work.

Our results from a recent Intel Ice Lake-SP platform indicated that RAPL energy counters of the memory domain report notably different readings than in older architectures. Regardless of the memory type (DRAM or PMM), RAPL measurements overestimate the actual power consumption of the DIMMs. The relative offset between reference and RAPL measurements increases with lower load on the memory or smaller memory capacity installed. A plausible reason for this is a change in the RAPL's input measurement point that now includes power losses at the voltage regulators of the memory system. We conclude that this must be taken into account when using RAPL for memory power consumption estimations or performance on Intel Ice Lake-SP or similar architectures.

Investigating fully populated memory channels, platforms supporting newer DDR5 memory (Sapphire Rapids), and other system components, such as the CPU package and GPUs, are future directions for exploration.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback to improve the manuscript. Special thanks are extended to Paul Kapinos, Marcus Wagner, Sascha Bücken, and Simon Schwitanski for providing access to the hardware and assisting with the organizational matters, as well as Vincent M. Weaver for sharing additional details on the experimental setup of their work. Parts of this work have been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 446185093. Computing resources were granted by RWTH Aachen University under project thes1426.

REFERENCES

[1] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 2016. A Validation of DRAM RAPL Power Measurements. In *Proceedings of the Second International Symposium on Memory Systems*. ACM, Alexandria VA USA, 455–470. ISBN: 978-1-4503-4305-3. DOI: 10.1145/2989081.2989088.

[2] Corey Gough, Ian Steiner, and Winston Saunders. 2015. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress, Berkeley, CA. ISBN: 978-1-4302-6638-9. DOI: 10.1007/978-1-4302-6638-9.

[3] Daniel Hackenberg, Robert Schone, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. 2015. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, Hyderabad, India, 896–904. ISBN: 978-1-4673-7684-6. DOI: 10.1109/IPDPSW.2015.70.

[4] Thomas Ilsche. 2020. *Energy Measurements of High Performance Computing Systems: From Instrumentation to Analysis*. Ph.D. Dissertation. Technical University Dresden. Retrieved June 12, 2023 from https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-716000.

[5] Thomas Ilsche, Daniel Hackenberg, Stefan Graul, Robert Schone, and Joseph Schuchart. 2015. Power measurements for compute nodes: Improving sampling rates, granularity and accuracy. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. IEEE, Las Vegas, NV, USA, 1–8. ISBN: 978-1-5090-0172-9. DOI: 10.1109/IGCC.2015.7393710.

[6] Infineon. 2023. TDA38640 OptiMOS iPOL 40A Single-voltage Synchronous Buck Regulator with SVID and I2C. Retrieved Feb. 10, 2024 from https://www.infineon.com/dgdl/Infineon-TDA38640-0000-DataSheet-v02_08-EN.pdf?fileId=8ac78c8c80027ecd018042f2337f00c9.

[7] Infineon. 2020. VR13 and VR12.5 Multi-rail / Multiphase Digital Controllers. Infineon, (July 20, 2020). Retrieved Oct. 15, 2023 from https://www.infineon.com/dgdl/Infineon-Multiphase_digital_controllers_PXE1_PXM1-DataSheet-v01_00-EN.pdf?fileId=5546d46272e49d2a01736b7cd95b3c4a.

[8] Texas Instruments. 2023. CS-AMPLIFIER-ERROR-TOOL Calculation tool. Retrieved Aug. 16, 2023 from https://www.ti.com/tool/CS-AMPLIFIER-ERROR-TOOL.

[9] Texas Instruments. 2022. INAx180 Low- and High-Side Voltage Output, Current-Sense Amplifiers. Retrieved Oct. 3, 2023 from https://www.ti.com/lit/ds/symlink/ina2180.pdf.

[10] Intel. 2022. 3rd Gen Intel® Xeon® Scalable Processor, Codename Ice Lake-SP Datasheet, Volume One: Electrical. Retrieved Aug. 17, 2023 from https://www.intel.com/content/www/us/en/content-details/732800/3rd-gen-intel-xeon-scalable-processor-codename-ice-lake-sp-datasheet-volume-one-electrical.html.

[11] Intel. 2023. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. en. (2023). Retrieved Feb. 10, 2024 from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[12] Intel. 2021. Intel® Memory Latency Checker v3.9a. Retrieved July 16, 2023 from https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[13] Intel. 2023. Intel® Server Board M50CYP2SB Technical Product Specification – Rev. 1.42. Retrieved Feb. 10, 2024 from https://www.intel.com/content/dam/support/us/en/documents/server-products/single-node-servers/m50cyp2sb-server-board-tps.pdf.

[14] Intel. 2022. Running Average Power Limit Energy Reporting / CVE-2020-8694 , CVE-2020-8695 / INTEL-SA-00389. Retrieved Aug. 25, 2023 from https://www.intel.com/content/www/cn/zh/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[15] Intel. 2009. Voltage Regulator Module (VRM) and Enterprise Voltage Regulator-Down (EVRD) 11.1 - Design Guidelines. Retrieved Sept. 2, 2023 from https://www.intel.it/content/dam/doc/design-guide/voltage-regulator-module-enterprise-voltage-regulator-down-11-1-guidelines.pdf.

[16] JEDEC. 2019. JEDEC Standard No. 21C, Release 29 - DDR4 SDRAM Registered DIMM Design Specification. Retrieved Sept. 6, 2023 from https://www.jedec.org/standards-documents/docs/module4_20_28.

[17] JEDEC. 2022. JEDEC Standard No. 305 DDR5 Load Reduced (LRDIMM) and Registered Dual Inline Memory Module (RDIMM) Common Specification. (Jan. 2022). Retrieved Oct. 18, 2023 from https://www.jedec.org/standards-documents/docs/jesd305.

[18] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 3, 2, 1–26. DOI: 10.1145/3177754.

[19] Rahul Khanna, Fadi Zuhayri, Murugasamy Nachimuthu, Christian Le, and Mohan J Kumar. 2011. Unified extensible firmware interface: An innovative approach to DRAM power control. In *2011 International Conference on Energy Aware Computing*. IEEE, Istanbul, Turkey, 1–6. DOI: 10.1109/ICEAC.2011.6136703.

[20] Anara Kozhokanova, Bo Wang, Christian Terboven, and Matthias Mueller. 2023. Power-aware Computing with Optane Persistent Memory Modules. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, St. Petersburg, FL, USA, 26–31. DOI: 10.1109/IPDPSW59300.2023.00017.

[21] Linux. 2023. Numa(3) - Linux manual page. Retrieved Nov. 14, 2023 from https://man7.org/linux/man-pages/man3/numa.3.html.

[22] Linux. 2023. Numactl(8) - Linux manual page. Retrieved Nov. 14, 2023 from https://man7.org/linux/man-pages/man8/numactl.8.html.

[23] Linux. 2023. Perf: linux profiling with performance counters. (2023). Retrieved Nov. 14, 2023 from https://perf.wiki.kernel.org/index.php/Main_Page.

[24] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 355–371. ISBN: 978-1-72818-934-5. DOI: 10.1109/SP40001.2021.00063.

[25] Heike McCraw, James Ralph, Anthony Danalis, and Jack Dongarra. 2014. Power monitoring with PAPI for extreme scale architectures and dataflow-based programming models. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Madrid, Spain, 385–391. ISBN: 978-1-4799-5548-0. DOI: 10.1109/CLUSTER.2014.6968672.

[26] Measurement Computing Corporation. 2022. MCC128 - 16-bit Voltage Measurement DAQ HAT for Raspberry Pi. Measurement Computing, (Dec. 2022). Retrieved June 15, 2023 from https://www.mccdaq.com/PDFs/specs/DS-MCC-128.pdf.

[27] Microchip Technology, Inc. 2017. MCP2221 USB 2.0 to I2C/UART Protocol Converter with GPIO. Retrieved Feb. 10, 2024 from http://ww1.microchip.com/downloads/en/devicedoc/20005292c.pdf.

[28] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the Intel optane byte-addressable NVM. en. In *Proceedings of the International Symposium on Memory Systems*. ACM, Washington District of Columbia USA, 304–315. ISBN: 978-1-4503-7206-0. DOI: 10.1145/3357526.3357568.

[29] PMem.io. 2023. Libvmmalloc | PMDK. Retrieved Nov. 14, 2023 from https://pmem.io/pmdk/manpages/linux/v1.3/libvmmalloc.3/.

[30] PMem.io. 2023. MEMKIND. Retrieved Nov. 14, 2023 from https://pmem.io/memkind/manpages/memkind.3/.

[31] Renesas. 2018. ISL69133 Digital, Dual Output, 4-Phase Configurable, VR13/IMVP8 PWM Controller. Renesas, (Feb. 2018). Retrieved Oct. 15, 2023 from https://www.renesas.com/us/en/document/sds/isl69133-data-short.

[32] Christopher R. Robertson. 2008. *Fundamental Electrical and Electronic Principles*. (3rd ed.). Newnes, Amsterdam. ISBN: 978-0-7506-8737-9.

[33] Robert Schone, Thomas Ilsche, Mario Bielert, Markus Velten, Markus Schmidl, and Daniel Hackenberg. 2021. Energy Efficiency Aspects of the AMD Zen 2 Architecture. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Portland, OR, USA, 562–571. ISBN: 978-1-72819-666-4. DOI: 10.1109/Cluster48925.2021.00087.

[34] SK hynix. 2016. DDR4 SDRAM Registered DIMM Based on 4Gb A-die. Retrieved Feb. 10, 2024 from https://www.datasheets360.com/part/detail/hma42gr7afr4n-uh/181739709597928 4182/.

[35] Texas Instruments. 2022. TPS544C26 4-V to 16-V Input, 35-A Synchronous Buck Converter With SVID And I2C Interfaces. Texas Instruments, (Sept. 2022). Retrieved Sept. 3, 2023 from https://www.ti.com/lit/ds/symlink/tps544c26.pdf.

[36] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: a lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, 207–216. DOI: 10.1109/ICPPW.2010.38.

[37] Vincent M. Weaver. 2015. RAPL Userspace Access without perf - Source Code. Retrieved Sept. 3, 2023 from https://web.eece.maine.edu/~vweaver/projects/rapl/rapl-read.c.

[38] Mario Willeit. 2020. Solutions for Powering Intel and AMD SoCs. (2020). Retrieved Feb. 10, 2024 from https://media.monolithicpower.com/mps_cms_document/m/p/mps_solutions_for_powering_intel_and_amd_socs_22.04.2020.pdf.

[39] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (FAST'20). USENIX Association, Santa Clara, CA, USA, 169–182. ISBN: 9781939133120.

A APPENDICES

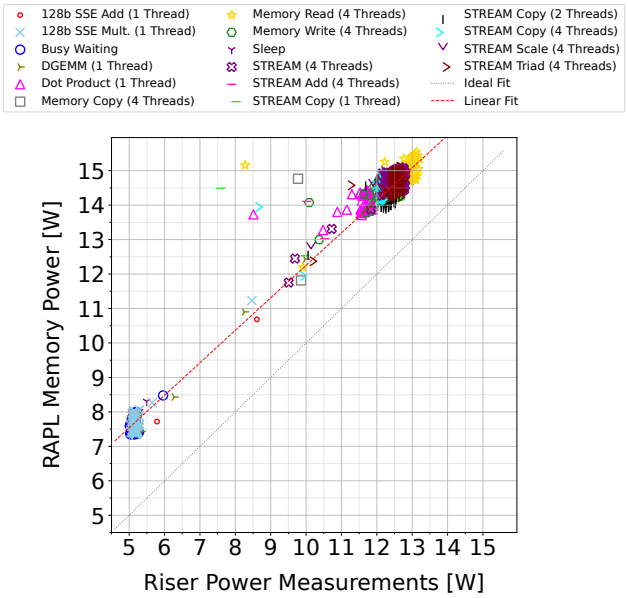
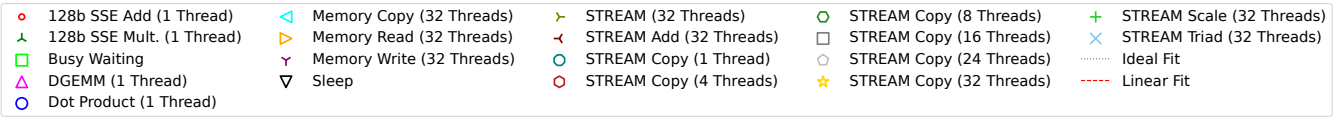


Figure 11: Ice Lake-SP: 32GB DDR4 + 128GB PMM (on PMM)

	Ice Lake-SP					Broadwell-EP
	128GB PMM + 32GB DRAM	128GB PMM + 32GB DRAM	1x16GB DDR4	1x32GB DDR4	2x32GB DDR4	1x16GB DDR4
Sleep	49.6%	48.4%	121.6%	118.1%	57.7%	35.3%
Busy Waiting	47.1%	47.6%	117.7%	107.4%	50.0%	7.9%
128b SSE Add. (1 Thread)	47.9%	47.3%	116.4%	108.8%	49.3%	8.2%
128 SSE Mult. (1 Thread)	49.4%	44.9%	119.2%	112.6%	46.5%	7.9%
Dot Product (1 Thread)	37.1%	20.8%	58.0%	54.8%	32.8%	4.0%
DGEMM (1 Thread)	49.2%	45.8%	119.0%	112.3%	50.3%	8.0%
Stream Copy (1 Thread)	29.9%	16.7%	46.2%	43.1%	23.4%	-2.3%
Stream Copy (8 Threads)	23.3%	***	33.4%	28.6%	10.3%	-8.8%
Stream Copy (32 Threads)	24.0%	***	26.8%	22.6%	8.3%	-9.5%
Memory Read (32 Threads)	42.5%	***	101.6%	87.1%	44.8%	-0.1%

[1] Allocation on DRAM [2] Allocation on PMM [3] Only up to 4 threads were used for PMM

Table 3: RAPL error comparison between the Ice Lake-SP and Broadwell-EP systems for different workloads and memory



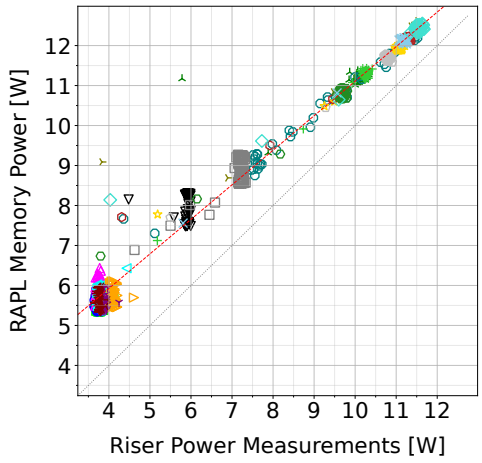


Figure 12: Ice Lake-SP: 2x32GB DDR4

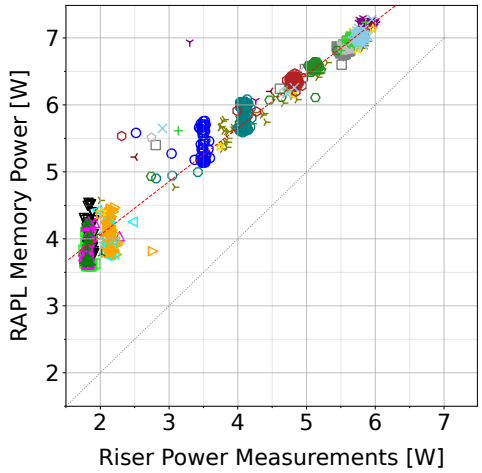


Figure 13: Ice Lake-SP: 32GB DDR4

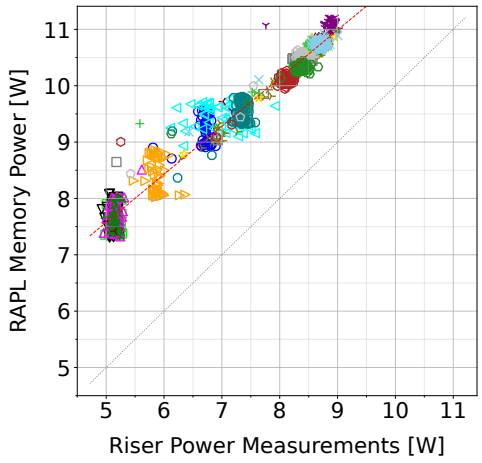


Figure 14: Ice Lake-SP: 32GB DDR4 + 128GB PMM (on DRAM)

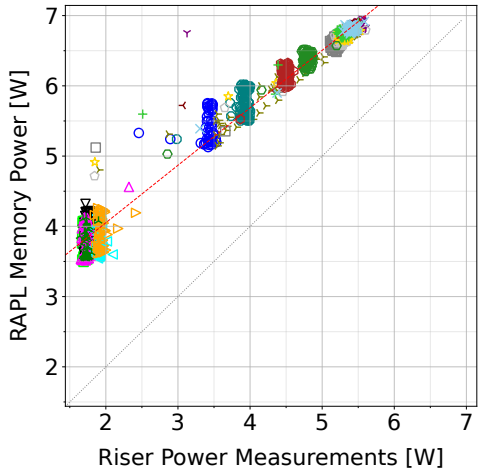


Figure 15: Ice Lake-SP: 16GB DDR4

Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays

Stephen Nicholas Swatman*
University of Amsterdam
Amsterdam, The Netherlands
s.n.swatman@uva.nl

Ana-Lucia Varbanescu
University of Twente
Enschede, The Netherlands
a.l.varbanescu@utwente.nl

Andy D. Pimentel
University of Amsterdam
Amsterdam, The Netherlands
a.d.pimentel@uva.nl

Andreas Salzburger
CERN
Geneva, Switzerland
andreas.salzburger@cern.ch

Attila Krasznahorkay
CERN
Geneva, Switzerland
attila.krasznahorkay@cern.ch

ABSTRACT

The layout of multi-dimensional data can have a significant impact on the efficacy of hardware caches and, by extension, the performance of applications. Common multi-dimensional layouts include the canonical row-major and column-major layouts as well as the Morton curve layout. In this paper, we describe how the Morton layout can be generalized to a very large family of multi-dimensional data layouts with widely varying performance characteristics. We posit that this design space can be efficiently explored using a combinatorial evolutionary methodology based on genetic algorithms. To this end, we propose a chromosomal representation for such layouts as well as a methodology for estimating the fitness of array layouts using cache simulation. We show that our fitness function correlates to kernel running time in real hardware, and that our evolutionary strategy allows us to find candidates with favorable simulated cache properties in four out of the eight real-world applications under consideration in a small number of generations. Finally, we demonstrate that the array layouts found using our evolutionary method perform well not only in simulated environments but that they can effect significant performance gains—up to a factor ten in extreme cases—in real hardware.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Mathematics of computing** → **Combinatorial optimization**; • **Information systems** → **Data layout**.

KEYWORDS

Morton curve, Z-order curve, space-filling curves, array layout, multi-dimensional data, evolutionary algorithms, caching, locality

*Also with CERN.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645034

ACM Reference Format:
Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger, and Attila Krasznahorkay. 2024. Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645034

1 INTRODUCTION

Structured multi-dimensional data are ubiquitous in high-performance computing [9]: three-dimensional fluid simulations, dense linear algebra operations, and stencil kernels are just a few examples of applications which rely fundamentally on multi-dimensional arrays. In spite of the importance of such applications, however, most modern computer systems have one-dimensional memories: from the perspective of the programmer, memory is nothing more than a very large one-dimensional array of bytes. This discrepancy between application requirements and hardware design requires programmers to carefully consider *array layouts*: injective functions which translate multi-dimensional indices into one-dimensional memory addresses.

Although array layouts do not impact the functional properties of programs, choosing a suitable layout can significantly impact application performance in modern processors with complex cache hierarchies [48]. Exploiting these caches is of critical importance to achieving high performance in all but purely compute-bound applications, but doing so requires locality of access—both temporal and spatial—in memory. Kernels often exhibit locality in multiple dimensions, and a well-chosen array layout maximizes the degree to which this application-level locality is translated to the address-level locality that caches are designed to exploit; as a result, that layout increases the efficacy of hardware caching and—by extension—the performance of an application.

Data in two-dimensions is commonly laid out in *row-major* order (shown in Figure 2a for an 8×8 array) or *column-major* order (Figure 2t) which provide good locality of access in a single dimension, but poor locality in all others. Thankfully, the design space for data orderings—in two dimensions or more—extends far beyond these canonical layouts: the *Morton* layout (Figure 2f), for example, is a layout based on a space-filling curve which provides balanced locality between multiple dimensions [46, 62]. Our work explores a family of data layouts which generalize the Morton order, and allow

(a) Row-major layout (b) Morton layout

Figure 1: Two-dimensional arrays laid out in memory along the gray arrows. An application accesses the array diagonally along the red arrows. Application locality is shown above, memory locality is shown below.

us to carefully tune the cache behavior in multiple dimensions to match a given application.

The design space of the aforementioned family of data layouts is dauntingly large; indeed, the number of possible layout for arrays at scales applicable to real-world problems is so large that it renders exhaustive search infeasible. In order to find suitable array layouts in tractable amounts of time, we propose to employ genetic algorithms—heuristics known to be able to efficiently find high-quality solutions in large search spaces [35]. To this end, we design a chromosomal representation of Morton-like array layouts, as well as a fitness function that uses cache simulation to estimate the performance of individual array layouts. Finally, we evaluate our evolutionary strategy and the array layouts it discovers.

In short, our paper makes the following contributions:

- We characterize the design space given by a generalization of the Morton array layout, and we show that that the size of this design space renders exhaustive search infeasible (Section 3);
- We propose an evolutionary methodology based on genetic algorithms for exploring the aforementioned design space based on the simulated cache-friendliness of layouts (Section 4);
- We design and execute a series of experiments to assess the accuracy of our fitness function, the efficacy of our evolutionary process, and the performance of the discovered array layouts, showing that our method can improve performance up to a factor ten (Section 5).

2 BACKGROUND AND RELATED WORK

In this section, we provide a brief overview of the basic concepts and notations which are essential to the remainder of this paper, and highlight relevant related work.

2.1 Indexing Functions and Canonical Layouts

Dense n-dimensional arrays can be imagined as structured grids in which each element is assigned to exactly one point in $\mathbb{N}^n$. In most modern processors, multi-dimensional arrays are a software-level abstraction over the one-dimensional memory of the machine; in order to actually access multi-dimensional data, we need to define a function that converts indices in n dimensions to memory addresses[1]. We refer to the class of such functions as *indexing functions*, and they are isomorphic to *array layouts*. In short, an n-dimensional indexing functions is an injective (often bijective) function of the following type, where N_i represents the size of the array in the ith dimension, $\bigtimes$ is the generalised Cartesian product, and $[\![a, b]\!]$ is the integer interval from a to b:

$$f : \bigtimes_{i=0}^{n-1} [\![0, N_i - 1]\!] \to \left[\!\!\left[0, \left(\prod_{i=0}^{n-1} N_i \right) - 1 \right]\!\!\right] \tag{1}$$

In a multi-dimensional grid, we denote the elements along a given axis—that is to say, the sequence of elements for which all indices except one are fixed—as *fibers* [41]. In a two-dimensional case, fibers along the x-axis are known as *rows*, and fibers along the y-axis as columns. In order to facilitate the description of arrays in three or more dimensions, we use the term *mode-m* fibers to describe fibers along the mth dimension, such that mode-0 fibers are synonymous with rows, mode-1 fibers refer to columns, and so forth.

The most common group of multi-dimensional indexing functions are the *canonical* layouts, sometimes known as the *lexicographic* layouts or, in the two-dimensional case, the *row-* and *column-major* layouts. In a canonical layout, one-dimensional array indices are calculated according to Equation 2, in which $x_0, \ldots, x_{n-1}$ are components of the n-dimensional index, and $N_0, \ldots, N_{n-1}$ represent the size of the array in each dimension:

$$f(x_0, \ldots, x_{n-1}; N_0, \ldots, N_{n-1}) = \sum_{i=0}^{n-1} \left(\prod_{j=0}^{i-1} N_j \right) x_i \tag{2}$$

An important corollary of Equation 2 is that the mode-0 fibers are contiguous in memory i.e., Equation 3 holds:

$$f(x_0 + 1, x_1, \ldots, x_{n-1}) = f(x_0, x_1, \ldots, x_{n-1}) + 1 \tag{3}$$

It is worth noting that the calculation of addresses in column-major layout—in which the mode-1 fibers are contiguous—is also given by Equation 2, with the order of the indices and sizes swapped. The canonical array layouts achieve perfect spatial locality in one dimension: if a kernel accesses memory along mode-m fibers, then a canonical layout where the mth dimension is major will provide the optimal translation between locality in the multi-dimensional space to locality in memory. Many real world applications, however, exhibit locality in multiple dimensions; a kernel might, for example, iterate diagonally over an array; an example of this—and the resulting locality in memory—is given in Figure 1a.

The performance of canonical storage layouts has been studied extensively. Park et al. discuss methods for compensating for the weaknesses of canonical layouts using tiling and recursive layouts [48]. Similarly, Kowarschik and Weiß propose a variety of strategies that mitigate cache misses in canconical storage layouts for numerical applications [42]. Weinberg et al. propose a metric for the locality of array layouts [66]. Jang et al. analyze the performance of access patterns in multi-dimensional data in graphics processing

[1] In reality, address calculations must also consider array offsets (the address of the first element) and scales (the size of each element). We skip over these complications as they are handled transparently by address generation units in modern hardware, and they affect all array layouts in the same manner.

units (GPUs) [40]. Che et al. propose a method for automatically optimizing storage layouts [16].

2.2 Morton Layouts

The Morton order is a notable example of a non-canonical array layout that provides balanced locality in multiple dimensions. It is conceptually simple to understand, efficient to implement in commodity hardware (as we will show in Section 3.3), and it has been shown to positively affect the efficacy of hardware caches: Al-Kharusi and Walker show the efficacy of the Morton layout in molecular dynamics applications [5], Perdacher et al. describe its benefits in matrix decomposition [51], and Thiyagalingam et al. provide an in-depth performance analysis of this array layout in a range of kernels [62]. Chatterjee et al. show the applicability of Morton layouts—as well as other non-canonical layouts—in matrix multiplication [15], and this work is expanded upon in [14]. Applications of the Morton order in more than two dimensions have been studied by Pawłowski et al. [50]. Mellor-Crummey et al. show the applicability of array layouts based on space-filling curves—like the Morton layout—for irregular applications [44]. The practical applicability of the Morton layout is further evidenced by the Opie compiler, which employs Morton array layouts natively [24].

The performance benefits of the Morton layout stem from its spatial structure: an example—which justifies why this layout is sometimes known as the *Z-order layout*—is given in Figure 1b; note the difference in locality in the address space compared to the canonical layout (Figure 1a). The Morton order layout has also been applied to data movement in parallel systems by Walker and Skjellum [65], and Deford and Kalyanaraman have applied the layout to workload distribution in parallel processes [20]. Bader explores a variety of applications of space-filling curves in scientific programs [10]. Armbrust et al. explore the application of Morton curves for the storage of databases, reducing the total amount of data read from persistent storage [8]; although the aforementioned paper considers a much higher level of abstraction than the methods in this paper—which operate at the level of hardware caches—we believe that the methods presented in this paper may generalize to a broader range of applications, including databases.

In the Morton order, multi-dimensional indices can be converted to one-dimensional addresses in a variety of ways. The Moser–de Bruijn sequence [36] is commonly used as it allows efficient conversions in two dimensions, but this method requires us to store the Moser–de Bruijn sequence in memory, and accessing this sequence causes additional overhead. Therefore, we prefer a different method based on the interleaving of the (unsigned) binary representation of multi-dimensional indices. As an example, the two-dimensional index $(3, 5)$ can be bijectively mapped into one-dimensional memory by finding the binary expansions of the indices i.e., $(011_2, 101_2)$, and interleaving the bits yielding $100111_2 = 39_{10}$. This is equivalent to first dilating and shifting the binary expansions of the numbers, and then taking their bitwise disjunction (OR): the first index is dilated yielding 000101_2 while the second index is dilated and shifted left yielding 100010_2. Taking the bitwise disjunction of these numbers yields the same address as using the interleaving strategy. The computation of Morton indices through bit manipulation can be extended to an arbitrary number of dimensions; the three-dimensional index $(3, 5, 4)$ expands to $(011_2, 101_2, 100_2)$, and the resulting memory address is $110001011_2 = 395_{10}$. Note that the relative significance of bits in each of the input indices is preserved in the output address. Gottschling et al. present the idea that the Morton layout can be generalized by allowing arbitrary bit-interleaving orders [27, 28], which is foundational to our work. This idea is further expanded on by Walker [64].

2.3 Genetic Algorithms

Genetic algorithms are a class of heuristics introduced by Holland which are designed to solve optimization and search problems by emulating the process of evolution as it happens in the natural world [34]. In genetic algorithms, *generations* of *individuals* i.e., sets of candidate solutions, iteratively explore a design space through genetic operators. In particular, *crossover* operators model the combination of the genetic material of two or more individuals [49], and *mutation* operators model random changes to the gene pool [57]. In genetic algorithms, individuals are removed from the population based on their *fitness* i.e., the quality of the solution they represent to the problem posed [55]. Genetic algorithms have seen successful application in an extremely broad range of fields, ranging from drug discovery [61] to music composition [25]. Genetic algorithms have also proven useful for design space exploration in computer systems; Pimentel shows that they can be applied in the design of embedded systems [52]. Sapra and Pimentel show that a broader class of evolutionary approaches can be used in the design of neural networks [54]. The optimization problem we consider in this paper is combinatorial in nature, and the application of genetic algorithms to such problems has also been extensively studied and proven across a variety of domains [7, 26, 31, 47]

3 GENERALIZED MORTON LAYOUTS

The Morton layout functions by interleaving the bits of the input indices in a fixed pattern: bits are drawn from each of the inputs in a round-robin manner. In this section, we generalize this idea, allowing bits to be interleaved in arbitrary order. This gives rise to more specialized layouts with different structure and, as a result, different extra-functional properties [27, 28, 64]. Figure 2 shows all 20 layouts that are given by different bit interleaving orders for an 8×8 array. As with the standard Morton layout, the generalized Morton layout can be applied to any number of dimensions. As an example, the following three-dimensional layout selects two bits from the second index, one bit from the third index, then two bits from the first index, etc.:

$$f(011_2, 101_2, 100_2) = \begin{array}{r} 000011000_2 \\ \vee\ 000100001_2 \\ 100000000_2 \\ \hline 100111001_2 \end{array} = 313_{10} \tag{4}$$

Our goal is to find Morton-like layouts i.e., bit-interleaving patterns, that improve application performance through an increase in cache efficacy. In this section, we will show that the design space for such layouts is very large, motivating the use of genetic algorithms. This necessitates a chromosomal representation of layouts, which we also present in this section. In addition, we describe how the canonical layouts can be described using the same representation,

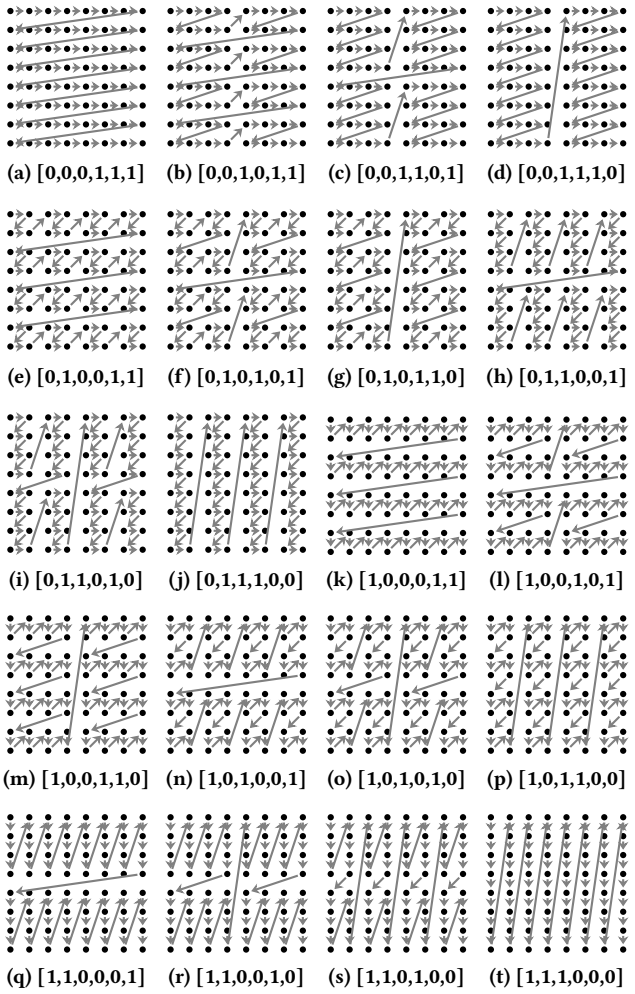


Figure 2: All 20 layouts for 8×8 arrays generated by the family of indexing schemes described in Section 3. Note that Figure 2a corresponds to a row-major layout, while Figure 2t corresponds to a column-major layout.

and we delve into practical considerations such as the computational cost of computing indices and support for same-instruction multiple-data (SIMD) processing.

3.1 Enumerating Layouts

We can characterize Morton-like layouts by the bit scattering pattern applied to each of the inputs (e.g., for Equation 4, the first index is scattered to the fourth, fifth, and eighth bits). However, such a characterization is unsound in the sense that is allows us to describe invalid layouts: if two bits from any of the input indices are mapped onto the same bit in the output, the bitwise disjunction becomes an information-destroying operation and the layout becomes non-injective—that is, it would cause multiple multi-dimensional indices to map onto the same location in memory, making the layout unusable.

We can instead characterize layouts in a manner that is both complete and sound by enumerating the *source* of each bit in the output index. In the remainder of this work we shall denote array layouts using sequences of the form $[i_0, \ldots, i_{n-1}]$, indicating the source indices in order of increasing bit significance: the least significant bit in the output index is drawn from the i_0th input index, the second-least significant bit is drawn from the i_1th input, and the most significant bit is drawn from the i_{n-1}th input. Note that each input bit must be used once and only once: whenever a bit is to be drawn from a given input index, we implicitly use the least significant bit for that input which has not yet been consumed. For the layout shown in Equation 4, the two least significant bits are drawn from the second input, the third-least significant bit is drawn from the third input, and so forth: the resulting array layout is denoted using the sequence $[1, 1, 2, 0, 0, 1, 2, 0, 2]$.

The aforementioned characterization of multi-dimensional layouts gives rise to families of layouts. The family of layouts over n inputs, where each input has $b_0, \ldots, b_{n-1}$ bits, is isomorphic to the set of permutations of the multiset $S = \{0 : b_0, \ldots, n-1 : b_{n-1}\}$. We denote this set of permutations as $\mathfrak{S}(S)$. For convenience, we obviate the intermediate multiset such that $\mathfrak{S}'(b_0, \ldots, b_{n-1}) = \mathfrak{S}(\{0 : b_0, \ldots, n-1 : b_{n-1}\})$. We can then determine the total number of possible layouts as the number of multiset permutations of $\mathfrak{S}'$ [13, p. 42]:

$$|\mathfrak{S}'(b_0, \ldots, b_{n-1})| = \binom{\sum_{i=0}^{n-1} b_i}{b_0, \ldots, b_{n-1}} = \frac{\left(\sum_{i=0}^{n-1} b_i\right)!}{\prod_{i=0}^{n-1}(b_i!)} \tag{5}$$

3.2 Including Canonical Layouts

It is worth noting that canonical layouts over arrays for which the size in each dimension is a power of two are, in fact, members of the family of Morton-like layouts. In order to sketch an informal argument for this, we recall that the indexing function for an n-dimensional canonical layout given array sizes $N_0, \ldots, N_{n-1}$ is defined as in Equation 2. If we assume that all sizes are powers of two, then the product of these sizes is guaranteed to be itself a power of two. Because multiplication by powers of two can be interpreted as a left-ward shift, the canonical layouts shift each input index $x_0, \ldots, x_n$ to a specific location in the binary expansion of the output index. Furthermore, because we assume $\forall i : x_i < N_i$, each bit in the output is determined by exactly one of the input indices; this allows us to interpret the summation as a series of bit-wise disjunctions, exactly like the definition of our Morton-like layouts. In general, a mode-0-major canonical layout of a $2^{b_0} \times \ldots \times 2^{b_{n-1}}$ array can be characterized—in the the scheme defined in Section 3.1—by contiguous subsequences of bits, each drawn from the same index i.e., a sequence of the following form:

$$[\underbrace{0, \ldots, 0}_{b_0 \text{ times}}, \underbrace{1, \ldots, 1}_{b_1 \text{ times}}, \ldots, \underbrace{n-1, \ldots, n-1}_{b_{n-1} \text{ times}}] \tag{6}$$

Canonical layouts with different major axes can be constructed by changing the order of the contiguous subsequences. The fact that the canonical layouts are members of the Morton-like family of array layouts allows us to evaluate the performance of these layouts in the exact same framework as the rest of the Morton-like layouts, and we will exploit this in Section 5.

3.3 Hardware-Accelerated Indexing

It is tempting to extend the aforementioned ideas to even more exotic indexing functions, like the Hilbert array layout [6, 32, 67]. The computational cost of many such functions renders them impractical, however: if the cost of computing memory addresses is too large, any performance gained by improving the cache-friendliness of a program will be negated. The Morton-like layouts we consider in this work allow efficient index calculations on modern commodity hardware, which we demonstrate in this section.

Under canonical array layouts, indices are calculated either iteratively through repeated addition and multiplication, or in parallel through parallel multiplication followed by reduction through addition. In n-dimensional cases both approaches require $n-1$ additions and $n-1$ multiplications, operations which can be efficiently performed on virtually all processors. Specifically, the Intel Haswell and AMD Zen 3 microarchitectures—on which we focus in this work—can perform 64-bit register addition (`ADD r64 r64`) with a latency 1 cycle and a reciprocal throughput of 0.25 cycles, while they can execute multiplication (`IMUL r64 r64`) with a latency of 3 cycles and a reciprocal throughput of 1 cycle [1].

Our bit-interleaving array layouts rely, in n-dimensional cases, on $n-1$ bitwise disjunctions and n bit-scatter operations. Such disjunctions (`OR r64 r64`) can be performed with a latency of 1 cycle and a reciprocal throughput of 0.25 cycles—the same as the `ADD` instruction—on both of the aforementioned microarchitectures. We perform the bit-scattering operation using the *parallel bit deposition* (`PDEP r64 r64 r64`) instruction, which is included in the BMI2 extension to the x86-64 instruction set [4]. The Intel Haswell and AMD Zen 3 microarchitectures both perform bit deposition with a latency of 3 cycles and a reciprocal throughput of 1 cycle, identical to the `IMUL` instruction. It follows that Morton-like indexing requires—in theory—only a single additional instruction over canonical index calculation.

The hardware extension required to perform bit deposition is widely supported: BMI2 has been included in Intel processors starting with the Haswell microarchitecture (2013) [29], and in AMD processors starting with the Excavator microarchitecture (2015), albeit in a limited fashion; AMD processors gained full hardware support for these instructions starting with the Zen 3 microarchitecture (2020) [22][2].

In order to further evaluate the competitiveness of Morton-like layouts compared to canonical layouts, we analyze implementations of both indexing schemes over a range of dimensionalities as compiled by GCC 12.3 and CLANG 15.0 using OSACA 0.5.2 [43]. All code was compiled using the `-O2` optimization flag. The results of this analysis are shown in Figure 3. Over the range of dimensionalities considered, the canonical layouts are consistently faster i.e., require fewer cycles to compute, than the Morton-like layouts. However, the difference in performance—approximately one cycle—is relatively small and overshadowed by the number of cycled saved due to a reduction in cache misses. Furthermore, we focus primarily on memory-bound applications, in which a small increase in index calculation time is unlikely to affect performance. We conclude, therefore, that Morton-like layouts are competitive with canonical layouts strictly in terms of address computation costs.

[2]Pre-Zen 3 processors supported parts of the BMI2 instruction set—the `PEXT` and `PDEP` instructions in particular—through emulation in microcode rather than in hardware, making them very slow.

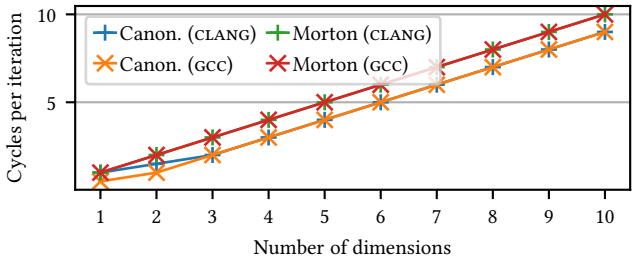


Figure 3: Throughput of a kernel calculating array indices using canonical layouts as well as Morton-like layouts on the Intel Haswell microarchitecture as given by OSACA.

3.4 Support for SIMD

An important consideration in the design of array layouts is the ability to vectorize kernels through single-instruction multiple-data (SIMD) operations. Canonical layouts guarantee the contiguity of fibers in the array, which facilitates the (automated) vectorization (e.g., the application of SIMD) of many operations, and this benefit is lost when applying the array layouts discussed in this paper. However, we posit that there remains ample opportunity to accelerate computation on Morton-like arrays using SIMD, and we argue this by distinguishing two classes of computation patterns.

The first class consists of *unstructured* patterns in which data is operated on element-wise without spatial context i.e., without consideration of nearby elements; a prominent example of such an operation is matrix *addition*. In such applications, SIMD can be trivially applied to the underlying one-dimensional memory, regardless of the layout of the data: since elements can be added point-wise in any order, doing so in the order in which the data is laid out in memory is both feasible and enables SIMD.

The second class of problems consists of *structured* patterns in which operations must be performed in a specific order. A prime example of such an operation is matrix *multiplication* where the inner product of fibers must be computed. In such cases, it is imperative that fibers can be accessed in contiguous blocks. The size of these blocks depends on the vectorization technology used as well as the size of the data type: in the x86 instruction set, SSE vectorisation requires two consecutive double-precision numbers or four consecutive single-precision numbers [38]; the much wider ARM SVE instruction set extension [58] may require up to thirty-two consecutive double-precision numbers or sixty-four single-precision numbers.

In order to facilitate vectorization for structured patterns of computation, we can impose certain constraints on the array layouts we consider. Indeed, if the n least-significant bits of an interleaving pattern are all drawn from the mth input index, then the layout guarantees that the mode-m fibers in the array are contiguous in blocks of 2^n elements. This requirement can be incorporated into the selection of array layouts; for example, we can enable efficient AVX2 vectorisation (with a vector width of 256 bits) using single-precision (32-bit) floating point numbers by ensuring that the three

least significant bits in an array layout are drawn from the same source. In other words, we can easily constrain our search space to include only array layouts with properties that favor vectorization, and we believe that doing so will enable SIMD-accelerated computation arrays laid out in Morton-like orders.

4 EXPLORATION THROUGH EVOLUTION

The canonical set of indexing bijections for laying out multi-dimensional memory is small: for two-dimensional data, there are two possible layouts, and the performance of these layouts can be evaluated using exhaustive benchmarks [23, 59, 62]. Exhaustively exploring the family of indexing function outlined in Section 3, however, is impractical owing to the sheer number of permissible permutations. Importantly, the number of canonical layouts increases only with the number of *dimensions*, while the number of Morton-like layouts increases with both the number of dimensions and the *size* of the array in each of those dimensions. By Equation 5, a small 4×4 array (indexed by two bits in each dimension) can be laid out in $^{(2+2)!}/_{2!2!} = 6$ ways. A larger array of size 4096×4096 (twelve bits in each dimension) can be laid out in $^{(12+12)!}/_{12!12!} = 2\,704\,156$ ways. A three-dimensional array of size $256 \times 256 \times 256$ has the same number of elements as the aforementioned 4096×4096 array, but permits $^{(8+8+8)!}/_{8!8!8!} = 9\,465\,511\,770$ permutations. As these examples indicate, the number of possible permutations quickly scales beyond what can be feasibly explored through exhaustive search; in order to tackle the explosive growth in the design space for Morton-like layouts, we propose the use of genetic algorithms (Section 2.3).

4.1 Genetic Algorithm Configuration

In this work, we employ a relatively simple (λ, μ)-ES genetic algorithm [34, 56]. The chromosomal representations of array layouts is identical to the characterization given in Section 3.1, and this gives rise to a combinatorial optimization problem. We facilitate the recombination of array layouts into novel layouts using the ordered crossover (OX) operator [18], and we employ inversion-based mutation [21]. Our approach differs from classical genetic algorithms in only one significant way: our initial population is not chosen randomly from the solution space. Instead, the initial populations for our evolutionary experiments always consist of two individuals, depicting two canonical layouts for a given array size, as described in Section 3.2. We choose to do this to ensure that our initial populations are unbiased and deterministic, allowing us to more easily assess the efficacy of our genetic strategy.

4.2 Fitness Function Design

There are two general strategies for evaluating the performance i.e., fitness, of a given array layout under a given cache hierarchy and access pattern: measurement and simulation. In order to assess fitness through *measurement*, we execute a program on actual hardware and measuring the running time of the process. Although such a fitness function is conceptually simple, it suffers from two primary flaws: (1) measurements are noisy and may suffer from run-to-run variance, which may hinder the performance of genetic algorithms [45]—in particular, our genetic algorithm is vulnerable to noise stemming from cache pollution effects; and (2) measurements require access to the target hardware, which may be inconvenient or even impossible—for example, in hardware-software co-design scenarios, where the hardware under consideration does not (yet) exist. For these reasons, we choose not to base our fitness function on measurements.

Instead, we employ *simulation* for which we need a simulator that can accurately compare the cache performance for different access-patterns on the same cache hierarchy. For this, we selected PYCACHESIM, a component of the KERNCRAFT toolkit [30]. We use PYCACHESIM by simulating an access pattern such as matrix multiplication and registering the relevant trace of load and store operations. After all accesses have been recorded, we force a write-back of the caches and collect the number of hits and misses in each cache level. We combine the number of hits in every cache level as well as in main memory with the latency of retrieving data from each of these levels to compute the total number of cycles spent retrieving data from the cache hierarchy. Given an array layout I, an access pattern A and a simulated cache hierarchy H, we calculate the total number of cycles using the following equation, in which $\mathrm{L}i_{\mathrm{hit}}$, $\mathrm{L}i_{\mathrm{miss}}$, and $\mathrm{L}i_{\mathrm{lat}}$ represent the number of hits, the number of misses, and the latency of the ith cache level, and M represents main memory:

$$C(I;A,H) = \mathrm{M}_{\mathrm{hit}}(I;A,H)\mathrm{M}_{\mathrm{lat}}(H) + \sum_i \mathrm{L}i_{\mathrm{hit}}(I;A,H)\mathrm{L}i_{\mathrm{lat}}(H) \quad (7)$$

From this, we compute an approximation of the number of accesses performed per cycle, giving rise to a higher-is-better fitness function defined as follows:

$$F(I;A,H) = \frac{\mathrm{L1}_{\mathrm{hit}}(I;A,H) + \mathrm{L1}_{\mathrm{miss}}(I;A,H)}{\mathrm{L1}_{\mathrm{lat}}(H) \cdot C(I;A,H)} \quad (8)$$

Intuitively, the numerator in Equation 8 counts the total number of memory accesses, as all accesses either hit or miss in L1. The denominator, then, estimates the total number of cycles spent retrieving data from the various cache levels. The denominator is multiplied by a normalizing factor equal to the latency of the L1 cache; it follows from Equation 7 that the achievable performance is softly bound by the reciprocal of the L1 access latency. Indeed, this performance is achieved if and only if all accesses hit the L1 cache. Normalizing the fitness function using the L1 cache latency improves our ability to compare results between different cache hierarchies.

5 EVALUATION

We evaluate the efficacy of the methods hitherto discussed by demonstrating that (1) our fitness function is well-chosen i.e., that is correlates with performance measurements in real hardware; that (2) our evolutionary process is capable of finding novel array layouts with favorable cache properties; and that (3) the layouts which are found by our evolutionary process actually lead to relevant performance gains in real hardware. Our validation is based on eight distinct access patterns and two processors with distinct cache hierarchies.

5.1 Experimental Setup

We consider a set of eight access patterns loosely based on the selection of algorithms used by Thiyagalingam et al. [62]. The access

```cpp
template <concepts::array<2> M>
void mm_ijk(const M & A, const M & B, M & C) {
    const auto m = C.get_size();
    for (std::size_t i = 0; i < m; ++i) {
        for (std::size_t j = 0; j < m; ++j) {
            typename M::value_type acc = 0.;
            for (std::size_t k = 0; k < m; ++k)
                acc += A.load(i, k) * B.load(k, j);
            C.store(acc, i, j);
        }
    }
}
```

Listing 1: Example of how an access pattern (MMijk) is described in C++. Metaprogramming allows the same source to be used for both simulation and execution on real hardware.

```yaml
caches:
  L1:
    sets: 64
    ways: 8
    line: 64
    replacement: LRU
    write_back: true
    store_to: L2
    load_from: L2
    latency: 4
  L2:
    sets: 512
    ways: 8
    line: 64
    replacement: LRU
    write_back: true
    store_to: L3
    load_from: L3
    victim_to: L3
    latency: 12
  L3:
    sets: 25600
    ways: 16
    line: 64
    replacement: LRU
    write_back: true
    latency: 36
memory:
  first: L1
  last: L3
  latency: 200
```

(a) Intel Xeon E5-2660 v3

```yaml
caches:
  L1:
    sets: 64
    ways: 8
    line: 64
    replacement: LRU
    write_back: true
    store_to: L2
    load_from: L2
    latency: 7
  L2:
    sets: 1024
    ways: 8
    line: 64
    replacement: LRU
    write_back: true
    store_to: L3
    load_from: L3
    victim_to: L3
    latency: 12
  L3:
    sets: 32768
    ways: 16
    line: 64
    replacement: LRU
    write_back: true
    latency: 46
memory:
  first: L1
  last: L3
  latency: 200
```

(b) AMD EPYC 7413

Listing 2: Two examples of cache specifications for different CPU models. Note that these configurations are approximations of the true cache hierarchies.

patterns were picked to represent common real-world applications (dense linear algebra and fluid dynamics), to represent both two-dimensional and three-dimensional applications, and to differ in critical properties such as memory size and number of loads and stores. A description of the access patterns we consider in this paper is given in Table 1.

All our access patterns are described using C++ code—see the example in Listing 1—which ensures high performance as opposed to the Python code used for our evolutionary processes; the interaction between the C++ and Python components of our project is managed using pybind11 [39]. We use template meta-programming to generalize our access patterns in such a way that a single definition can be used for both simulation and benchmarking without loss of performance due to run-time polymorphism; this eliminates any possible discrepancies between the code used for simulation and the code used for measurement.

We conduct our experiments on two different CPUs: the Intel Xeon E5-2660 v3 [37] based on the Haswell microarchitecture [29], and the AMD EPYC 7413 [2] based on the Zen 3 microarchitecture [22]. When we perform experiments on non-simulated Haswell processors we use the the DAS-6 cluster [11], whereas we use a machine located at CERN for experiments on Zen 3 processors. When we perform experiments based on simulation, we use the the DAS-6 cluster [11] and configure our cache simulator according the cache configurations shown in Listing 2a for the Haswell processor, and Listing 2b for the Zen 3 processor. Note that the cache configurations are based on the accessibility of caches from a single core. This is especially relevant for the L3 cache on the Zen 3 chip, which is shared across groups of cores rather than the entire CPU: in the case of the AMD EPYC 7413, the CPU comes equipped with 128 MiB of L3 cache, but only 32 MiB is accessible from any single core [22]. We simplify the cache replacement policies of the actual hardware by assuming LRU caches (i.e., caches with a least-recently-used eviction policy); in reality, the Haswell caches employ eviction policies consistent with tree-PLRU (tree-based pseudo-LRU) for the L1 and L2 caches [1, 63], while the L3 cache is consistent with a set-dueling-controlled adaptive insertion policy [1, 53]. Cache sizes were gathered from specification documents [3, 29], while cache latencies were obtained optimistically from sources on the fastest load-to-use latencies [3, 17]. The Zen 3 L1 cache has a fastest load-to-use latency of four cycles for integers and seven cycles for floating point values [3]—we use the latter in our simulations. Finally, we assume a constant 200 cycle access latency for main memory in both systems.

5.2 Fitness Function Validation

The fitness function we use in our evolutionary process (Section 4.2) is based on simulation results because simulation yields significant benefits over empirical measurements, primarily in terms of determinism and in the ability to simulate future hardware. However, this strategy is not without risk: the simulation we perform is based on a non-cycle-accurate simulator, uses simplified cache hierarchies, and ignores computation entirely. Consequently, we must evaluate the usefulness of our fitness function by establishing its correlation with execution time in real hardware.

Ideally, the running time of a kernel using a given array layout would correlate inversely linearly with our fitness function, therefore ensuring two important properties. Firstly and most importantly, it guarantees that running time decreases monotonically with the value of the fitness function, such that an array layout with a higher fitness value is guaranteed to run more quickly; this allows us to establish a ranking of layouts and enables us to reliably select the best-performing array layout. Secondly, linear correlation guarantees proportionality between fitness and running time, which facilitates the weighted selection of individuals.

To evaluate the degree to which the aforementioned criteria are met, we randomly select one hundred array layouts for each of the eight access patterns given in Table 1. We then evaluate the simulated fitness and measure the running time in real hardware of each pair of array layout and access pattern. The fitness functions of the pairs are calculated in parallel, as they are designed to be deterministic and impervious to cache pollution or resource contention.

Table 1: Overview of the access patterns used for evaluation, including the use of memory and the number of loads and stores.

Access pattern	Description	Mem. size	Loads	Stores
MMijk$(m;s)$	Multiplication of two $2^m \times 2^m$ matrices, both of s-byte real numbers.	$3 \cdot s \cdot 2^{2m}$ B	$2 \cdot 2^{3m}$	2^{2m}
MMTijk$(m,n;s)$	Multiplication of a $2^m \times 2^n$ matrix by a transposed $2^m \times 2^n$ matrix.	$s \cdot (2 \cdot 2^{m+n} + 2^{2n})$B	$2 \cdot 2^{2m+n}$	2^{2m}
MMikj$(m;s)$	Same as MMijk$(m;s)$ with the order of the inner loops switched.	$3 \cdot s \cdot 2^{2m}$ B	$3 \cdot 2^{3m}$	2^{3m}
MMTikj$(m,n;s)$	Same as MMTijk$(m,n;s)$ with the order of the inner loops switched.	$s \cdot (2 \cdot 2^{m+n} + 2^{2n})$B	$3 \cdot 2^{2m+n}$	2^{2m+n}
Jacobi2D$(m,n;s)$	Four-point stencil kernel over a $2^m \times 2^n$ array of s-byte real numbers.	$2 \cdot s \cdot 2^{m+n}$ B	$\sim 4 \cdot 2^{m+n}$	2^{m+n}
Cholesky$(m;s)$	Cholesky–Banachiewicz decomposition of a $2^m \times 2^m$ matrix.	$2 \cdot s \cdot 2^{2m}$ B	$2 \cdot 2^{2m}$	$\sim \frac{1}{2} \cdot 2^{2m}$
Crout$(m;s)$	Crout decomposition of a $2^m \times 2^m$ matrix of s-byte real numbers.	$2 \cdot s \cdot 2^{2m}$ B	$\frac{7}{2} \cdot 2^{2m}$	2^{2m}
Himeno$(m,n,p;s)$	Nineteen-point Himeno stencil [33] over $2^m \times 2^n \times 2^p$ arrays.	$12 \cdot s \cdot 2^{m+n+p}$ B	$24 \cdot 2^{m+n+p}$	2^{m+n+p}

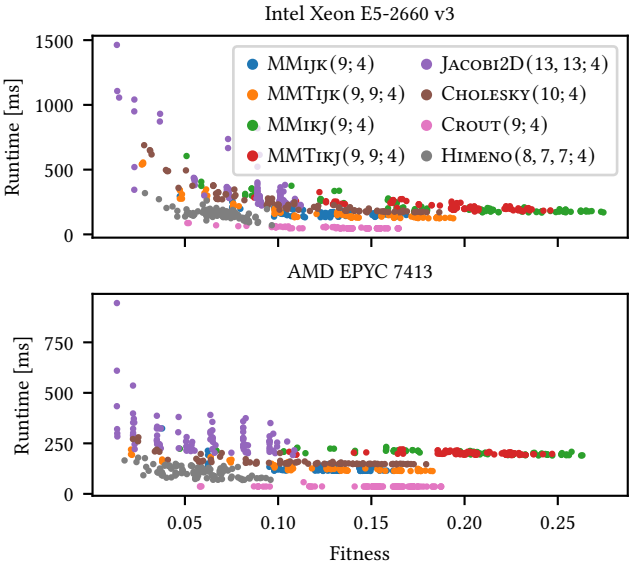


Figure 4: Scatter plot of the fitness and measured running time on an Intel Xeon E5-2660 v3 CPU and AMD EPYC 7413 for randomly chosen array layouts.

The empirical benchmarks are performed sequentially, ensuring that the benchmark is the sole user of the processor caches. All measurements are repeated ten times, and we report the mean and standard deviation of the running time.

The results of this experiment are shown in Figure 4. The coefficient of variation of the measurements never exceeded a value of $c_v = 0.0801$. Accordingly, we have opted to omit error bars from the figure. Upon visual inspection, it is clear that the correlation between our fitness function and running time is not linear, although the two do appear correlated. We confirm our suspicions of correlation by computing Pearson's coefficient of correlation (ρ_p) and Spearman's coefficient of rank correlation (ρ_s); the resulting statistics are given in Table 2. We observe that our fitness function and running time correlate moderately to strongly with running time for the Intel Xeon E5-2660 v3 processor, although the correlation is weaker for the AMD EPYC 7413 processor. Although it is clear that there is space for the fitness function to be improved, we believe that it correlates sufficiently with running time to enable its use in genetic algorithms.

Table 2: Pearson's coefficient of correlation (ρ_p) and Spearman's coefficient of rank correlation (ρ_s) between our simulation-based fitness function and true running time.

	Intel E5-2660 v3		AMD EPYC 7413	
Access pattern	ρ_p	ρ_s	ρ_p	ρ_s
MMijk$(9;4)$	−0.672	−0.480	−0.648	−0.489
MMTijk$(9,9;4)$	−0.810	−0.896	−0.863	−0.823
MMikj$(9;4)$	−0.845	−0.815	−0.800	−0.838
MMTikj$(9,9;4)$	−0.777	−0.744	−0.291	−0.405
Jacobi2D$(13,13;4)$	−0.760	−0.769	−0.390	−0.428
Cholesky$(10;4)$	−0.827	−0.953	−0.725	−0.892
Crout$(9;4)$	−0.846	−0.663	−0.213	−0.704
Himeno$(8,7,7;4)$	−0.607	−0.475	−0.561	−0.496

5.3 Genetic Algorithm Performance

To evaluate our evolutionary process (Section 4) as a whole, we intend to verify that it can, indeed, find Morton-like array layouts that have a higher simulated fitness than the canonical layouts. To this end, we perform the evolutionary process for each combination of our two simulated processors and eight access patterns, giving rise to a total of sixteen experiments. For all of these experiments, we configure our genetic algorithm to use $\mu = 20$, $\lambda = 20$, and a mutation rate of 25%. We simulate a total of 20 generations in each case.

Figure 5 shows a violin plot of the fitness distribution of all individuals considered during the evolutionary process. Figure 6 shows the evolution of population fitness over the course of our experiments. Note that each of these experiments represents a single evolutionary process. We notice that for the MMTijk, MMikj, Jacobi2D, and Himeno access patterns, our method does not manage to discover any layouts with higher fitness than the initial population of canonical layouts. In the experiment on the MMijk access pattern, we discover layouts with a fitness 149.8% higher than the canonical layouts on the Intel Xeon E5-2660 v3 processor, and we improve on the fitness of canonical layouts by 187.5% for the AMD EPYC 7413. We also find layouts with improved fitness for the MMTikj (109.6% and 141.1% for the Intel and AMD processors, respectively), Cholesky (26.4% and 36.8%), and Crout (545.9% and 541.1%) access patterns. It is notable that we are able to find layouts with high fitness in few generations.

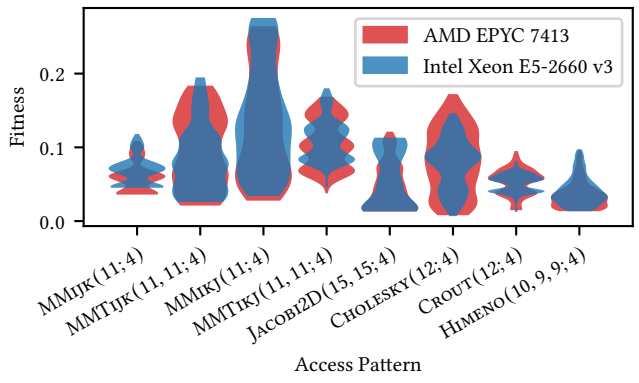


Figure 5: Distribution of the fitness values for all individuals found across evolution experiments for eight access patterns and two processors.

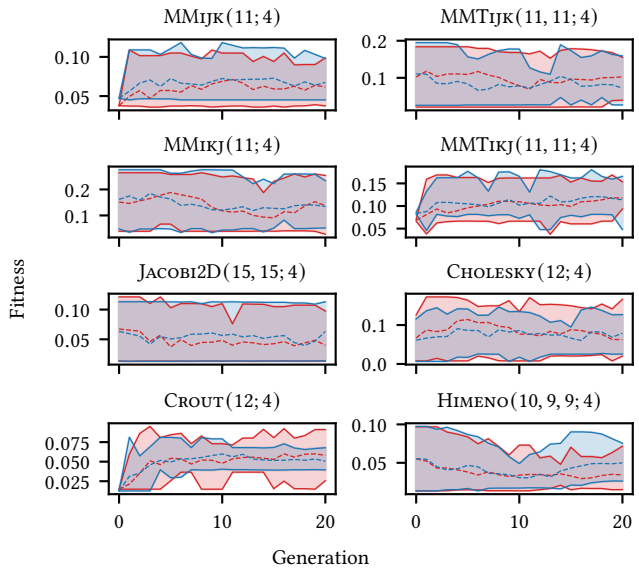


Figure 6: Range of fitness values across eight experiments for the Intel Xeon E5-2660 v3 (blue) and AMD EPYC 7413 (red). Mean fitness values are given by the dashed lines.

5.4 Real-World Performance

In order to evaluate whether the layouts identified by our evolutionary algorithms as superior to canonical layouts are indeed better, we evaluate them on real hardware. We collect the fittest individual from each of the successful evolution experiments—i.e., experiments in which our method improved upon canonical layouts, as indicated by the top boundary in Figure 6 exceeding the maximum fitness in the first generation—and evaluate the performance of those layouts compared to the canonical layouts on real hardware. Given that our genetic algorithm discovered superior layouts for four access patters—MMijk, MMTikj, Cholesky, and Crout–and that we evaluate a discovered layout and two canonical layouts for each access pattern, this gives rise to twenty-four experiments.

Table 3: Comparison of running time between the best-performing canonical layout and the best-performing layout found by our evolutionary process for four access patterns.

Access pattern	Best can.	Best evo.	Speedup
Intel Xeon E5-2660 v3			
MMijk(11; 4)	17.84 s	10.94 s	63.1%
MMTikj(11, 11; 4)	18.13 s	13.96 s	29.9%
Cholesky(12; 4)	11.84 s	11.43 s	3.6%
Crout(12; 4)	158.54 s	43.72 s	262.6%
AMD EPYC 7413			
MMijk(11; 4)	37.71 s	9.58 s	293.8%
MMTikj(11, 11; 4)	32.35 s	15.21 s	112.6%
Cholesky(12; 4)	9.72 s	9.55 s	1.0%
Crout(12; 4)	232.84 s	21.03 s	1007.0%

We repeat each experiment ten times to compensate for run-to-run variance.

The results of our experiments are shown in Table 3; they show that some access patterns—the Cholesky pattern in particular—benefit very little from our method, with speed-ups ranging from small on the Haswell processor to insignificant on the Zen 3 processor. The matrix multiplication access patterns benefit more, and performance for these access patterns is improved significantly. The Crout access pattern stands out as achieving very large speedup—up to a factor ten—from our method. It is worth noting that, in most cases, the Zen 3 processor benefits more from our evolutionary methodology than the Haswell processor; we do not currently have a satisfactory explanation for this behavior.

It is important to note that we do not claim to have discovered a novel way of performing matrix multiplication or matrix decomposition that outperforms existing implementations. Indeed, our experiments are based on relatively naive implementations of these algorithms; high-performance implementations of matrix multiplication commonly rely on tiling to significantly improve the cache behavior of the application [48], and the performance of tiled matrix multiplication surpasses what we achieve in this paper. The purpose of the methodology described in this paper, rather, is to provide an alternative way of improving the cache behavior of an application in a manner which is fully agnostic of the application: unlike tiling and other application-specific optimizations, our methodology of altering the array layouts can be applied to any multi-dimensional problem without the need for application-specific knowledge. In addition, our approach requires few code changes, making it easy to implement.

6 LIMITATIONS AND THREATS TO VALIDITY

Throughout this work, we evaluate cache efficacy through a simplified lens which may reduce the applicability of our methods in more complex, real-world applications. Indeed, we consider accesses to memory in isolation, decoupled from computation and cache-polluting effects. We assume single-threaded execution without scheduling, which means that our caches will not be polluted by processes sharing (parts of) the cache hierarchy, nor will the application have its cached data evicted due to context switching. We

also assume scalar, in-order execution of memory accesses. Finally, we take an optimistic view of cache latencies, using the fastest load-to-use latencies provided by hardware manufacturers; in real-world scenarios, cache latencies may be both more pessimistic and less stable than we assume. The results shown in Section 5.4 indicate, however, that our fitness function is sufficiently accurate to be effective in real hardware.

In addition, the family of array layouts described in this work requires array sizes to be powers of two in each dimension. In applications where this is not the case, arrays must be over-allocated. For n-dimensional applications, using the layouts described in this paper requires over-allocation by a factor of $O(2^n)$. Furthermore, applications using such layouts must consider the use of SIMD vectorization: it remains an open question which operations on arrays laid out in non-standard ways can be (automatically) vectorized. We have argued for the feasibility of SIMD in Morton-like arrays in Section 3.4.

Finally, our work considers only multiset permutations, in which the rank significance of bits in the input indices is preserved. This decision is based on current commodity hardware, which is capable of efficiently permuting bits only under this condition. There exists an even larger family of layouts in which rank bit significance is not preserved[3]; such layouts could be of practical use in theoretical future processors with more advanced bit manipulation instructions, or in current FPGA and ASIC devices which permit the implementation of custom bit manipulation operations. Although we have not tested our approach on this further generalization, we are confident that an evolutionary approach like the one presented in this paper could be beneficial in exploring this (even larger) design space.

7 REPRODUCIBILITY AND REUSABILITY

The evolutionary algorithms, scripts for the processing and visualisation of data, and other software used in this paper are permanently archived on Zenodo [60], and have been made available at doi:10.5281/zenodo.10567243. The aforementioned artifact also contains all data that was gathered and processed during the work presented in this paper. For more information about the use of the artifact accompanying this paper, see the included `README` file.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have discussed a generalization of the Morton layout for multi-dimensional data and we have shown that there exist families of array layouts with strongly varying cache behavior which, in turn, impact the performance of applications. We have shown how these layouts can be systematically described, and that the number of possible layouts quickly exceed the limits of what can be feasibly explored using exhaustive search. We have proposed a method based on evolutionary algorithms for the exploration of the design space of such layouts. We have evaluated the fitness of different array layouts using cache simulation and we have presented results indicating that our fitness function correlates with real world performance. Furthermore, we have shown that the methodology described in this paper can be used to improve the performance of applications on real hardware by up to ten times.

In the future, we intend to investigate the use of multi-objective optimization using NSGA-II [19] in order to find array layouts that provide favorable cache behavior across multiple applications. We also intend to explore more advanced genetic algorithms which are known to perform well in combinatorial problems, such as RKGA [12] and BRKGA [26]. It is our belief that exploring more evolutionary strategies will give us more insight into the convergence properties of various methods, and allow us to select the most efficient one. Although our fitness function correlates with real-world performance, the correlation is not perfect; we believe that the efficacy of our method could be improved through the development of more advanced fitness function, perhaps through the use of machine learning methods. In particular, we believe that the field of metric learning may enable us to develop more accurate fitness functions, and we aim to explore this avenue of research in the future. Finally, we aim to expand our research to a broader range of access patterns and hardware, including graphics processing units (GPUs).

[3]That is to say, the layout $[0_0, 0_1, 1_0, 1_1]$ (which draws its least significant bit from the least significant bit of the first index) is distinct from the layout $[0_1, 0_0, 1_0, 1_1]$ (which instead draws its least significant bit from the *second-least* significant bit of the first index).

ACKNOWLEDGMENTS

The work presented in this paper was done in the context of the CERN Doctoral Student Programme. Many of the experimental results shown in this paper were gathered on the Advanced School for Computing and Imaging (ASCI) DAS-6 compute cluster [11].

REFERENCES

[1] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, Rhode Island, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, New York, USA, 673–686. https://doi.org/10.1145/3297858.3304062

[2] Advanced Micro Devices, Inc. 2014. *AMD EPYC 7413*. Advanced Micro Devices, Inc. https://www.amd.com/en/products/cpu/amd-epyc-7413

[3] Advanced Micro Devices, Inc. 2020. Software optimization guide for the AMD family 19h processors.

[4] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E. Porter. 2019. x86-64 Instruction Usage among C/C++ Applications. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, New York, USA, 68–79. https://doi.org/10.1145/3319647.3325833

[5] Ibrahim Al-Kharusi and David W. Walker. 2019. Locality properties of 3D data orderings with application to parallel molecular dynamics simulations. *The International Journal of High Performance Computing Applications* 33, 5 (2019), 998–1018. https://doi.org/10.1177/1094342019846282

[6] João Nuno Ferreira Alves, Luís Manuel Silveira Russo, and Alexandre Francisco. 2022. Cache-Oblivious Hilbert Curve-Based Blocking Scheme for Matrix Transposition. *ACM Trans. Math. Softw.* 48, 4, Article 37 (dec 2022), 28 pages. https://doi.org/10.1145/3555353

[7] Edward J. Anderson and Michael C. Ferris. 1994. Genetic Algorithms for Combinatorial Optimization: The Assemble Line Balancing Problem. *ORSA Journal on Computing* 6, 2 (1994), 161–173. https://doi.org/10.1287/ijoc.6.2.161

[8] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[9] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A View of the Parallel Computing Landscape. *Commun. ACM* 52, 10 (oct 2009), 56–67. https://doi.org/10.1145/1562764.1562783

[10] Michael Bader. 2012. *Space-filling curves: an introduction with applications in scientific computing*. Vol. 9. Springer Science & Business Media, Berlin, Heidelberg, Germany.

[11] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* 49, 05 (5 2016), 54–63. https://doi.org/10.1109/MC.2016.127

[12] James Carl Bean. 1994. Genetic Algorithms and Random Keys for Sequencing and Optimization. *ORSA Journal on Computing* 6, 2 (1994), 154–160. https://doi.org/10.1287/ijoc.6.2.154

[13] Richard A. Brualdi. 1977. *Introductory combinatorics* (5 ed.). Pearson Education, London, United Kingdom.

[14] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. 1999. Nonlinear Array Layouts for Hierarchical Memory Systems. In *Proceedings of the 13th International Conference on Supercomputing* (Rhodes, Greece) *(ICS '99)*. Association for Computing Machinery, New York, New York, USA, 444–453. https://doi.org/10.1145/305138.305231

[15] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. 1999. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures* (Saint Malo, France) *(SPAA '99)*. Association for Computing Machinery, New York, New York, USA, 222–231. https://doi.org/10.1145/305619.305645

[16] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. 2011. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) *(SC '11)*. Association for Computing Machinery, New York, New York, USA, Article 13, 11 pages. https://doi.org/10.1145/2063384.2063401

[17] Vali Codreanu, Joerg Hertzer, Cristian Morales, Jorge Rodriguez, Ole Widar Saastad, and Martin Stachon. 2017. Best practice guide Haswell / Broadwell.

[18] Lawrence Davis. 1985. Applying Adaptive Algorithms to Epistatic Domains. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1* (Los Angeles, California, USA) *(IJCAI'85)*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 162–164.

[19] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and Thirunavukarasu Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. https://doi.org/10.1109/4235.996017

[20] Daryl Deford and Ananth Kalyanaraman. 2013. Empirical Analysis of Space-Filling Curves for Scientific Computing Applications. In *2013 42nd International Conference on Parallel Processing* (Lyon, France). IEEE, New York, New York, USA, 170–179. https://doi.org/10.1109/ICPP.2013.26

[21] Agoston E. Eiben and Jim E. Smith. 2015. *Representation, Mutation, and Recombination*. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 49–78. https://doi.org/10.1007/978-3-662-44874-8_4

[22] Mark Evers, Leslie Barnes, and Mike Clark. 2022. The AMD Next-Generation "Zen 3" Core. *IEEE Micro* 42, 3 (2022), 7–12. https://doi.org/10.1109/MM.2022.3152788

[23] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science* (New York, New York, USA) *(FOCS '99)*. IEEE, New York, New York, USA, 285–297. https://doi.org/10.1109/SFFCS.1999.814600

[24] Steven T. Gabriel and David S. Wise. 2004. The Opie Compiler from Row-Major Source to Morton-Ordered Matrices. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture* (Munich, Germany) *(WMPI '04)*. Association for Computing Machinery, New York, New York, USA, 136–144. https://doi.org/10.1145/1054943.1054962

[25] Andrew Gartland-Jones and Peter Copley. 2003. The Suitability of Genetic Algorithms for Musical Composition. *Contemporary Music Review* 22, 3 (2003), 43–55. https://doi.org/10.1080/0749446032000150870

[26] José Fernando Gonçalves and Mauricio G. C. Resende. 2011. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics* 17, 5 (01 Oct 2011), 487–525. https://doi.org/10.1007/s10732-010-9143-1

[27] Peter Gottschling, David S. Wise, and Michael D. Adams. 2007. Representation-Transparent Matrix Algorithms with Scalable Performance. In *Proceedings of the 21st Annual International Conference on Supercomputing* (Seattle, Washington, USA) *(ICS '07)*. Association for Computing Machinery, New York, New York, USA, 116–125. https://doi.org/10.1145/1274971.1274989

[28] Peter Gottschling, David S. Wise, and Adwait Joshi. 2009. Generic support of algorithmic and structural recursion for scientific computing. *International Journal of Parallel, Emergent and Distributed Systems* 24, 6 (2009), 479–503. https://doi.org/10.1080/17445760902758560

[29] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted Burton. 2014. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro* 34, 2 (2014), 6–20. https://doi.org/10.1109/MM.2014.10

[30] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2017. Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels. In *Tools for High Performance Computing 2016* (Stuttgart, Germany), Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel (Eds.). Springer International Publishing, Cham, Switzerland, 1–22. https://doi.org/10.1007/978-3-319-56702-0_1

[31] Brian Hegerty, Chih-Cheng Hung, and Kristen Kasprak. 2009. A comparative study on differential evolution and genetic algorithms for some combinatorial problems. In *Proceedings of the 8th Mexican International Conference on Artificial Intelligence* (Guanajuato, Mexico), Vol. 9. Springer Verlag, Berlin, Heidelberg, Germany, 13.

[32] David Hilbert. 1891. Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.* 38, 3 (1891), 459–460. https://doi.org/10.1007/BF01199431

[33] Ryutaro Himeno. 2001. The Riken Himeno CFD Benchmark. https://i.riken.jp/en/supercom/documents/himenobmt/

[34] John Henry Holland. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, Cambridge, Massachusetts, USA.

[35] John Henry Holland. 1992. Genetic algorithms. *Scientific American* 267, 1 (1992), 66–73.

[36] OEIS Foundation Inc. 2023. Moser–de Bruijn sequence, entry A000695 in The On-Line Encyclopedia of Integer Sequences. https://oeis.org/A000695

[37] Intel Corporation. 2014. *Intel Xeon Processor E5-2660 v3*. Intel Corporation. https://ark.intel.com/content/www/us/en/ark/products/81706/intel-xeon-processor-e52660-v3-25m-cache-2-60-ghz.html

[38] Intel Corporation. 2023. Intel 64 and IA-32 Architectures Software Developer's Manual.

[39] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11 – Seamless operability between C++11 and Python. https://github.com/pybind/pybind11

[40] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2011. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2011), 105–118. https://doi.org/10.1109/TPDS.2010.107

[41] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500. https://doi.org/10.1137/07070111X

[42] Markus Kowarschik and Christian Weiß. 2003. *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 213–232. https://doi.org/10.1007/3-540-36574-5_10

[43] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. 2018. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (Dallas, Texas, USA). IEEE, New York, New York, USA, 121–131. https://doi.org/10.1109/PMBS.2018.8641578

[44] John Mellor-Crummey, David Whalley, and Ken Kennedy. 2001. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *International Journal of Parallel Programming* 29, 3 (01 Jun 2001), 217–247. https://doi.org/10.1023/A:1011119519789

[45] Brad L. Miller and David E. Goldberg. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems* 9, 3 (1995), 193–212.

[46] Guy Macdonald Morton. 1966. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. Technical Report. International Business Machines Company.

[47] Heinz Mühlenbein. 1991. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Parallelism, Learning, Evolution* (Wildbad Kreuth, Germany). Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 398–406.

[48] Neungsoo Park, Bo Hong, and Viktor K. Prasanna. 2003. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems* 14, 7 (2003), 640–654. https://doi.org/10.1109/TPDS.2003.1214317

[49] G. Pavai and T. V. Geetha. 2016. A Survey on Crossover Operators. *ACM Comput. Surv.* 49, 4, Article 72 (dec 2016), 43 pages. https://doi.org/10.1145/3009966

[50] Filip Pawłowski, Bora Uçar, and Albert-Jan Nicholas Yzelman. 2019. A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations. *Journal of Computational Science* 33 (2019), 34–44. https://doi.org/10.1016/j.jocs.2019.02.007

[51] Martin Perdacher, Claudia Plant, and Christian Böhm. 2020. Improved Data Locality Using Morton-order Curve on the Example of LU Decomposition. In *2020 IEEE International Conference on Big Data (Big Data)* (Atlanta, Georgia, USA). IEEE, New York, New York, USA, 351–360. https://doi.org/10.1109/BigData50022.2020.9378385

[52] Andy D. Pimentel. 2017. Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration. *IEEE Design & Test* 34, 1 (2017), 77–90. https://doi.org/10.1109/MDAT.2016.2626445

[53] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2008. Set-Dueling-Controlled Adaptive Insertion for High-Performance Caching. *IEEE Micro* 28, 1 (2008), 91–98. https://doi.org/10.1109/MM.2008.14

[54] Dolly Sapra and Andy D. Pimentel. 2020. Constrained Evolutionary Piecemeal Training to Design Convolutional Neural Networks. In *Trends in Artificial Intelligence Theory and Applications. Artificial Intelligence Practices*. Springer International Publishing, Cham, Switzerland, 709–721.

[55] S.N. Sivanandam and S.N. Deepa. 2008. *Genetic Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 15–37. https://doi.org/10.1007/978-3-540-73190-0_2

[56] Adam Slowik and Halina Kwasnicka. 2020. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications* 32, 16 (01 Aug 2020), 12363–12379. https://doi.org/10.1007/s00521-020-04832-8

[57] M. Srinivas and Lalit M. Patnaik. 1994. Genetic algorithms: a survey. *Computer* 27, 6 (1994), 17–26. https://doi.org/10.1109/2.294849

[58] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (2017), 26–39. https://doi.org/10.1109/MM.2017.35

[59] Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger, and Attila Krasznahorkay. 2023. Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* (Coimbra, Portugal) *(ICPE '23)*. Association for Computing Machinery, New York, New York, USA, 55–66. https://doi.org/10.1145/3578244.3583723

[60] Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger, and Attila Krasznahorkay. 2024. *Artifact for "Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays"*. https://doi.org/10.5281/zenodo.10567243

[61] Lothar Terfloth and Johann Gasteiger. 2001. Neural networks and genetic algorithms in drug design. *Drug Discovery Today* 6 (2001), 102–108. https://doi.org/10.1016/S1359-6446(01)00173-8

[62] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly. 2006. Is Morton layout competitive for large two-dimensional arrays yet? *Concurrency and Computation: Practice and Experience* 18, 11 (2006), 1509–1539. https://doi.org/10.1002/cpe.1018

[63] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2020. CacheQuery: Learning Replacement Policies from Hardware Caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, United Kingdom) *(PLDI 2020)*. Association for Computing Machinery, New York, New York, USA, 519–532. https://doi.org/10.1145/3385412.3386008

[64] David W. Walker. 2018. Morton ordering of 2D arrays for efficient access to hierarchical memory. *The International Journal of High Performance Computing Applications* 32, 1 (2018), 189–203. https://doi.org/10.1177/1094342017725568

[65] David W. Walker and Anthony Skjellum. 2023. The Impact of Space-Filling Curves on Data Movement in Parallel Systems. https://doi.org/10.48550/arXiv.2307.07828 arXiv:2307.07828 [cs.DC]

[66] Jonathan Weinberg, Michael O. McCracken, Erich Strohmaier, and Allan Snavely. 2005. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (Seattle, Washington, USA). IEEE, New York, New York, USA, 50–50. https://doi.org/10.1109/SC.2005.59

[67] Albert-Jan Nicholas Yzelman and Rob Hendrik Bisseling. 2012. A Cache-Oblivious Sparse Matrix–Vector Multiplication Scheme Based on the Hilbert Curve. In *Progress in Industrial Mathematics at ECMI 2010*. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 627–633.

Energy Efficiency Features of the Intel Alder Lake Architecture

Robert Schöne
Markus Velten
Daniel Hackenberg
Thomas Ilsche
robert.schoene@tu-dresden.de
markus.velten@tu-dresden.de
daniel.hackenberg@tu-dresden.de
thomas.ilsche@tu-dresden.de
CIDS, Information Services and High Performance Computing (ZIH)
TU Dresden
Dresden, Germany

ABSTRACT

Intel's first heterogeneous processor, Alder Lake, combines two different core architectures from the Core and Atom families: Golden Cove and Gracemont, respectively. While the heterogeneity of this chip can improve performance and energy efficiency, it also increases the complexity of scheduling decisions and power saving mechanisms. In this paper, we analyze performance and energy characteristics of an Alder Lake system and describe effects of power saving mechanisms. We evaluate the factors that influence the time required to switch core and uncore frequencies and waking cores from idle states. In addition, we assess the efficiency of the two core architectures across various workloads. We show that in states with low power consumption, RAPL energy measurements are inaccurate, and actual (externally measured) power consumption also exhibits peculiar patterns. Through experiments, we also examine the newly introduced user space idle states, and the novel telemetry capability. This information can be used by other researchers to design efficient software and further experiments, and explain measured performance on heterogeneous Intel processors.

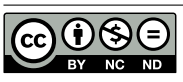
CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; **Heterogeneous (hybrid) systems**; • **Hardware** → **Chip-level power issues**; **Platform power issues**.

KEYWORDS

Intel, power management, energy efficiency, DVFS, idle state, RAPL

ACM Reference Format:
Robert Schöne, Markus Velten, Daniel Hackenberg, and Thomas Ilsche. 2024. Energy Efficiency Features of the Intel Alder Lake Architecture. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645040


This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645040

1 INTRODUCTION

The continuous evolution of processors requires vendors to translate ever-growing transistor budgets into performance improvements, e.g., by including more functional units, memory controllers, input/output (I/O) interfaces, graphics processing units (GPUs), and caches. This trend also increases complexity, which cannot be fully hidden from the operating system (OS) or application domains. Issues like *where to place threads* if cores have different frequency ranges or architectures, or *where to perform a task* that might be hardware-accelerated cannot be decided on a hardware level. Moreover, performance improvements need to be achieved within a limited power envelope with energy efficiency as a first order design goal. Introduced power saving techniques, however, can contradict OS and applications performance assumptions.

Several processor vendors offer heterogeneous processor architectures, such as ARM's big.LITTLE or Apple M1, combining high-performance and power-efficient cores. Intel's first such architecture, Alder Lake, integrates different core architectures and various accelerating components. This work presents an architecture overview of Alder Lake and an in-depth analysis of its power efficiency properties and techniques. For example, this includes frequency scaling of different components, idle states and their latencies, integrated energy measurement capabilities, and recently introduced processor feedback interfaces and OS integration.

2 BACKGROUND AND RELATED WORK ON ENERGY EFFICIENCY MECHANISMS

Dynamic Voltage and Frequency Scaling (DVFS) describes the ability of a processor to change frequencies and voltages at runtime as a trade-off of power and performance. Contemporary processors have multiple frequency and voltage domains, e.g., one for each core and one for uncore components. The decision to change frequencies is influenced by multiple factors: first, the allowed frequency or frequency range set by the operating system, which can be influenced by the user [22, 34]; second, the internal control mechanisms, which uses one of the allowed frequencies [16, Section 14.4]; and third, protection mechanisms to prevent overheating and a high power consumption [16, Section 14.10]. After the decision is made, frequencies are not changed instantaneously, but only with some delays to change voltages as well. While the information about

the duration of a frequency change can be communicated by the hardware to the OS in ACPI P-state tables [32, Section 8.4.5], these values are not necessarily correct [21]. Moreover, internal control mechanisms delay the decisions further [6, 28].

DVFS has been used by researchers to optimize the energy efficiency of programs using two different approaches: First, the region-based tuning of frequencies uses the characteristic of a code region to lower the frequency of components when they are not used. One example is the reduction of processor core frequencies when code is memory-bound, as described, for example, by Kumaraswamy et al. [19] and Vysocky et al. [33]. The second approach applies to parallel applications where the non-critical paths can be slowed down to reduce the energy consumption of the cores that execute them. This has been shown for example by Rountree et al. [25]. Characteristics of frequency transitions have been studied by Mazouz et al. [21]. We described additional details in [6, 28, 29].

Idle States are hardware power saving mechanisms that can be used by the OS or the hardware to switch off the clock (*clock gating*) or voltage (*power gating*) of a part of the processor that is not actively used to lower power dissipation. Operating systems typically use instructions like `hlt` or `mwait` to let CPUs idle [23]. Processors can also employ idle states for whole packages, including cores and uncore components [26]. Parts of a processor core can also be disabled when they are not used [28]. However, idle mechanisms introduce latencies when re-enabling the idling components. Characteristics of idle state transitions have been studied by our previous work for various architectures in [8, 26, 28, 29].

Modern high performance processor architectures typically operate under thermal and/or power constraints when fully utilized [7, Sec. 1.5]. Consequently, processors are equipped with mechanisms for **Power Limiting and Thermal Protection** to enforce operation within the given constraints. Intel introduced the Running Average Power Limit (RAPL) [16, Sec. 15.10], which aims to maximize performance while ensuring safe operation. RAPL also provides energy measurement data that can be read from counters for certain power domains. Monitoring power for thermal protection is also available on other platforms such as AMD [4, 29] and IBM [31]. In previous work we detailed accuracies and other properties of processor and platform power measurement interfaces [4, 6, 28, 29, 31]. Processors can also use clock modulation to lower power consumption for thermal protection. Here, parts of a processor are periodically clock gated for a certain timeframe to lower power dissipation [27]. Another approach forces cores into idle states periodically, including all benefits and costs discussed earlier. This concept is called Hardware Duty Cycling (HDC) on Intel processors [16, Section 15.5].

3 ALDER LAKE PROCESSOR ARCHITECTURE AND TEST SYSTEM

The Intel Alder Lake processor family is a heterogeneous architecture that can include the following computing components: Up to eight performance cores (P-cores) using the Golden Cove architecture, up to eight efficiency cores (E-cores, located in two modules with four cores per module) using the Gracemont architecture, an Intel Xe Gen 12.2 GPGPU with up to 96 execution units (EUs) and a Gaussian-Neuronal-Network-Accelerator (GNA Version 3).

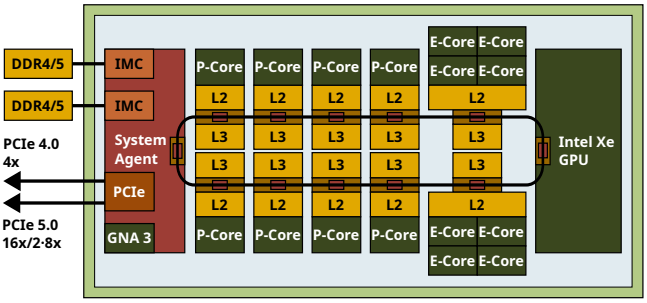


Figure 1: Block diagram of the Intel Core i7-12900K processor

The processor cores are connected via shared caches[1]. To transport data from and to the processor, it supports 16 PCIe 5.0 and 4 PCIe 4.0 lanes, Thunderbolt 4.0, Direct Media Interface (DMI) 4.0 and memory controllers for DDR4 and DDR5. The P-, H-, and U-processor line also include an Image Processing Unit (IPU), which provides support for camera functions (e.g., white balance and color matching). A Volume Management Device [11, Section 2.8] adds hardware RAID support below the OS level. The specification of Alder Lake processors offer a range of configurable parameters for thermal management / power control. Four power limits (PL1 - PL4) define increasing thresholds, starting with PL1 as the average power over long time, up to PL4 as a limit never to be exceeded (see [11, Section 4.1.1]). From these, PL3 and PL4 are disabled by default. When a system (platform) power measurement is available, platform power limits (PsysPL) can further enforce thermal limits beyond the scope of the processor. PL1 and PL2 limitations can also affect main memory accesses, as [13, Section 3.3.13] hints.

Our test system hosts an Intel Core i9-12900K processor with 8+8 processor cores and 32 GPU EUs with a TDP[2] of 125 W. Appendix B lists the full test systems specifications. Figure 1 shows a block diagram of the processor. The Intel powercap kernel module reports 4 kW for the `long_term` and `short_term` RAPL constraints (`power_limit_uw`). According to settings regarding Temperature Targets [13, Section 3.3.28], the processor throttles at 100 °C and the fans are engaged at 80 °C. While the interfaces for Hardware Duty Cycling are listed in [17, Table 2-39], the feature itself is not available on our system according to `cpuid`, and the MSRs are not accessible. We measure power consumption on the AC side using a ZES LMG450 power meter and collect data externally using MetricQ [9] for processing.

4 DYNAMIC VOLTAGE AND FREQUENCY SCALING

Alder Lake processors have clock domains for: cores, the GPU, the memory controller, the system agent, and the uncore including L3-slices and ring. Different interfaces can be used to change core frequencies: The operating system can use model-specific registers (MSR) with the Enhanced Intel SpeedStep Technology [11, Section 2.4.8] [16, Section 14.1]. While these MSRs can be accessed

[1] All cores share the last level cache (LLC / L3), a set of four E-cores shares a mid-level cache (MLC / L2)

[2] The acronym TDP originates from Thermal Design Point [24] or Thermal Design Power but is now also referred to as Processor Base Power [13, Table 1].

per CPU[3], Intel states that *"all active processor IA cores share the same frequency and voltage"* [11, Section 2.4.8]. We validate this in Section 4.1. Alternatively, the processor can control core frequencies transparently using Hardware-Controlled Performance States (HWP) [16, Section 14.4], also known as Intel Speed Shift Technology [11, Section 2.4.10]. This can still be influenced by the OS by regulating the minimal and maximal allowed frequency and the preference for performance or power saving. Since frequency control is implemented with MSRs, a remote core's frequency can only be changed by interrupting that core's work. To cover this issue, the operating system can use a new mechanism Remote Action Request (RAR) [11, Section 2.4.16]. With RAR, a HWP request can be broadcast to all cores of the system [2]. However, on our system, the RAR information register [2, Table 4-1] is not accessible. We describe how long it takes a core to change its frequency in Section 4.2. Frequencies can be increased above the nominal frequency using Turbo mechanisms [11, Sections 2.4.5, 2.4.7, 2.4.10], within the given thermal and power limits. One of the reasons for a high power consumption of processors and a possible reduction of frequencies is the execution of compute-intense instructions. On server processors specific frequency bands are applied when such instructions are used [6, 28]. In Section 4.3, we check whether such mechanisms are also present in the Alder Lake architecture.

The frequency of uncore components is regulated by the processor but can also be influenced by the operating system [17, pp. 2-332f], which exposes this functionality with the *uncore-frequency* driver. Previous work showed that these definitions cannot be considered to be hard limits and that a regulation mechanism will adapt the uncore frequency to workloads on server processors [28]. We cover this for Alder Lake processors in Section 4.4.

The frequency of the integrated GPU can also be changed by the operating system and the hardware using different interfaces [11, Section 3.4.3]. Linux exposes this option to the userspace with the *i915* driver. The datasheet [13, Section 3.3.22] states that compute (slice) components of the GPU have a different frequency than other (unslice) components. We describe the operating system interfaces and the supported frequencies in Section 4.5.

[3]We use *CPU* to refer to a logical OS CPU, which corresponds to a hardware thread.

4.1 Frequency Interdependencies

At first, we check if the core frequencies depend on each other. We run a `while(1);` workload on the tested CPU with a low frequency and set a higher frequency on a different CPU that is either active or idle. An increased frequency of the tested CPU suggests a shared frequency domain. The experiment reveals that all active cores of the processor share one frequency domain running with the highest frequency set for any of these cores. In addition, idling P-core CPUs can increase the frequency of the non-idling CPU of the same core. Likewise, idling E-cores influence other cores in their module.

4.2 Frequency Change Latency

To measure the time until a frequency change is applied, we use the method introduced by Mazouz et al. in [21] and refined in [6, 28]. We pin the frequency of unused CPUs to 800 MHz. The workload starts at the source frequency, waits for a random time between 0 ms and 10 ms, triggers a frequency switch via sysfs, measures the start time and monitors the runtime of a short loop until it fits the performance expected for the target frequency. As soon as that happens, it takes the end time and verifies that the new performance is stable. Unstable outliers are marked as such and filtered from the analysis. Afterwards, it resets the frequency to the initial frequency, verifies it by measuring the short workload and starts over by waiting for a random time to measure the next switch. We take 2×10 000 samples for P-cores and E-cores, respectively.

Figure 2 shows the latencies for P- and E-cores switching from 3.2 GHz to 3.0 GHz and 1.0 GHz to 0.8 GHz depending on the wait time after resetting to the initial frequency. Initially, the transition is fast, e.g., 38.7 µs (P-core, wait time < 2 ms, 3.2⟶3.0 GHz, median). Starting approx. 2.1 ms after the last frequency change, the behavior for P-cores changes and now falls in two categories: For the 3.2⟶3.0 GHz-switch, we see a periodic pattern with a period of ≈200 µs where a new core frequency can be applied at absolute points in time. Such a behavior was reported for server processors in [6, 28, 29]. The latencies for this periodic behavior range from 69.6 µs to 272.2 µs (P-core, wait time > 2.5 ms, 3.2⟶3.0 GHz, 1 % and 99 %-quantiles). In contrast, the 1.0 GHz⟶0.8 GHz transition is now significantly faster without a further time dependent pattern.

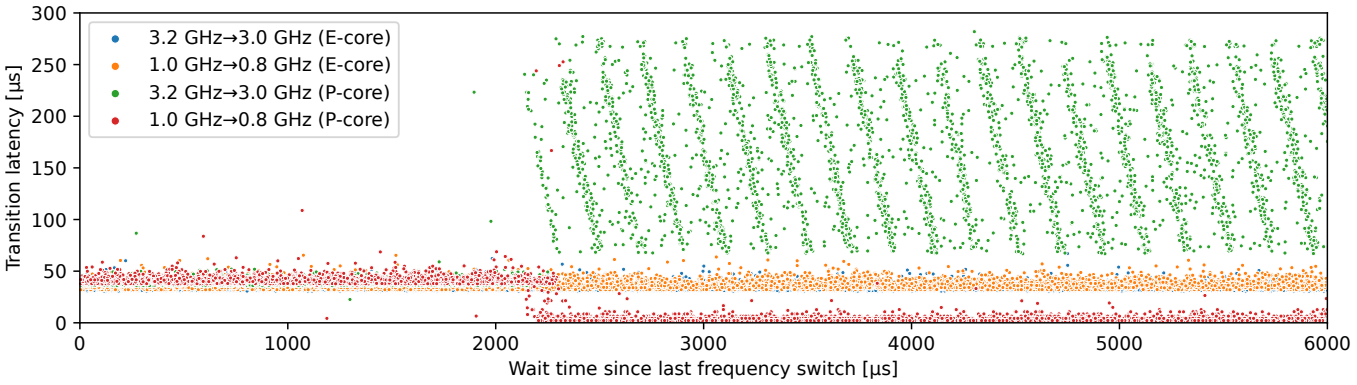
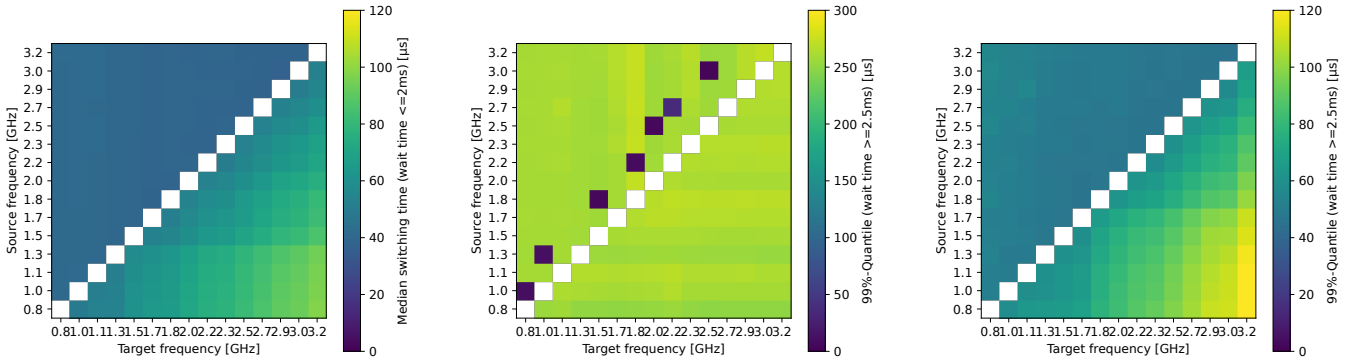


Figure 2: The time until a new frequency is applied depends on the wait time since the last frequency change. The latency for E-cores is independent of that wait time. Latencies on P-cores behave identical to those of E-cores for low wait times. For longer wait times, patterns emerge for P-cores which depend on the start and target frequencies.



(a) P-cores: Median of frequency latencies (<2 ms)

(b) P-cores: 99th percentile of frequency latencies (>2.5 ms)

(c) E-cores: 99th percentile of frequency latencies (>2.5 ms)

Figure 3: Frequency switch latencies, differentiated to short (< 2 ms) and long time (> 2.5 ms) after last switch.

Overall, the transition latencies within the first 2 ms after a previous switch depend only on source and target latencies (see Figure 3a). While lowering frequencies can be done faster without waiting for the voltage to change, increasing frequencies first needs to increase voltages. Figure 3b shows that, for P-cores after 2 ms, most combinations follow the pattern of the 3.2 GHz→3.0 GHz transition. However, there are some combinations with a low and constant latency similar to the 1.0 GHz→0.8 GHz transition. The latter have some things in common: first, it is always a reduction of frequency and second, the source and target frequencies are relatively close. This could be due to voltages not being changed during the last frequency reset. In addition to that, some target frequencies also show a pattern where a significant number of samples were not valid. We provide more data with the reproducibility package.

These complex effects do not occur on E-cores, as shown in Figure 2 and Figure 3c. Here, the initial pattern of a stable minimal latency and a range of values above the minimum is visible even for higher wait times since the previous frequency switch.

4.3 AVX Frequencies

The perfmon events website lists the event *CORE_POWER* supported by P-cores [10, Event CORE_POWER]. The same event name is used on Intel Skylake processors to monitor processor cycles spent in different frequency bands (standard, AVX, AVX-512 frequencies). On AlderLake P-cores the event lists three different *power licenses*: 1-3, which can be selected using the umasks 0x02, 0x04, and 0x08 respectively. We validate these events by running FIRESTARTER [5] and a `while(1);`-loop while sampling the performance monitoring counter (PMC) every 1 s using `perf stat`. During the `while(1);` workload, the PMC counts cycles if the umask is 0x1. We therefore argue that the umask 0x1 refers to license 0. Running FIRESTARTER workloads with SSE and AVX triggers the PMC to count licenses 1 and 3, respectively.

License 2 is used whenever switching from license 3 to license 0, as determined with `perf record`. Based on the number of cycles spent in license 2 and the duration of a workload, cores presumably spend about 640 µs in license 2 before switching back to license 0. This time correlates with the time to switch back from the AVX-512 to the standard frequency band on Skylake server processors [28, Section VII]. From this, we conclude that it is likely that a mechanism for applying AVX frequency-ranges is implemented in Golden Cove cores. However, we could not see any impact of this mechanism on applied frequencies.

4.4 Uncore Frequency

As in previous products, the uncore frequency is usually regulated within a pre-defined frequency range by an internal control loop [6, 28]. For our processor, the default range spans from 800 MHz to 4700 MHz according to the `UNCORE_RATIO_LIMIT` MSR. By manipulating this register we can reduce the lowest frequency to 400 MHz, but we cannot increase the maximal frequency. In all cases, the uncore frequency is set 200 MHz below the core frequency[4]. The exceptions are the following:

(1) The given bounds are not exceeded in the default case – i.e., at a core frequency of 800 MHz, the uncore still uses a frequency of 800 MHz
(2) Whenever a workload runs on P-cores and E-cores with enabled turbo frequencies, the uncore frequency is reduced to 3.6 GHz–3.7 GHz (200 MHz below E-core turbo). This also overrides the previous exception. Even if the minimum is set to > 3.7 GHz, the uncore frequency stays at the same level.

In some scenarios, users might want to manipulate the uncore frequency manually for energy efficiency reasons, e.g., lowering it during code sections with no offcore accesses. We evaluate the occurring latencies with the methodology introduced in [28]. We measure 1000 transitions for each pair of source and target frequencies and evaluate the median, as depicted in Figure 4. We found the transition latency to depend on the source and the target frequency. While the qualitative pattern is the same as for frequency switches on cores (see Figure 3a and Figure 3c) the quantitative values are different. Lowering the frequency is much faster at about 26.5 µs. Increasing uncore frequencies can take longer compared to core frequency changes.

[4]When disabling the BIOS setting *Ring to Core offset (Down Bin)* the uncore frequency will equal the core frequency. However, by default it is enabled.

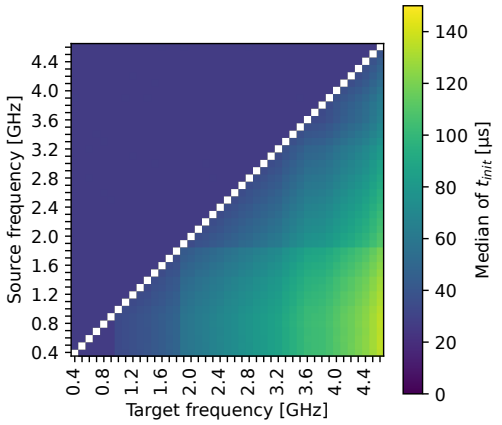


Figure 4: At the lowest core frequency (0.8 GHz), changing the uncore frequency takes between 26.4 µs (clocking down) and 135.0 µs, depending on source and target frequency.

4.5 GPU Frequencies

We use the sysfs interface and `perf stat` events for the i915 device to test valid frequencies on the GPU. The sysfs interface provides three different files to specify the minimal, maximal, and boost frequency in MHz (`gt_{min|max|boost}_freq_mhz`), from which we set the latter two. `perf` lists multiple events for the i915 device, from which we use *actual-frequency* and *requested-frequency*. Both counters only increase when the GPU is actively used by the system. In a first step, we check the supported frequencies and steps. To do so, we increase the requested frequency in 1 MHz-steps while running a matrix multiplication on the GPU. Results show that the actual frequency is always a multiple of 50 MHz. The requested frequency is a multiple of 16.667 MHz. To fit this scheme to the given interface requested frequencies are rounded as follows: First, it is rounded to the nearest multiple of 16.667 MHz. Then it is rounded up to a multiple of 50 MHz.

5 IDLE STATES

Operating systems use idle states to lower power consumption whenever there is no task scheduled on a CPU. Information from ACPI tables, populated by hardware, are the basis to decide which idle state to use [32]. This includes an estimation of the time to re-enable CPUs. As latencies can be several hundred microseconds [26], deep idle states should be disabled for latency sensitive scenarios [3]. In other cases, a lower average power consumption is preferred. While ACPI tables can hold information on projected power consumption, this information is often not set or invalid for idle states. Latency and power consumption additionally depend, for example, on applied frequencies and the activity of other cores.

Alder Lake cores implement three different core C-states: C0 (active), C1 (clock gating, can be combined with DVFS to C1E), and C6 (power gating). As previous processors, the core can autonomously switch from C6 to C1/C1E, which is called auto-demotion. Additional package C-states are used whenever all cores and the GPU reside in a higher C-state (C8, C10). Package C-states can limit the functionality of PCI links and other busses like USB and xHCI [13, Section 3.7f] or flush the L3 cache. During package C-state transitions or whenever a device is still active, PKG C2 can be used where cores are still in a deep idle state, but uncore components can be active. The datasheet [13, Table 8] lists more details about package idle states. We describe the usage of C-states and the power saving potential of our system in Section 5.1 and the time to return to an active state in Section 5.2.

Alder Lake introduces two new idle states[5] (C0.1 and C0.2) that – unlike other C-states – can be entered from userspace [15, Table 4-21]. The two different idle states vary in their wakeup time, power savings, and effects with Simultaneous Multi-Threading (SMT): In C0.2, the wakeup time and power savings are higher and the performance of the second CPU on a core improves. We measure power consumption of C0.1 and C0.2 states in Section 5.1 and analyze the latency to return from these idle states in Section 5.3.

5.1 C-States and Power Consumption

We use different idle states on all cores of the system to determine their power consumption. Our experiments cover the five C-states available through the OS: C0 (POLL), C1E (hlt), C6, C8, and C10, as well as active idling workloads (widely unrolled `NOP`, `PAUSE` for C0, and `TPAUSE` for user idle states C0.1 and C0.2). We use the register `MSR_IA32_POWER_CTL` to disable the transition to C1E. We further test at different core frequencies. To measure the usage of hardware C-States we monitor the idling periods with `perf stat`'s event groups `cstate_core/` and `cstate_pkg/`. Concurrently, we monitor system power consumption using the out-of-band MetricQ framework [9] . We use the mean power measured over an interval of 10 s (11 seconds measured, data from the first second is omitted) with a sampling rate of 20 Sa/s.

According to our measurements, P-cores use the C6 state if requested by the OS. The event counter `cstate_core/c7-residency/` increases if C8 or C10 are requested. E-cores on the other hand use the C6 state, even if higher states are requested. This is surprising, since the OS requests C10 to a high extent according to the usage stats provided by the idle driver. With these core C-states, the highest package C-state that can be used is package C6. Nevertheless, only the package C-states PC2 and PC3 are used, where attached USB and video devices result in a higher proportion of PC2 usage. We provide data, the analysis script and plot for these numbers in the reproducibility data package.

The power consumption resulting from using different idles states is shown in Figure 5. As expected, power consumption in C1 and C0 (including active idle routines) increases with the applied core frequency since power gating is not used. For other C-states, the power consumption is independent of the applied frequency. The usage of C8 and C10 results in the same power consumption, which is consistent with the previous analysis with `perf`. If C8 and C10 are not allowed, the cores use C6 and power consumption increases from 37 W to 43 W. The plot show a particular anomaly: at low core frequency, the power is slightly higher when using C6 than under C1. However, due to a peculiarity that we describe in Section 7, power consumption in these cases is generally inconsistent.

[5] Originally, they were introduced to Atom line with Tremont processors. Now they are also available for performance cores.

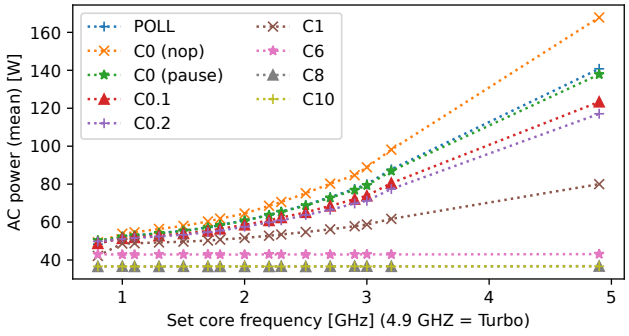


Figure 5: System power consumption depending on used idle mechanism and frequency by cores. 4.9 GHz refers to the usage of Turbo frequencies (measured for P-cores in C0).

5.2 Idle State Latencies

To measure latencies for returning from idle states, we use the methodology from [8]. Here, a thread running on one core (caller) sends a `pthread_cond_signal` to another core (callee), which waits using `pthread_cond_wait`. We measure times as a difference between the Linux kernel events *sched:sched_waking* from the caller and a *power:cpu_idle* to an active state at the callee.

The core C-state C10 is not used (see Section 5.1). Surprisingly, the latencies for C6 and C7 (as initiated by allowing C8 and C10 states) are similar (not depicted). As shown in Figure 6, latencies decrease with an increasing core frequency up to about 2 GHz. Afterwards, there is no clear pattern. The measured times are in the same order of magnitude as Skylake server processors [28].

5.3 User Space Idle State Latencies

In addition to idle states that can only be called from an operating system, Intel implements new instructions that can trigger idle behavior from user space. These include a timed pause instruction and a user-space implementation of `monitor/mwait`. With these, threads can indicate to the processor that it should stop fetching and executing new instructions while still being scheduled on the CPU. To limit the duration of these idle periods, operating systems can configure a maximal sleep time in the `UMWAIT_CONTROL` MSR (25 000 cycles on our system).

We check the latency of waking cores from user-space idle states by accessing `umonitor`'ed data. To do so, we setup `umwait` to use a specific state and schedule two threads on two CPUs: a caller and a callee. The callee waits for a volatile date to reach a certain value, then it monitors this value using `umonitor` and waits for it to change using `umwait`. The caller waits for a specific time to take a timestamp using `rdtsc` and change the monitored value. The change causes the callee to wake up and also take a time stamp. We store the difference of both timestamps for the analysis. In Figure 7, we show the distributions for waking up different core types from CPU 0. Here, we see that E-cores (CPU 20) have a lower latency than P-cores (CPU 1). In C0.1 we see an anomaly, where about 8 % of the samples have a higher latency. It seems that the E-cores do not support the C0.2 state: The latencies including the C0.1 anomaly are the same, regardless of the requested state. Also, the spatial distance of cores has an influence on these latencies, but the analysis is out of scope. Data can be found in the reproducibility package.

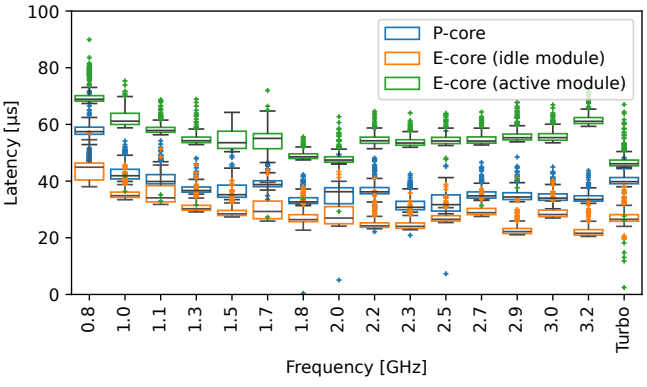


Figure 6: C-state wakeup latencies for C6. For P-cores and E-cores depending if all cores in it's module are idle

6 EFFICIENCY OF IMPLEMENTED COMPUTE-ARCHITECTURES

The two different core architectures and multiple accelerator architectures of Alder Lake processors each implement their own performance and power profile. According to [18, Section 2.2.1] *P-cores provide single or limited thread performance, while E-cores help provide improved scaling and multithreaded efficiency* (see also [18, Section 2.3, Section 4.1]). To support an efficient usage of this heterogeneity, Alder Lake provides a feedback interface for the operating system that gives information about the recently executed workload, distinguishing four classes: *Non-vectorized integer or floating-point code, [...] vectorized code [...], Intel [Deep Learning] Boost code, Pause [...] dominated code* [18, Section 2.2.2.1]. We evaluate this interface in Section 6.1. Intel also implements a Hardware Feedback Interface (HFI), which describes the performance and energy efficiency of each available CPU [16, Section 15.6]. This interface can, e.g., be used by the OS to select specific cores for different workloads. The entries *"may change at runtime as a result of changes in the operating conditions of the system or the action of external factors"* [1], which then can be used for scheduling decisions.

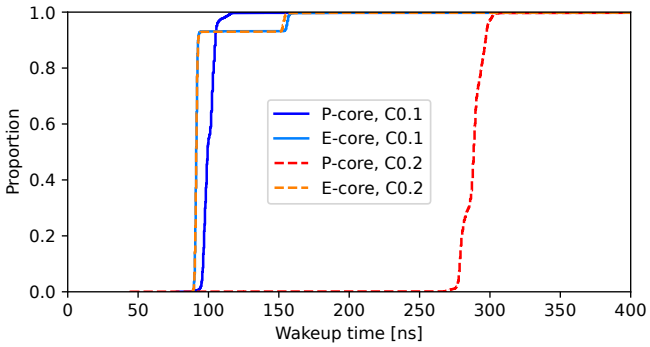


Figure 7: User space idle state wakeup latencies (ECDF)

On Alder Lake processors, Intel implements an Intel Xe Gen 12.2 GPU [13, Section 9.1.1], which can also be used to accelerate computing. Section 6.2 compares performance and energy efficiency of BLAS routines on processor cores and the GPU. Shared memory resources like main memory can have a strong influence on the energy efficiency of workloads [6, 28, 29]. Unfortunately, an analysis of this topic is out of scope for this paper.

6.1 Intel Thread Director

The OS can use the Thread Director interface to observe and schedule threads to cores that are deemed efficient for their workload. Intel lists typical workloads that represent the four classes in [18, Section 2.2.2.1]. We use these workloads in a loop running on all CPUs of the processor. Then, we enable the mechanism via the MSRs `HW_FEEDBACK_CONFIG` and `HW_FEEDBACK_THREAD_CONFIG` and poll the `THREAD_FEEDBACK_CHAR` MSR while executing the workloads. Finally, we compare the provided mapping to the executed workload. E-cores never provide valid samples (bit 63 is set to 0). P-cores can correctly identify classes 0, 2, and 3, but map the class 1 workload to class 0. As the information is missing on E-cores, we doubt that the OS can effectively use this interface to manage scheduling decisions, e.g., migrating unsuitable threads await from E-cores.

6.2 Comparison of Performance and Efficiency

To compare the efficiency of P-cores, E-cores, and GPU, we use three different BLAS functions: dot, sgemv, and sgemm. We vary the frequency of the respective computational unit(s) across the set of specifically selectable frequencies, but not the full turbo range. Thus, the processor runs significantly under its power budget. For GPU workloads, core frequencies are set to 3.2 GHz. The problem sizes are set such that data does not fit in caches. We use the Intel Math Kernel Library (MKL) implementations of sdot, sgemv, sgemm with $N = 1G, 40k, 10k$ respectively. Specific core types are selected with `taskset` for CPU and OpenMP target directives for GPU whereas MKL controls the final number and distribution of threads. We use 100 repetitions for sdot/sgemv and 10 repetitions for sgemm to achieve stable power consumption, but use the median of power samples to avoid impact from initialization and measure the overall execution time to compute the floating-point performance.

Figure 8 shows the resulting performance depending on processing and uncore frequencies. First, we focus on P-core performance at a variable uncore frequency (default). Due to the low arithmetic intensity, the performance of sdot and sgemv is dominated by memory accesses with diminishing benefits from high core frequencies. At default uncore settings, the core frequency only indirectly affects performance via the variable uncore frequency. E.g., P-core frequencies of 800 MHz and 1000 MHz show the same performance since both imply an uncore frequency of 800 MHz as described in Section 4.4. We validate this assumption with the results from a fixed uncore frequency of 3.2 GHz. On the E-cores and the GPU, the uncore bottleneck does not apply and the performance does not appear to saturate in a memory-bound configuration. The sgemm-kernel is purely compute bound for all architectures and frequencies with a performance growing linearly with the core frequency. While we executed the GPU kernels from different host core types and at different uncore frequencies, both performance and power consumption only depends on the GPU frequency.

P-core configurations exhibit the highest overall power consumption, except at the lowest core frequencies. Due to their higher turbo frequencies, P-core power can be even higher, which is not covered by this benchmark. However, due to their superior performance, they are still the most energy-efficient choice for sdot and sgemv with optimal core frequencies around 1800 MHz at default uncore frequency. For sgemm, the GPU at its highest frequency is the most efficient. This efficiency considers the total system energy for the given hardware allocation when exclusively using one architecture. The results are therefore influenced by the power consumption of the common resources, e.g., memory and fans and would differ with a different hardware allocation.

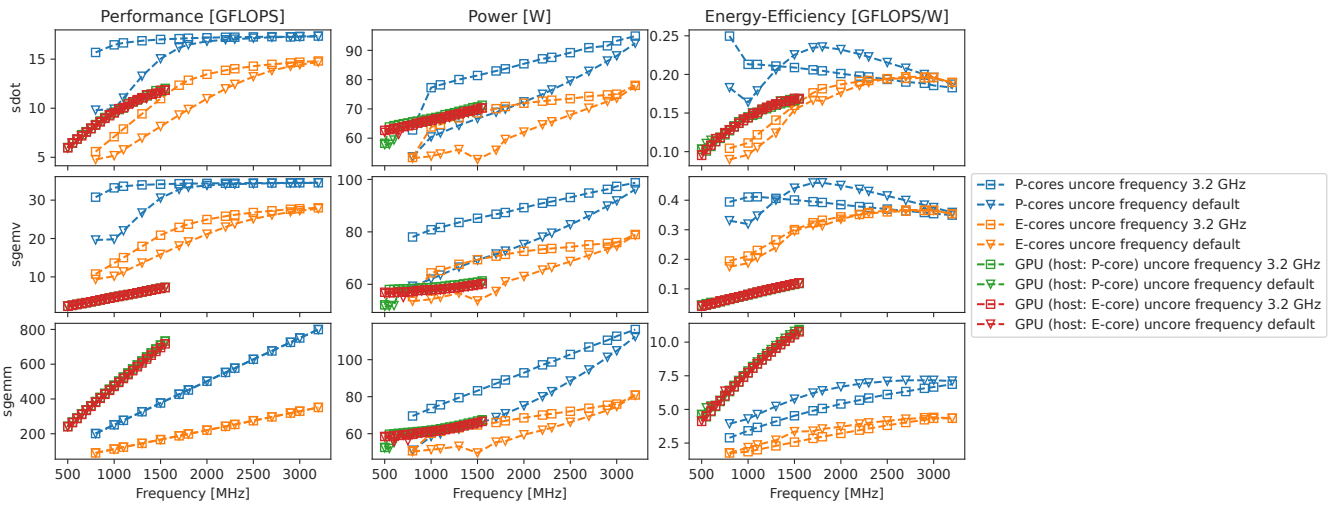


Figure 8: Performance, power and efficiency for different compute kernels on different compute architectures.

7 MONITORING ENERGY-EFFICIENCY-RELATED INFORMATION

The Running Average Power Limit (RAPL) [16, Section 15.10] provides enerhy counters since the Intel Sandy Bridge architecture. While early implementations used models, which were not necessarily accurate [4], newer server processors use physical measurements [6, 28]. The usual update rate of RAPL is 1 ms but can get as low as 50 µs for the PP0 domain (processor cores) on desktop processors. This can be used to retrieve processed data in a side-channel attack [20]. While this can be fixed by limiting access to RAPL counters from the OS side, Intel also implements a filtering technique via the `ENERGY_FILTERING_ENABLE` [17, Table 2-2] entry in an MSR. This filtering adds random noise to the reported values and can only be disabled with a system reset [14]. We analyze the temporal granularity and the filtering feature in Section 7.1

Power measurements for Alder Lake Processors can benefit from the VCCIN AUX IMON Feature to achieve *"more accurate package power reporting and better accuracy"* [11, Section 3.3]. This affects the enforcement of package power limits, but most likely also the accuracy of measuring package power consumption via RAPL. We analyze the accuracy of RAPL measurements in Section 7.2.

Another monitoring infrastructure is the Platform Monitoring Technology (PMT) or Telemetry Aggregator [11, Section 2.6.3], which records metrics of the processor out-of-band and was introduced with Tiger Lake. The data is made available to user space with the *intel_pmt_telemetry* driver and can include information about energy-related features. The definition of the encoded data is not documented. However, an Intel code repository mentions that the interface passes thermal, voltage and frequency information for Sapphire Rapids[6]. The description also notes that some information might be only available under a non-disclosure agreement. In Section 7.3, we analyze the information available on our system.

This study focuses on package (PCKG) and core (PP0) counters, since the DRAM domain reports 0 and the uncore component domain (PP1) reports 0 if the GPU [16, Section 15.10.2] is not used.

[6]https://raw.githubusercontent.com/intel/Intel-PMT/73cfa682/xml/SPR/OOBMSM/CORE/spr_aggregator.xml

7.1 Filter and RAPL Granularity

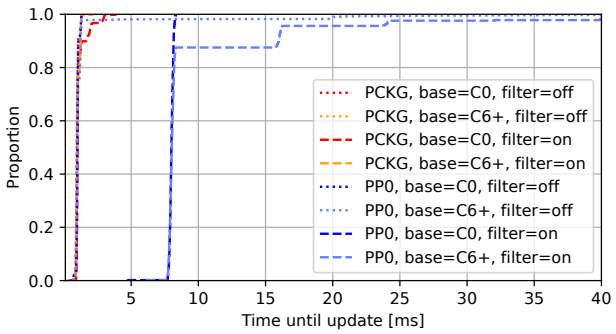
To measure the update rate of RAPL counters, we continuously poll the MSR from CPU 0 for 5 s. We take timestamps before reading the MSR and — if the measured energy changed — we store timestamp, read data, and the number of reading attempts since the last change. As Figure 9a shows, the temporal granularity depends on the monitored domain and the usage of the filter. For the PP0 domain, the temporal granularity is 8 ms if the filter is enabled. For all other cases, the granularity is about 1 ms. If the power consumption is too low for the energy in the MSR to increase, it will be reported in the next available moment (i.e. 1 or 8 ms later). Figure 9b shows effects of the filter on the reported energy. Package power is mostly the same as core power with an offset of about 1.4 mJ, i.e. 1.4 W over 1 ms when idling cores execute the C0 polling routine. If C6 is used, for more than 60 % of the samples, this difference is also visible. However for some samples, the reported package power is the same as the core power. This could hint at a mechanism, which takes energy of uncore components into account only if it is above a certain threshold.

Enabling the filter leads to significant changes: For the package power consumption, a lower energy is reported often. The values are mostly on par with PP0 without a filter for any C-state condition used. For core power consumption, the reported energy increases, as the time frame increases as well. However, the average power consumption over time does not change significantly.

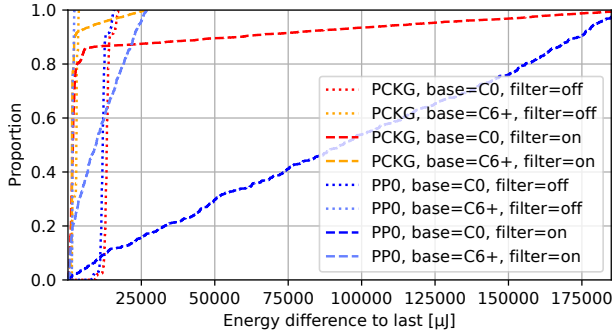
7.2 Accuracy of RAPL Measurements

To further understand the impact of the filter, we measure data-dependent power consumption with and without the filter enabled. To that end, we use run the `vxorps`[7] instruction with different number of set bits on all P-cores with nominal frequency and measure the energy with RAPL. Details of the method are explained in [29]. The experiment measures the `vxorps`-loop 1000 times with 0/50/100% of the bits being set (the operand weight). The result is then split according to the operand weight (defining color) and

[7]We chose vxorps for two reasons: it can be used in encryption algorithms (XOR cipher) and to be comparable to other architectures [29].

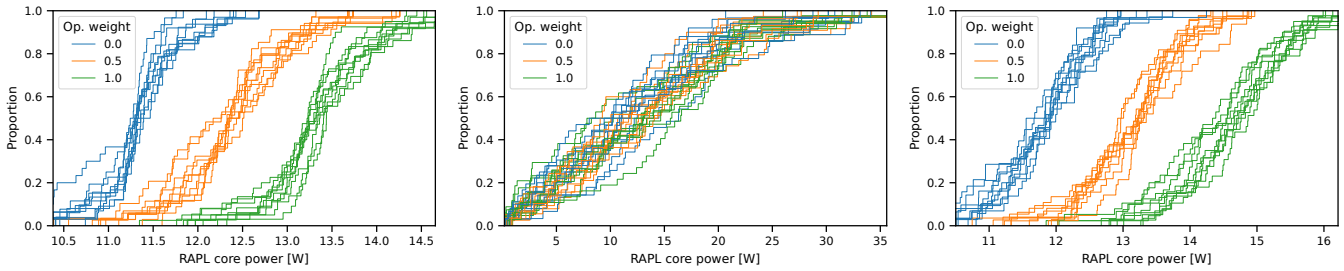


(a) Distribution of time between updates: With an enabled filter, the PP0 domain only provides updates every 8 ms, otherwise RAPL values are updated every 1 ms. If the load is too low, some updates might be skipped, e.g., the next update for PP0 and an enabled filter is at 16 ms.



(b) The minimal increase of a measurement is 1 energy unit of 61.035 µJ. Enabling the filter leads to a significant influence on measurements for the PP0 domain and a measureable influence on PCKG measurements.

Figure 9: RAPL information at 2.3 GHz core frequency, which is high enough to enable energy counting on the PP0 domain.



(a) No filter, 10 ms runtime (b) filtered, 10 ms runtime (c) filtered, 1000 ms runtime

Figure 10: RAPL power samples for measuring a `vxorps`-loop that runs for a given time with a given proportion of set operand bits (operand weight). Each line represents an empirical density function of 100 samples.

split each of these sets into 10 different subsets (to understand repeatability of the effect) that we plot in Figure 10. While the influence of data weight on power consumption is clearly exposed with 100 10 ms-samples without a filter, it cannot be reconstructed with enabled filtering. However, a 100-fold increase of the monitoring time still shows the data-dependent aspect of computations with a similar clarity. This shows that, given enough time, the filter does not prevent side-channel attacks via RAPL monitoring.

On a broader scale, we evaluate the RAPL implementation on our test system using a synthetic workload generator comparing RAPL values (average power derived from energy and time) with the external measurement of average AC power. The reference measurement covers a different domain, including power supply unit (PSU) losses, memory, and other off-chip components of the system. While a direct RAPL measurement error cannot be determined, this approach can be used to expose inconsistencies and systematic errors. The workload generator uses multiple microkernels that stress different components, including a focus on computation and memory. We run each kernel in a large number of configurations, varying core frequency, number of threads, and thread distribution across P-cores, their SMT-threads and E-cores. Each configuration runs for 30 s to prevent timing effects and focus on energy/average power (cf. [4]).

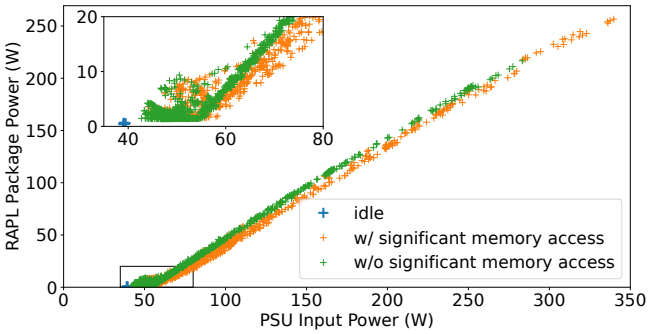


Figure 11: Comparison of power consumption measured by RAPL for the package and a full-system reference measurement. Each point represents the average power within a 30 s interval of the same workload configuration.

Figure 11 plots average AC power against RAPL package power, where each point represents one configuration of kernel, core frequency, and thread configuration. There is a consistent correlation between the two, with two exceptions. First, kernels that utilize the memory subsystem significantly, appear more noisy and report a lower RAPL package power for similar PSU input powers. This can be explained by RAPL not supporting the DRAM domain on this system. While this does not necessarily indicate that RAPL values are wrong, it confirms that RAPL package power alone cannot be used to accurately model total system power consumption, e.g., for energy optimization. The second abnormality occurs at low power consumption, where RAPL and the reference measurement diverge strongly. All kinds of workload kernels and thread configurations, including configurations of P-core and E-core usage exhibit this weak correlation. However, this only affects low frequency configurations: At nominal frequencies, a single thread executing any kernel uses more power than the abnormal cluster.

Figure 12 shows power measurements of different low-power computations with gradually increasing number of threads. While PSU power generally increases with the number of threads, it becomes noisy at 11 active threads with a significantly reduced average. Contrary, the reported RAPL power remains relatively constant at approx. 1.5 W for up to 10 active threads. Later, it increases with substantial noise for configurations where AC power is reduced and noisy Finally, it follows a more consistent pattern at higher power consumption configurations. This behavior is reproducible and we observed similar patterns for other workloads and low core frequencies at different thread count thresholds. We measured several hardware counters (core and uncore frequencies, instructions, power licenses), core temperatures, and workload utility. None of these correlated with the anomalies. We were not able to model or provide an explanation why the actual system power consumption and the RAPL measurements behave in such a way at low power.

We did not include GPU workloads in this evaluation, hence the GPU domain reports 0 W. The PP0 domain follows the package domain with a difference of 1.41 W for workloads w/o substantial memory accesses, up to 2.65 W for workloads w/ memory access, and 0.35 W in idle. In summary, RAPL offers plausible energy measurements with the exception of particularly low power configurations and the limitations to the processor itself.

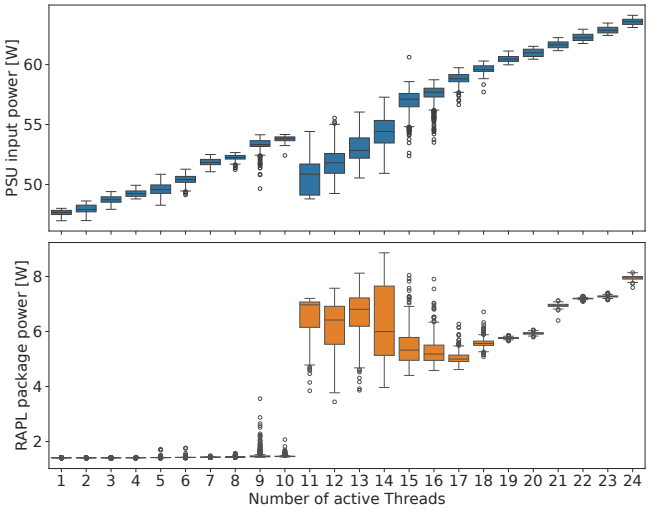


Figure 12: RAPL and reference power consumption sampled at 100 ms / 50 ms intervals respectively. Double precision matrix multiplication kernel at 0.8 GHz running for 60 s each at increasing number of active threads.

7.3 Platform Monitoring Technology

Our test system hosts two Intel PMT telemetry sources with GUIDs 0x85b7c2a0 and 0x409072a0 according to sysfs information. These provide 2752 Byte and 320 Byte of data, respectively. The latter always reads as 0. We run various workloads on all cores or a subset of cores and change hardware properties like core frequencies. We concurrently monitor the data from the main telemetry source by dumping the content of `/sys/class/intel_pmt/telem0/telem` in regular intervals as 4-Byte integers for the follow-up analysis.

In a first analysis step, we check whether the initial guess of data sizes (4 Byte) suits the measured data and distinguish between increasing counters and data that describes a current status. While the former increase over time until they overflow, the latter can change their values more freely. Some rules for the conversion of data types are: A 4-B-date a_i that is increasing towards the maximal integer and increases a_{i+1} on an overflow can be considered an 8-B-date. Any entry representing a current status where the upper 2 Byte are similar to the lower 2 byte represent two 2-B-dates. The same goes for four 1-B-dates. Any entry where a set of 8, 16, or 24 lower bits are not flipped can be shifted by that number of bits.

Some of the most interesting things that we reconstructed are the current frequency of P-cores and E-core-modules (beginning at offset 20), an activity bit-mask for them at offset 76, and their temperature at offset 144. Moreover, the 38.4 MHz signal, which might be related to a clock source of the chipset [12, Section 21], can be read at offset 280, the number of 38.4 MHz cycles a P- or E-core is active are stored at offsets 1840ff, and the uncore clock counts with 200 MHz below the highest frequency at offset 1368. However, some readings of the uncore clock were incorrect (decreasing) during package idle phases. Here it seems that the lower 4 B of the 8 B counter are reset during package idle, while the upper 4 B keep their content. This possible bug also affects other counters, e.g., per core activity related counters starting at offset 1192.

Table 1: Intel PMT data in 85b7c2a0

Starting offset [B]	Assumed content	Size [B]	Data type
20, 24, ...	Frequency of P-cores in 100 MHz	1	Instant.
52, 56	Frequency of E-core-modules in 100 MHz	1	Instant.
76	Bit mask: active P-cores and E-core modules	1.5	Instant.
144, 148, ...	Temperature of P-cores & E-core modules [°C]	1	Instant.
184-212	Core-related, peaks before throttling	2	Instant.
280	Increases with with 38.4 MHz	8	Increasing
1368	Uncore Clock Counter (200 MHz under core frequency) in cycles	8	Increasing
1840-1960	Increases with 38.4 MHz when core is not idling (first p-cores, then e-cores)	8	Increasing

8 SUMMARY AND OUTLOOK

This paper provides a multitude of analyses of power management and energy efficiency features of the first Intel processor generation with heterogeneous core architectures. We found that frequency switch timings differ for P- and E-cores. Only the former show a pattern known from server processors. Even though the architecture does not seem to use AVX frequency ranges, P-cores still support accounting for those. The uncore frequency depends directly on the core frequency and can fall below the set range when cores use Turbo frequencies. E-cores seem not to support some of the idle states of P-cores, namely C0.2 and idle states above C6. Waking cores from deep idle states takes as long as on server processors. User space idle states save power and have a wakeup time in the order of hundreds of nano seconds. The Intel Thread Director fails to identify class 1 workloads and is not present on E-cores. While the integrated GPU can be used to run floating point intense code efficiently, accessing memory seems to pose a bottleneck. When running memory-bound codes, P-cores are most efficient in terms of performance and energy efficiency. Increasing the uncore frequency can increase performance for memory-bound codes at low core-frequencies. The update rates for RAPL are at 1 ms. The filtering functionality can add noise to counter side-channel attacks. Enabling this filter leads to a lower update rate for the core domain. Still, data can be inferred from RAPL readings using longer observation. While RAPL is generally consistent with external measurements, it does not include DRAM and is inaccurate in low power scenarios. The platform monitoring technology provides a sideband measurement of power-related information. We unveil encoded information and describe a possible bug in 8 B-counters.

While not all of these various findings can be generalized beyond our specific system, they serve as a guideline of relevant effects. Moreover, we provide a detailed methodology as well as a reproducibility package [30] to facilitate translating the results to other systems. Some energy aspects are not covered in this paper and remain future work, including the following: In addition to DVFS, the integrated GPU supports various power saving mechanisms [11, Section 3.4] that could be investigated. We also did not analyze the Power Management Integrated Circuits mentioned in [11, Section 3.4] due to a lack of information. Other idle states like S-states or G-states could be investigated further. However, information on their effects are described in [11, 12]. It will also be interesting to look at Sapphire Rapids, which also implements Golden Cove cores, including an analysis of user space idle states, and AVX frequencies.

REFERENCES

[1] Linux Kernel documentation, x86-specific Documentation, Chapter 14: Hardware-Feedback Interface for scheduling on Intel Hardware. URL https://docs.kernel.org/arch/x86/intel-hfi.html. (accessed 2024-02-14).

[2] Intel Corporation. Remote Action Request White Paper, July 2021. URL https://www.intel.com/content/dam/develop/external/us/en/documents/341431-remote-action-request-white-paper.pdf. (accessed 2023-11-17).

[3] Corey Gough, Ian Steiner, and Winston A. Saunders. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress, USA, 1st edition, 2015. ISBN 1430266376.

[4] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E. Nagel. Power Measurement Techniques on Standard Compute Nodes: A Quantitative Comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, April 2013. doi: 10.1109/ispass.2013.6557170.

[5] Daniel Hackenberg, Roland Oldenburg, Daniel Molka, and Robert Schöne. Introducing FIRESTARTER: A processor stress test utility. In *2013 International Green Computing Conference Proceedings*, 2013. doi: 10.1109/IGCC.2013.6604507.

[6] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2015. doi: 10.1109/IPDPSW.2015.70.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, 6th edition, 2019. ISBN 9780128119051.

[8] Thomas Ilsche, Robert Schöne, Philipp Joram, Mario Bielert, and Andreas Gocht. System Monitoring with lo2s: Power and Runtime Impact of C-State Transitions. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018. doi: 10.1109/IPDPSW.2018.00114.

[9] Thomas Ilsche, Daniel Hackenberg, Robert Schöne, Mario Bielert, Franz Höpfner, and Wolfgang E. Nagel. MetricQ: A Scalable Infrastructure for Processing High-Resolution Time Series Data. In *2019 IEEE/ACM Industry/University Joint International Workshop on Data-center Automation, Analytics, and Control (DAAC)*, pages 7–12, 2019. doi: 10.1109/DAAC49578.2019.00007.

[10] Intel Corporation. Perfmon events Website - Alder Lake. URL https://perfmon-events.intel.com/ahybrid.html. (accessed 2023-11-17).

[11] Intel Corporation. 12th Generation Intel Core Processors Datasheet, Volume 1 of 2. Technical Report 655258, Rev.: 009, Intel Corporation, 2022. URL https://cdrdv2.intel.com/v1/dl/getcontent/655258. (accessed 2022-09-01).

[12] Intel Corporation. Intel 600 Series Chipset Family Platform Controller Hub Datasheet, Volume 1 of 2. Technical Report 648364, Rev 003, Intel Corporation, May 2022. URL https://cdrdv2.intel.com/v1/dl/getcontent/648364. (accessed 2023-11-17).

[13] Intel Corporation. 12th Generation Intel Core Processors Datasheet, Volume 2 of 2. Technical Report 655259, Rev 003, Intel Corporation, April 2022. URL https://cdrdv2.intel.com/v1/dl/getcontent/655259. (accessed 2023-11-17).

[14] Intel Corporation. Software Security Guidance Running Average Power Limit Energy Reporting / CVE-2020-8694 , CVE-2020-8695 / INTEL-SA-00389. Technical report, Intel Corporation, February 2022. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html. (accessed 2023-11-17).

[15] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, M-U. Technical Report 253667-081US, Intel Corporation, September 2023. URL https://cdrdv2.intel.com/v1/dl/getContent/671241. (accessed 2023-11-17).

[16] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. Technical Report 253669-081US, Intel Corporation, September 2023. URL https://cdrdv2.intel.com/v1/dl/getContent/671427. (accessed 2023-11-17).

[17] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers. Technical Report 335592-081US, Intel Corporation, September 2023. URL https://cdrdv2.intel.com/v1/dl/getContent/671098. (accessed 2023-11-17).

[18] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1. Technical Report 248966-048, Intel Corporation, August 2023. URL https://cdrdv2.intel.com/v1/dl/getContent/671488. (accessed 2023-11-17).

[19] Madhura Kumaraswamy, Anamika Chowdhury, Andreas Gocht, Jan Zapletal, Kai Diethelm, Lubomir Riha, Marie-Christine Sawley, Michael Gerndt, Nico Reissmann, Ondrej Vysocky, Othman Bouizi, Per Gunnar Kjeldsberg, Ramon Carreras, Robert Schöne, Umbreen Sabir Mian, Venkatesh Kannan, and Wolfgang E. Nagel. Saving Energy Using the READEX Methodology. In *Tools for High Performance Computing 2018 / 2019*, pages 27–53. Springer International Publishing, 2021. ISBN 978-3-030-66057-4. doi: 10.1007/978-3-030-66057-4_2.

[20] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 355–371, 2021. doi: 10.1109/SP40001.2021.00063.

[21] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. Evaluation of CPU Frequency Transition Latency. *Computer Science - Research and Development*, 2014. doi: 10.1007/s00450-013-0240-x.

[22] Venkatesh Pallipadi and Alexey Starikovskiy. The Ondemand Governor Past, Present, and Future. In *Proceedings of the Ottawa Linux Symposium (OLS)*, 2006. URL https://www.kernel.org/doc/ols/2006/ols2006v2-pages-223-238.pdf. (accessed 2023-11-17).

[23] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle: Do nothing, efficiently. In *Proceedings of the Ottawa Linux Symposium (OLS)*, 2007. URL https://www.kernel.org/doc/ols/2007/ols2007v2-pages-119-126.pdf. (accessed 2023-11-17).

[24] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2), 3 2012. ISSN 0272-1732. doi: 10.1109/MM.2012.12.

[25] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 460–469, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584980. doi: 10.1145/1542275.1542340.

[26] Robert Schöne, Daniel Molka, and Michael Werner. Wake-up Latencies for Processor Idle States on Current x86 Processors. *Computer Science - Research and Development*, 2014. doi: 10.1007/s00450-014-0270-z.

[27] Robert Schöne, Thomas Ilsche, Mario Bielert, Daniel Molka, and Daniel Hackenberg. Software Controlled Clock Modulation for Energy Efficiency Optimization on Intel Processors. In *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing*, E2SC '16, pages 69–76. IEEE, 2016. ISBN 978-1-5090-3856-5. doi: 10.1109/e2sc.2016.015.

[28] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, 2019. doi: 10.1109/HPCS48598.2019.9188239.

[29] Robert Schöne, Thomas Ilsche, Mario Bielert, Markus Velten, Markus Schmidl, and Daniel Hackenberg. Energy efficiency aspects of the amd zen 2 architecture. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 562–571, 2021. doi: 10.1109/Cluster48925.2021.00087.

[30] Robert Schöne, Markus Velten, Thomas Ilsche, Mario Bielert, and Daniel Hackenberg. Reproducibility package for paper "Energy Efficiency Features of the Intel Alder Lake Architecture", as submitted to ICPE 2024, March 2024. URL https://doi.org/10.5281/zenodo.10782722.

[31] Hannes Tröpgen, Mario Bielert, and Thomas Ilsche. Evaluating the Energy Measurements of the IBM POWER9 On-Chip Controller. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23, page 67–76, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700682. doi: 10.1145/3578244.3583729.

[32] UEFI Forum, Inc. Advanced Configuration and Power Interface (ACPI) specification, revision 6.5, August 2022. URL https://uefi.org/sites/default/files/resources/ACPI_Spec_6_5_Aug29.pdf. online at uefi.org (accessed 2023-11-17).

[33] Ondrej Vysocky, Martin Beseda, Lubomír Říha, Jan Zapletal, Michael Lysaght, and Venkatesh Kannan. MERIC and RADAR Generator: Tools for Energy Evaluation and Runtime Tuning of HPC Applications", booktitle="High Performance Computing in Science and Engineering. pages 144–159, Cham, 2018. Springer International Publishing. ISBN 978-3-319-97136-0. doi: 10.1007/978-3-319-97136-0_11.

[34] Rafael J. Wysocki. `intel_pstate` CPU Performance Scaling Driver. In *The Linux kernel user's and administrator's guide*, 2017. URL https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html. (accessed 2023-11-17).

A ACKNOWLEDGMENTS AND REPRODUCIBILITY

We would like to thank Mario Bielert, Hannes Tröpgen, and the anonymous reviewers for their valuable input. We provide artifacts for reproducibility at [30]. This work was funded by the Bundesministerium für Bildung und Forschung via the EECliPs project under grant no. 16ME0602. It is also financed on basis of the budget passed by the Saxon State Parliament in Germany.

B TEST SYSTEM SPECIFICATION

The test system we used is specified in Table 2.

Table 2: Test system details

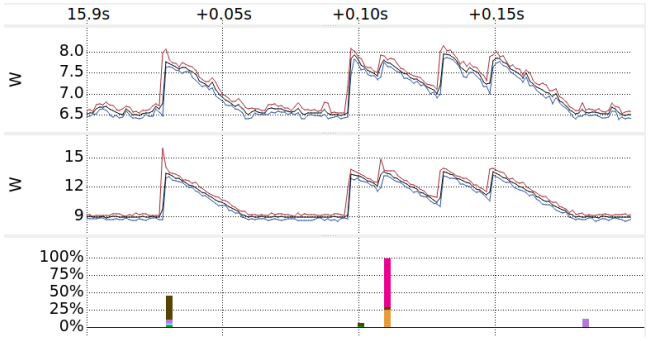
Processor	Intel Core i9-12900K
Performance cores / P-cores / Golden Cove	
Nr. cores / hardware-threads	8 / 16
Frequency range (selectable)	0.8 GHz to 3.2 GHz
Turbo frequency	up to 5.1 GHz*
Efficiency cores / E-cores / Gracemont	
Nr. cores / hardware-threads	8 / 8
Frequency range (selectable)	0.8 GHz to 3.2 GHz
Turbo frequency	up to 3.9 GHz*
Intel® UHD Graphics 770	
Nr. Execution Units (EUs)	32
Frequency range (selectable)	0.3 GHz to 1.5 GHz
Turbo frequency	up to 1.55 GHz
Uncore frequency scaling (UFS)	0.8 GHz to 4.7 GHz
Hardware Performance States (HWP)	disabled
RAPL Power Limit	4095 W
RAM	2 × 16 GiB DDR5-4800
Motherboard	Gigabyte Tech. Co. Z690 UD
Operating system	Ubuntu 22.04
Kernel version	5.19.1 & 6.2.0
Power meter	ZES LMG450
Accuracy	0.07 % + 0.25 W

*or 5.2 GHz on one core or with Turbo Boost Max Technology 3.0

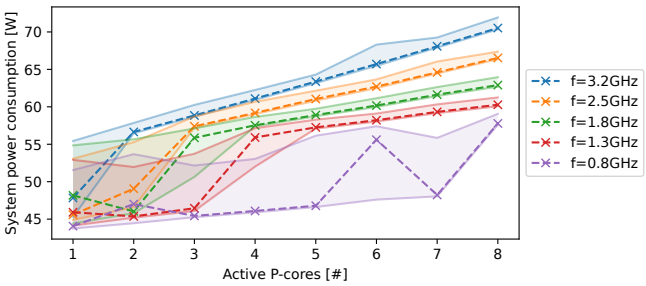
C ADDITIONAL DATA ON POWER MONITORING

On an idling system, additional noise can increase the average power consumption significantly, as we show in Figure 13a. Often, when a CPU on an idling system gets active, the processor power consumption (monitored with an LMG670 and 20 kSa/s) increases from 9 W to 13.3 W (with spikes of more then 15 W). Simultaneously, the power consumption of the 5 V rail increases from 6.5 W to 7.9 W. This increased power consumption only decreases gradually and affects the power consumption of the system even after the CPUs are already back in an idling phase.

We see the multiple power levels mentioned in Section 7.2 for an idling system as well where some cores use the POLLing idle routine instead of the default C10. Figure 13b shows the minimal, maximal, and median power on an idling system with a number of cores POLLing instead of using C10 and an uncore frequency of 3.2 GHz for different core frequencies. The power data is retrieved over a 10 s time period with `metricq-summary`. System power consumption grows linearly with the number of used cores if the system power consumption is above 54 W (with some noise induced outliers). There is a gap if the frequency and number of cores are too low to reach this threshold. There seems to be another linear relation at the minimal values, just on a lower level. Median and average are located randomly between these two lines.



(a) Unused cores can significantly increase power consumption once they become active due to OS noise. Vampir visualization of a lo2s trace of the idling test system. Top: Power consumption (5V rail). Mid: Power consumption (12V processor). Bottom: activity of CPUs. A short usage of CPUs increases the power consumption from 9 to 15 Watt, which is then reduced only gradually. Not only CPU activity can lead to power spikes, e.g., at offset +0.15s there is a spike without any CPU core being active.



(b) System power consumption, depending on the number of cores in OS poll loop and the core frequency. Minimal, median and maximal power values plotted as lines. The uncore frequency is set to 3.2 GHz.

Figure 13: Detailed look at different power measurement aspects.

Developing Index Structures in Persistent Memory Using Spot-on Optimizations with DRAM

Xingsheng Zhao*
xingsheng.zhao@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

Prajwal Challa
vxc5208@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

Chen Zhong
chen.zhong@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

Song Jiang
song.jiang@uta.edu
University of Texas at Arlington
Arlington, Texas, USA

ABSTRACT

The emergence of persistent memory (PMem) is greatly impacting the design of commonly used data structures to obtain the full benefit from the new technology. Compared to the DRAM, PMem's larger capacity and lower cost make it an attractive alternative for hosting large data structures, such as indexes of in-memory databases, especially for those that require data persistency. However, simply using existing index structures in the PMem can be unexpectedly inefficient for three reasons. (1) Index accesses are composed of small writes and reads. (2) Each small write is required to come with expensive fence and flush operations. And (3) PMems usually prefer large accesses for high performance with their internal block-like access designs despite being byte-addressable. For example, Intel Optane DC PMem has a 256-byte access unit (XPLine), leading to significant read/write amplification for small accesses.

In this work we systematically study a series of techniques, including application-managed write-buffering, read-caching, and out-of-place updates and their synergistic effect on performance of some representative indexes (hash table, B+ tree, and skip list) designed for PMems. We then apply the knowledge obtained from this investigation into the design of a high-performance PMem index, named Spot-on tree (SPTree), that facilitates applications to selectively cache read-intensive components of an index and to buffer written data to index structure, while providing crash consistency and quick recovery upon crash. Compared to the state-of-art indexes, SPTree provides up to 2X and 4X higher write and read throughput, respectively.

CCS CONCEPTS

• **Information systems** → **Data access methods**; • **Theory of computation** → **Concurrent algorithms**; • **Hardware** → **Non-volatile memory**.

*now at Google

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645032

KEYWORDS

B-tree, hash table, non-volatile memory

ACM Reference Format:
Xingsheng Zhao, Prajwal Challa, Chen Zhong, and Song Jiang. 2024. Developing Index Structures in Persistent Memory Using Spot-on Optimizations with DRAM. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645032

1 INTRODUCTION

The memory/storage hierarchy, which consists of multiple levels including CPU cache, DRAM, and block devices, such as SSDs and HDDs, has been stabilized for decades. Accordingly, the principal management designs for data across its levels, such as set-associative CPU caches, page-based virtual memory, and block-based read cache and write-back buffer, are well established by carefully considering individual devices' performance characteristics to maximize the hierarchy's performance. However, with the emergence of byte-addressable persistent memory (PMem), such as Intel Optane DC persistent memory [1] and NVDIMM (with flash storage) [4, 5], a new level/tier was introduced into the hierarchy. We contend that it is necessary for the DRAM to serve as a cache level for the PMem to boost its effective performance. And it is a challenging task for accessing of data structures on the PMem to be accelerated with an efficient use of DRAM.

1.1 DRAM as a Cache of PMem

Like DRAM, the PMem is a byte-addressable memory device that can be directly accessed via load and store instructions. However, its performance gap with the DRAM can be still substantial. As the Optane PMem has a more consistent performance behavior, we use it as a representative of PMem hereafter. The performance of the Optane PMem is lower than that of DRAM by 2-3X or more in terms of its latency and throughput [12]. In the meantime, the PMem can have a 5-10x increase of per-module capacity over the DDR4 DRAM while its per-GB price is 2X-5X cheaper than DRAM. Therefore, placing the PMem underneath the DRAM in the hierarchy has the potential of taking advantage of both DRAM's high performance and PMem's larger capacity and lower cost. Indeed, the Optane PMem has a memory mode in which DRAM acts as a cache for data accessed on the Pmem. Though few details are known on how Intel

CPU's IMC (Integrated Memory Controller) enables this transparent caching, a constraint in the use of the mode highlights the challenge in the management of this memory level. The constraint is that the persistent memory has to be treated as a volatile memory, or data in the memory cannot survive a system restart, though the capability of retaining data on the PMem is one of its major features that are attractive to many potential users.

1.2 DRAM as a Write Buffer of PMem

When DRAM is used as PMem's cache, it not only should be used as a read cache, but also must be used as a write buffer to enable the write-back policy for three reasons. First, DRAM is faster than the PMem by around 2-3X for reads and is around 4-6X for writes. Second, recent studies have shown that the Optane PMem has an access unit of 256 bytes to the memory's media (in comparison, NVDIMM with flash has a 0.5∼4KB page-size access unit). Any write smaller than the size leads to a write amplification and reduction of effective throughput [3]. For example, with 64-byte random writes the PMem's throughput is reduced to about 1/4 of its peak one [24]. Third, to ensure crash consistency for written data, an application may have to frequently use expensive fence and flush instructions between writes to the PMem, which may significantly degrade write efficiency. DRAM can be used as a write buffer to coalesce writes and then flush data in the buffer to the PMem. By doing so multiple random writes to the PMem may be transformed into one big sequential write that aligns with PMem's access granularity so as to receive high throughput from the PMem. It is tempting to use the large DRAM space to make up for the PMem's shortcoming.

However, it is difficult to retain the PMem's persistence feature by simply using a DRAM as its cache/buffer. Some fundamental challenges exist due to some unique characteristics of the DRAM-PMem layers. Unlike CPU-caches/DRAM layers, the DRAM as a cache for the PMem presents unique challenges. This is primarily due to the fact that the CPU cache is much smaller than the DRAM and can be managed with an affordable cost with hardware supports. Instead, the gap between sizes of DRAM and the PMem is much less significant. Furthermore, the CPU cache can be battery/capacitor protected to keep dirty data in it from being lost upon a power failure. But the large DRAM cache/buffer is unlikely to have such a support. This presents a dilemma about the use of DRAM as a cache/buffer for the PMem. Using DRAM as the PMem's buffer enables a large write-back space while simultaneously presents a risk of losing data due to DRAM's non-volatile nature.

Long before emergence of the commercially available PMems, the advantages of non-volatile memory (NVM) have been recognized. Significant efforts have been made to migrate popular in-DRAM data structures to the NVM with optimizations of their crash-consistent implementations by efficiently using fence and flush operations [10, 13, 18, 25]. Among the data structures, indexes, such as hash table and B+ tree, are the most performance-critical due to their frequent and on-the-critical-path accesses. In the meantime, they are vulnerable to performance loss due to their frequent use of pointers and accessing of small data. One of their major use cases is the development of key-value (KV) stores in an NVM with the objectives of high write throughput, low read latency, low DRAM footprint, and rapid recovery and restart after a system crash. An unexploited opportunity in the efforts is to leverage DRAM to conduct spot-on caching and buffering for individual components of an index where the performance gap is large. Unlike system-level caching and buffering services that are indiscriminately covering the entire DRAM and NVM levels with a fixed space unit (e.g., disk block), this proposed spot-on approach is customized to the structure of an index. Therefore, performance-limiting operations, such as pointer chasing and random writes, in the PMem can be made efficient with dedicated caches/buffers for individual components such as inner nodes and leaf nodes in a tree structure.

In this paper, we made several important contributions on the improvement of major data structures' performance in the PMem.

- We identify unique issues and opportunities in the efforts of bridging the performance gap between DRAM and the PMem in the memory hierarchy.
- We propose and evaluate a comprehensive and complementary set of techniques to enable DRAM's spot-on caching and buffering for the PMem.
- Using the techniques we design and implement a new index structure, Spot-on Tree (SPTree), that can efficiently leverage limited DRAM space to reduce read and write amplifications and the cost for maintaining crash-consistency in the PMem.
- We introduce a unique design of the write-ahead-logging (WAL) technique to enable instant service resumptions.
- Experiment results show that SPTree can achieve up to more than 4X throughput and less than 1/5 latency over some of the state-of-the-art indexes, such as Fast&Fair and PACTree.

2 BACKGROUND AND RELATED WORK

In this section, we describe the background of Intel Optane Persistent memory, which is selected in this study as the representative of the PMem technology, and some related works on persistent indexes to motivate this study.

2.1 The PMem

The Optane Persistent memory can be configured in two different modes. The first one is named `Memory Mode`, in which the CPU considers the Optane as its main memory and uses the entire DRAM as its cache. While the cache is so large and the caching unit is page, it would be too expensive and thus infeasible to keep dirty pages in the DRAM persistent. Therefore, in this mode the Optane PMem does not provide persistency at all. And the PMem becomes essentially a larger but slower DRAM. The second mode is the App-Direct mode. In this mode, the PMem works as a persistent device. A file system supporting Direct Access (DAX) provides direct access to the persistent memory, and bypasses the file system block I/O. In this mode, a program is exposed to the aggregate space of the DRAM and the PMem as well as their distinct performance characteristics.

Though the PMem is byte-addressable, the physical media access granularity is 256 bytes (XPLine) [3, 23, 24]. Any non-contiguous writes of data smaller than the XPLine size requires a read-modify-write operation, leading to write amplification and reduced effective memory bandwidth. To reduce write amplification, the PMem employs a write-combining buffer to merge adjacent small writes. However, its size is only 16KB [3, 22]. Writes to memory addresses

with a coverage larger than this scope cannot benefit from the feature. It is unknown how to flexibly and dynamically set up buffers in the DRAM to overcome this PMem's limitation on write performance. Similarly, the CPU cache is often not sufficient to improve its read performance on par with that of DRAM because of the PMem's large size and programs' weak access locality.

2.2 Persistent Indexes

There are mainly two kinds of persistent indexes. One is persistent hash table, such as CCEH[18], Level hashing[25], and Dash[16]. And the other is persistent range indexes whose keys are sorted and support range search, including FastFair[10], FP-Tree[9], and PACTree [13].

Persistent Hash Table. Being aware of higher write cost in a persistent memory, existing works on development of persistent hash tables mostly focus on reducing number of writes in the memory, such as PFHT [6], level hashing [25], and CCEH [18]. In particular, PFHT is a cuckoo hashing variant that limits number of displacements to only one to reduce memory writes during service of a write request. Level hashing adopts a two-level hash scheme so that each key can have three buckets as the candidates for its insertion, which helps reduce key relocations in the table and improve the load factor. CCEH is an extendable hashing to minimize performance impact of rehashing of the entire table. It organizes buckets into 64KB segments to leverage fast sequential access in a persistent memory. It then uses the linear probing strategy in its search for either a key or a bucket with empty slot(s). A successful insertion requires only one memory write. While these works reduce number of writes in the memory, each write takes place directly at the memory's location determined by the hash table design. While the locations are spread out in the memory, the performance loss due to write amplification in the PMem can be significant.

Persistent Range Indexes. There have been some works on designs of B-trees for persistent memory. FastFair[10] is a lock-free-read B+-tree that avoids expensive copy-on-write and logging to tolerate transient inconsistency. BzTree[2] relies on the Persistent Multi-word Compare-And-Swap(PMwCAS) primitive to implement a lock-free tree. FP-Tree[9] stores inner nodes of the tree in the DRAM to achieve high performance. However, it has to scan all nodes on the persistent memory to reconstruct the inner nodes after a reboot or a crash before resuming its service. PACTree [13] employs a persistent trie index as its internal nodes and asynchronously updates the internal nodes using a structural-modification-operation (SMO) log. In these works, the DRAM is not leveraged to buffer writes to accommodate the PMem's block access unit. Even if reads are accelerated by partitioning an index structure between the DRAM and the PMem, access of in-PMem leaf nodes, where a majority of data is stored, is not accelerated. Furthermore, the in-DRAM sub-structure has to be entirely rebuilt during a recovery process before new accesses can be resumed. This compromises a promise made for the persistent memory, which is the instant service resumption.

3 THE THREE SPOT-ON TECHNIQUES

In this section, we analyze three representative persistent index data structures and propose a series of techniques leveraging DRAM to optimize the PMem performance in a spot-on fashion. Three data structure are CCEH[18] (a persistent hash table), FastFair [10] (a persistent B-tree) and P-Skiplist (a persistent skiplist) [20], as shown in Figure 1. The set of techniques are *Buffering*, *Out-of-place-update*, and *Caching*. They are intended to be applied in a sequence, which is the *Buffered, Out-of-place merging, and then Caching (BOC)* design.

While each of the techniques is a well known one and has been extensively practiced, this work focuses on their customized use on specific index structures in a spot-on manner. Rather than keeping and managing the cached/buffered data at one centralized space in the faster memory, the BOC approach distributes the space to the carefully selected index components to maximize its utilization. It doesn't rely on a replacement algorithm to determine where the cache space should be allocated. While effectiveness of a replacement algorithm is often limited and the algorithm's time and space overhead for managing small pieces of data can be significant, the BOC approach effectively addresses this issue with its customized cache space distribution design.

The key BOC takeaways are that, the read-modify-write in a 256-byte XPLine and long latency of random access in the PMem are the fundamental performance bottleneck for persistent index data structures. It suggests that a persistent index should (1) buffer small writes in the DRAM then update them to the PMEM in a batch manner, and use (2) out-of-place merging to reduce flush&fence, which is a well-understood performance bottleneck. When there are frequent pointer chasing accesses in the indexes, one should (3) cache the search path in the DRAM to minimize the search latency. This represents a holistic design approach in recognition that any of the individual techniques could not adequately recoup PMem's performance loss due to its performance idiosyncrasies.

3.1 Technique One: Buffering

The PMem differs from the DRAM in several ways. One of them is that there is a mismatch between CPU cache-line access granularity (64 bytes) and the 3D-Xpoint media's access granularity (256 bytes) [23]. To overcome this mismatch, it has a write-combining buffer (16KB) to merge small writes and reduce write amplification [23]. Given the small size, it is hard to exploit the locality to frequently hit the buffer.

For writes in the persistent index data structures, most of them are small writes, such as insertion of a new record (e.g., a 16-byte key-value pair), structural modification operations (SMO) in a B+-tree, which lacks access locality considering the small write buffer in the PMem. Consequently, most of the small writes result in the read-modify-write operations within the PMem and leads to high write amplifications and reduced effective memory bandwidth.

In this study of the spot-on buffering technique, we allocate small write buffers in the DRAM and assign them individually to selected components in the index structures (see Figure 1). Specifically, for CCEH each write buffer is as large as half of the segment size. For FastFair each leaf node has a write buffer of half of the leaf node size. P-Skiplist only creates write buffers for nodes whose height is higher than two in the skip list. And each buffer can store at most 16 KV pairs. The operations in the indexes are not changed except that writes into the selected components are first admitted to their respective write buffers. When a buffer becomes full, all key-value

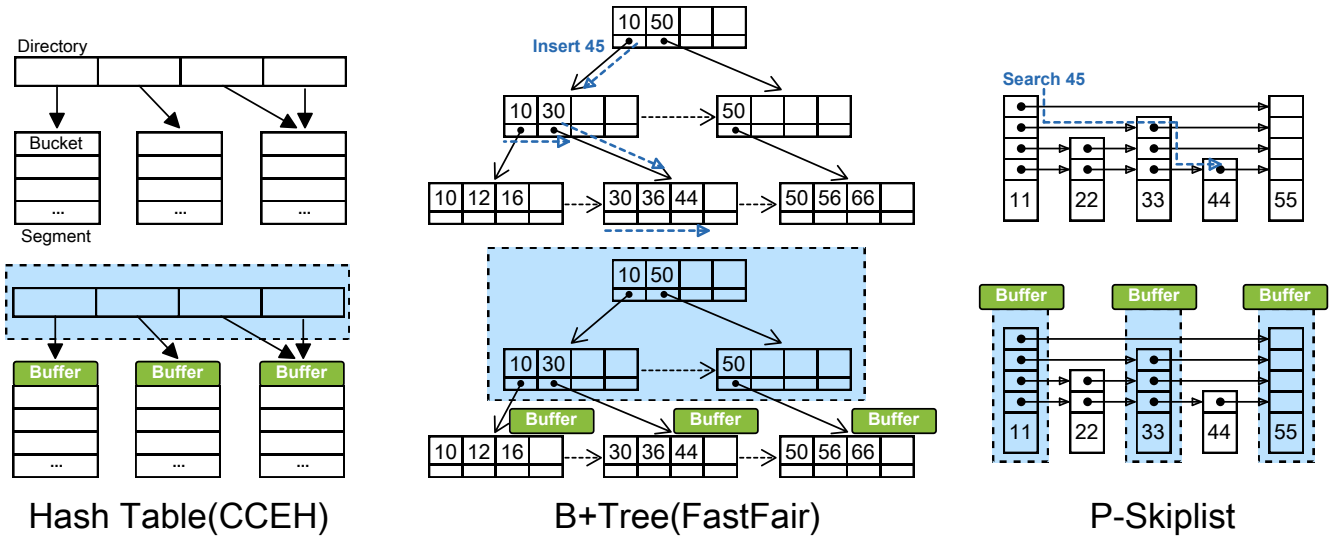


Figure 1: Adding write buffers to the three selected index data structures

(KV) pairs in it are written to its corresponding index component in a batch. This is actually a merging of KV pairs in the buffer for a batched write.

We then run a benchmark to insert 120 million KV pairs (8-byte key and 8-byte value) into each of the indexes with uniformly distributed and non-redundant keys. In the experiment we measure amount of raw data written to the PMem's media using the ipmctl tool [11]. Figure 2 shows the amount of the raw data with or without the write buffers for each of the indexes. It shows that the total amount of the data written to the PMem is reduced by up to four times. The reductions are especially higher for CCEH and FastFair where writes to the last-level nodes are buffered. This observation demonstrates that spot-on buffering can effectively enable batched writes and much reduced write amplification by greatly improving spatial access locality.

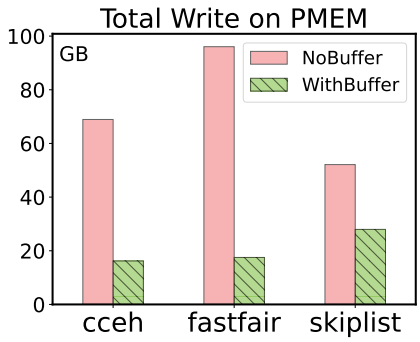


Figure 2: Total amount of raw data written to the PMem's media after insertion of 120 million 16-byte key-value pairs to the three indexes with and without the buffers (with one thread and without using the WAL log).

3.2 Technique Two: Out-of-place Update

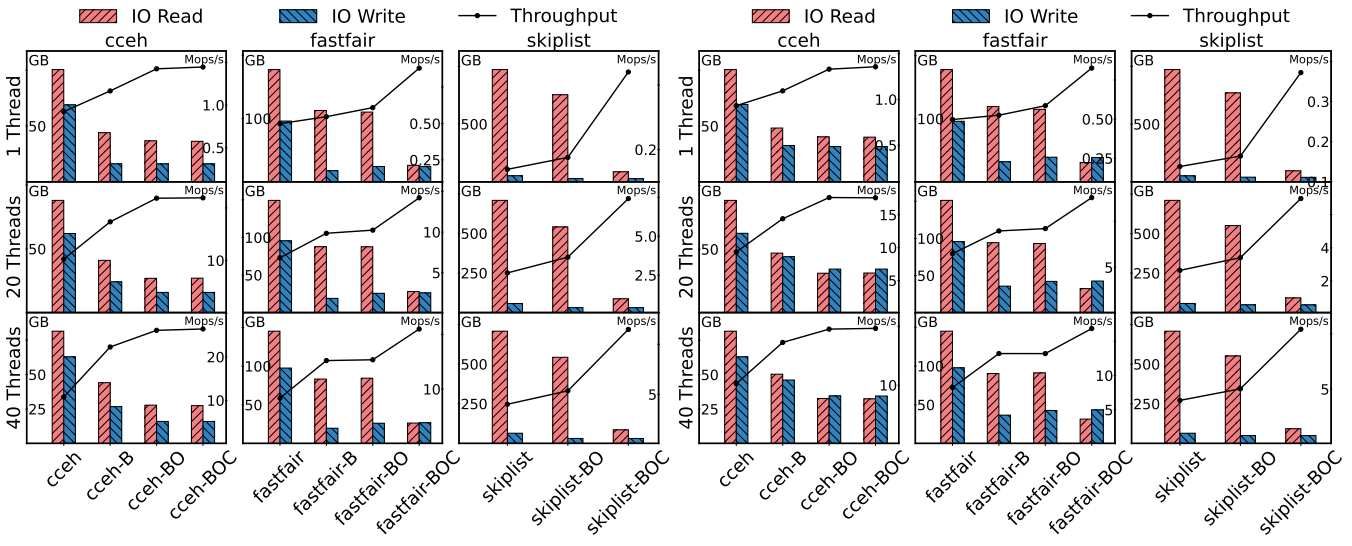
There is a potential issue with adding write buffers to the persistent indexes. When we write the KV pairs in a buffer to the PMem in an in-place manner (i.e., updating the index component in place), we may need to repeatedly add flush&fence within a small contiguous range of the PMem space (such as a segment in CCEH, or a leaf node in FastFair) to enforce its crash consistency in case of power failure. It has been reported that reading a recently flushed cacheline after fence instructions could experience much higher latency as the read has to wait the flush to complete [23].

To analyze the effect of flush&fence on the buffer merging, we implement an out-of-place buffer merge in CCEH and FastFair. Since P-Skiplist's KV pairs are stored separately in the linked list nodes, the insertion has been in the out-of-place manner. So we do not consider P-Skiplist in this experiment.

In CCEH, when a write buffer of a segment is full, we first copy the segment from the PMem to the DRAM. We then merge the KV pairs in the write buffer into the DRAM copy and write it back to the PMem in a newly allocated space. Finally, we atomically change the directory pointer to this new segment. For FastFair, we employ a similar approach for its leaf nodes. During the merging, a new leaf node is created to hold all of the KV pairs from both the old leaf node and write buffer. Then the new leaf replaces the old leaf by atomically changing the parent pointer and sibling's left 'next' pointer. An out-of-place merge is essentially conducted in the background. Thus, flush&fence operations for individual KV pairs are avoided. Another benefit of this technique is that the current buffer and the index component are still available for serving read requests during the merge.

Figure 3 shows the amount of raw data read from and raw data written to the PMem media with each of the three indexes when 120 million KV pairs are inserted. It also shows the insert throughput. Different number of threads are used (1, 20, and 40 threads at the top, middle, and bottom, respectively, of the figure). For a particular index, the three optimization techniques are incrementally applied.

The figure shows that even when the read and write amounts remain unchanged, the throughput of CCEH and FastFair with only



(a) Without WAL **(b) With WAL**

Figure 3: Amount of raw media read/write (denoted as "I/O Read/Write") and throughput for inserting 120 million key-value pairs (8-byte key and 8-byte value) with and without using a Write-Ahead-Log (WAL) log. For each index, the three techniques are incrementally added ('B' for "Buffering", 'O' for "Out-of-place Update", and 'C' for "Caching"). For example, "cceh", "cceh-B", "cceh-BO", and "cceh-BOC" refer to the CCEH index without any optimizations, with Technique 1, with Techniques 1 and 2, and with Techniques 1, 2, and 3, respectively.

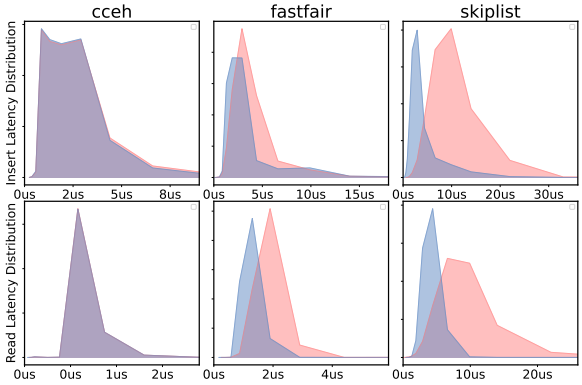


Figure 4: Distributions of insert/read latency for an index without caching (the red area) and with caching (the purple area).

Buffering ("cceh-B" and "fastfair-B") becomes higher after applying "Out-of-place Update" ("cceh-BO" and "fastfair-BO"). This improvement is attributed to the removal of the flush&fence overheads.

3.3 Technique Three: Caching

For index data structures whose operations contain many random reads (in the form of pointer chasing), the performance could be bottlenecked by slow random reads in the PMem. Read latency on the Optane PMem is considerably higher than DRAM (about 2X-3X) because reads need to fetch data from the 3D-Xpoint media, which has longer media latency [3]. Meanwhile, most of the pointer chasing operations happen in the internal nodes whose total size may account for only a small portion of the entire data structure size. So it is worth enabling caching individually for the internal nodes in the DRAM to boost the lookup performance.

In this experiment study, we cache internal nodes in all three indexes (directory in CCEH, inner nodes of FastFair, nodes with height higher than two in P-Skiplist) and measure their insert/read latency for 120 million insert requests followed by 120 million read requests.

As shown in Figure 4, caching internal nodes have little effect on CCEH's latency as it has only one pointer chasing operation for each access, which is for locating the segment from the directory. However, we observe significant latency improvement for FastFair and P-Skiplist as they need to do intensive pointer chasing (random reads) across the internal nodes before finding a target node, indicating that caching is a necessary optimization for many-level indexes.

3.4 Put them Together

With all these three techniques available (*buffering, out-of-placed Update, and then caching(BOC)*), we apply them to three indexes one by one and measure the performance improvement after insertion of 120 million key-value pairs. In the experiment, we experiment with two cases: one without the WAL log, and another one with the WAL log. With the log, all data written to the leaf nodes will also be added to the log in the PMem. Note that writes to the log are fully sequential and incur little amplification. As shown in Figure 3, the write buffer not only reduces write amplification but also read amplification. This is not a surprise since during the buffer merging, the PMem data to be merged has been cached in

the DRAM. The cost of reads is amortized via batching. Meanwhile, the benefits of out-of-place update (merging) are consistent from the single thread execution to the multi-thread execution. The throughput with the addition of "Out-of-place Update" ("cceh-BO" and "fastfair-BO") is always higher than that without the optimization ("cceh-B" and "fastfair-B", respectively). Finally, caching the internal nodes not only reduces latency of pointer chasing, but also removes a large amount of the PMem's media read ("Read I/O") during search ("fastfair-BOC", "skiplist-BOC"), significantly increasing the PMem's effective memory bandwidth. Furthermore, the performance advantages remain with the increase of thread count and with addition of the WAL log.

4 THE DESIGN OF SPTREE

Understanding the benefits of the three spot-on optimization techniques, we propose SPTree (SPot-on Tree), a DRAM-PMem hybrid persistent tree index that uses spot-on caching and buffering to address a sequence of issues challenging performance of in-PMem indexes, including long latency of pointer chasing, write amplification due to mismatch between KV pair size and PMem's media access unit, and quick recovery after a crash. As shown in Figure 5, SPTree consists of three layers (the top, middle, and bottom layers).

4.1 The Three Layers in SPTree

The Top Layer. As shown in Figure 5, the top layer caches the internal nodes of the tree in the DRAM, which is an in-DRAM index that walks a key to its corresponding middle level node and leaf node. SPTree uses the DRAM index to address the high latency issue of pointer chasing in the PMem. That is, all search operations in the SPTree take place in the DRAM top layer before a target leaf node in the bottom layer is reached. We modify the ARTree [14] as the DRAM top-layer. Meanwhile, the index has a PMem backup, organized as a FastFair B+-tree, for a quick recovery after a reboot or a system crash. It is updated asynchronously by the background threads to move the slow updates on the PMem off the critical path. Each time when a leaf node splits or merges, the new leaf node's indexing information is synchronously updated in the DRAM index, and sent to the background threads and then asynchronously propagated to the in-PMem index. If some of the updates have not yet been reflected in the PMem index when a system crash happens, we can still recover the missing information by checking the possible smallest key(the low ley or 'lkey') and the largest key(high key or 'hkey') in the leaf nodes. More details on the recovery are in Section 4.6. Note that updates to the in-PMem index are much less frequent than those to the leaf nodes.

The Middle Layer. The middle layer consists of Mnodes (middle-layer nodes) in the DRAM. Each leaf node in the PMem has its corresponding Mnode in the DRAM. The Mnode records the range of keys in the leaf node (the smallest key (lkey) ... largest key(hkey)). This key range serves the purpose similar to that of the version number that allows non-blocking reads during a write. When a search reaches to an Mnode, the search key is checked against its key range. If the search key does not belong to this Mnode, it indicates that concurrent insertion or deletion operations have caused splitting or merging of Mnodes (and also leaf nodes), and we need to go back to the top layer for a retry.

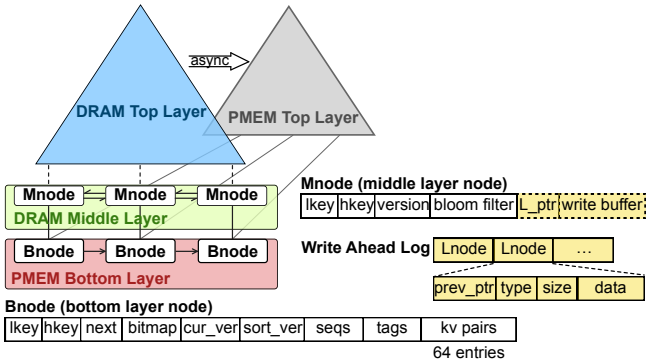


Figure 5: The SPTree tree structure

An Mnode also stores a Bloom filter which remembers all the keys in its leaf node. Each insert updates the Bloom filter in its corresponding Mnode. The Bloom filter is used to filter out most of the point queries for non-existing keys before they reach the in-PMem leaf nodes. An Mnode can also be configured with a write buffer, which buffers all the newly inserted KV pairs (up to 14) to reduce write amplification. Once a buffer is full, all the KV pairs in the buffer will be merged to the leaf node.

The Bottom Layer. The bottom layer stores the leaf nodes, or Bnodes (bottom layer nodes), in the PMem. All Bnodes are organized as a singly linked-list. Each Bnode can store up to 64 KV pairs. The leaf node groups fingerprints (hash values) of its keys as key tags in an array for a quick preliminary search by using SIMD instruction (`_mm_cmpeq_epi8_mask`). Meanwhile, it applies a two-phase insertion approach to reduce the use of flush and fence instructions. In the first phase, KV pairs and their tags are batch written to the Bnode followed by a flush&fence. Only in the second phase will they be validated by setting the bitmap in the node followed with another flush&fence. In this way, only two flush&fence operations are needed for multiple insertions. In this way, SPTree minimizes the use of flush&fence to improve efficiency.

4.2 Concurrency Control

SPTree relies on `Optimistic Lock Coupling` [15] for concurrency control. An optimistic lock consists of a lock and a version counter (packed into 8 bytes). For writers, the optimistic lock provides exclusive access that allows only one writer at a time. Upon an unlock, the lock is released and the version number is incremented by one atomically. For readers, they do not acquire the lock. Instead, they wait until the write lock is freed. Then they compare the version number before and after reading the value. If the reading changes, they will retry until a consistent reading is reached. Both of the DRAM top layer and the middle layer use the optimistic lock.

4.3 Search Operation

Search for a given key is the most frequently used operation on the index. It is not only used to service a read request, but also has to be employed at the beginning of the execution of every insert/update/delete/lookup/scan operation. A search operation first traverses the DRAM top layer to look for the largest key that is smaller or equal to the search key. This largest key is the lkey

in an Mnode. Then the search key is compared with the hkey in the Mnode to ensure the search key is in the key range of this Mnode. If it is in the range, the search continues to the Bnode in the PMem. Otherwise, the search travels in the middle layer, which is a doubly linked list, to locate the target Mnode. During the search operation, the PMem top layer is not accessed because it is only used for recovery.

4.4 Insert/Update/Delete/Lookup/Scan

Insert. The thread servicing an insert request first searches the DRAM top layer to locate an Mnode whose key range includes the inserted key. Then the write lock is acquired on the Mnode. If the Mnode has a write buffer, the KV pair is inserted into the write buffer, and also appended to a write ahead log (WAL) for an after-crash recovery. Otherwise, the thread inserts the KV pair directly into the corresponding Bnode on the PMem. In the first case, if the write buffer is full, SPTree writes all the KV pairs in a batch to the Bnode, and clears the write buffer. SPTree uses a two-phase insertion in the batch write operation. (1) All the KV pairs and their tags are written to the Bnode followed by a flush&fence. (2) The bitmap indicating valid pairs is set and the version (*cur_ver*) advances by one. Meanwhile, a split is conducted if necessary. Finally, the Bloom filter in the Mnode is updated for the inserted key.

Update. Update is similar to insert during its search for the target Mnode. After a target Mnode is found, the Bloom filter is checked to see if the key exists. If not, it returns immediately. Otherwise, it acquires the write lock. If there is a write buffer with the target Mnode, it tries to update the KV in the buffer, and write the new value to the WAL if the update is a success. Otherwise, we check the corresponding Bnode for the update.

Delete. For delete, after the thread reaches a target Mnode, it checks the Bloom filter first to avoid unnecessary access to the PMem. If the Bloom filter returns true, and the target Mnode has a write buffer, it removes the delete key in it if it exists in the buffer. Additionally, it adds a DELETE tombstone record in the write buffer. The DELETE tombstone is also written to the WAL log to record this deletion. In this case, the key will be physically deleted later during a compaction when a KV pairs in the write buffer are written back. Note that this tombstone is necessary to prevent follow-up reads for this key from mistakenly returning the deleted KV pair from the corresponding Bnode instead of a "none". If the Mnode doesn't have a write buffer, it will immediately try to remove the key from the Bnode.

Lookup. Lookup operation in SPTree is a lock-free read. By applying the optimistic lock, the thread checks the version in the target Mnode before and after the read. If the two versions match, it means no inserts take place during the read and it can return the value safely. Otherwise, the lookup will start over.

Scan. Scan operation first looks for the target Mnode whose 'lkey' is equal to or smaller than the lower bound of the scan range and whose 'hkey' is larger than the lower bound. It then checks whether a reconstruction of the sequence array (`'seqs'`) is required by comparing `'cur_ver'` and `'sort_ver'`. If `'cur_ver'` is larger than `'sort_ver'`, it means new inserts happen after the last reconstruction of the `'seqs'`, it then acquires the write lock and sorts the keys in the Bnode and stores them in the ascending order in the `'seqs'` array. Meanwhile, it sets `'sort_ver'` to `'cur_ver'` to indicate this Bnode is ready for scan. When there are keys stored in the Mnode write buffer, a batch write of the KV pairs is triggered before scanning. During the scan of a Bnode, the keys that fall in the scan range are collected in a buffer. Then the values of `'cur_ver'` before and after the scan are compared. If they are equal, those buffered KV pairs are appended in the output array. Otherwise, it retries the scan in this Bnode. Scan continues to the sibling Bnode until the output array is full or the key is out of scan range. The operation's pseudocode description is at Algorithm 1.

Algorithm 1: Scan

```
1  Function Scan(start_key, result_size, results):
       /* Find mnode that has lkey <= start_key <= hkey */
2      mnode = LocateMNode(start_key)
3      bnode = mnode.bnode
4      while Output results array not full do
5          ReadLockGuard(bnode)
6          if mnode write buffer is not empty then
7              WriteLockGuard(bnode)
8              Flush write buffer;
9          if bnode.cur_ver > bnode.sort_ver then
10             WriteLockGuard(bnode)
11             Sort(bnode)
12             bnode.sort_ver = bnode.cur_ver
13         Collect keys falling in scan range
14         bnode = bnode.next
```

4.5 Split and Merge

When a Bnode (leaf node) is full, SPTree conducts a split, which creates a new Bnode along with its new Mnode. Then a `[lkey, Mnode pointer]` mapping is inserted to the DRAM top layer while a `[lkey, Bnode pointer]` is sent to the background thread to asynchronously update the PMem top layer. A split operation involves four steps. (1) The writer first acquires the lock for the current Bnode and its next sibling. (2) It then allocates a new Bnode, moves the right half of the KV pairs from the full node to the new Bnode, and sets the `'next'` pointer in the full node pointing to the new node. These three operations are atomically conducted using the leak-free PMem allocator (such as Intel PMDK's `pmemobj_alloc()`) to prevent memory leak. (3) Then `'bitmap'` and `'hkey'` in the full Bnode are modified to remove the split-out keys. And (4) finally, the mapping information is updated in the DRAM top layer and propagated to the PMem top layer.

When a delete operation detects that keys in two adjacent nodes become fewer than half of a node's capacity, a merge operation is triggered. (1) The thread first acquires the two nodes' write locks. (2) Then it shifts the KV pairs in the right Bnode to the left Bnode using the two-phase insertion (set the KV pairs first, then the bitmap). (3) Next it modifies the left Bnode's hkey as the right Bnode's hkey, marks the right Bnode as deleted in its `'cur_ver'` and drops the Bnode. (4) Finally, it updates the mapping in the top layer.

4.6 Recovery and Crash Consistency

Since SPTree uses the DRAM top layer and the DRAM middle layer to provide service, these two layers need to be reconstructed after a reboot or a power failure. In a normal reboot, all the updates in the

DRAM top layer have been propagated to the PMem top layer. And the PMem top layer stores the pointers to the Bnodes in the PMem. During a recovery, SPTree can quickly collect all the pointers to the Bnodes by scanning the PMem top layer, which is only around 2% of entire index size. Then it rebuilds the Bnode's Mnode and the DRAM top layer simultaneously.

If this recovery is after an unexpected crash, we need to fix the incomplete state. If a crash happens before a split's Step (2) completes, SPTree can recover to the state before the split (guaranteed by the leak-free allocator). If the crash happens after Step 2 and before Step 3, then a dummy leaf node is linked to the bottom layer. An example incomplete status looks like this: [`split node, lkey..10, hkey..100`] -> [`dummy node, lkey..50, hkey..100`] -> [`next node, lkey..100, hkey..200`]... . The dummy node will have a `'lkey'` lower than its previous node's `'hkey'`. If we find a dummy node during the recovery, we fix it by dropping the dummy node. If the crash happens during Step (3) (`'hkey'` is set and the `'bitmap'` has not been set), then the split node will have keys that do not belong to it. This can be fixed by ignoring those out-of-range keys during the next split. If the crash happens before Step (4) completes, we only need to fix the missing mappings in the top layer. We can identify the missing mappings by comparing the adjacent Mnode's `'lkey'` and `'hkey'` after a non-crash recovery. An Mnode's `'hkey'` is designed to be the same as its next sibling's `'lkey'`. If not, we scan the linked list from the current Mnode's Bnode to recover the missing Mnodes.

If a crash happens before a merge's Step (2) competes, SPTree can recover to the state before the merge because the shifted KV pairs have not been exposed by the bitmap. If the crash happens before Step (3), the shifted KV pairs can be filtered out using the `'hkey'`. If the crash is after setting the `'hkey'` and before dropping the right node in Step (3), an example incomplete status looks like this: [`merge-left node, lkey..10, hkey..100`] -> [`'merge-right node, lkey..50, hkey..100`] -> [`'next node, lkey..100, hkey..200`]. The merge-right node has an `'lkey'` lower than merge-left node's `'hkey'`. We fix this by dropping the merge-right node during the recovery. If the crash happens before Step (4), the dummy mapping will point to a deleted Bnode. We then delete this mapping during the recovery.

Regarding data consistency in Bnodes, all the KV pairs that have not completed the second phase (setting the bitmap) will not be exposed. So any partially updated pairs will not be visible.

If an Mnode has its write buffer, the KV pairs in the buffer can be recovered by replaying the WAL log. However, any KV pairs in its corresponding Bnode cannot be used to serve look requests until all of the lost pairs in the write buffer are recovered from the log. These pairs are scattered in the log. It would be too slow to resume the service of requests if one had to wait for the scan of the entire log to be completed.

4.7 Instant Service Resumption with a Structured WAL

The WAL approach has been widely used for preventing data loss in systems such as LevelDB [8, 19], RocksDB [7, 21], and Kangaroo [17]. Existing use of the technique is simplistic: data in new inserts, updates, and deletes that are sent to the in-DRAM data

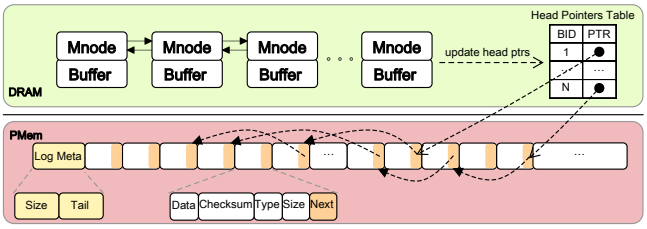


Figure 6: The structured WAL log.

structures are also appended at the tail of a log in the persistent storage. This approach enables efficient I/O by writing data sequentially. However, the side effect is that the data are not well organized and a particular data cannot be efficiently located for quick access. Maintenance of a separate index for the log is way too expensive to be practical. This is especially problematic with the persistent memory as an instant restart is often expected.

To this end, SPTree proposes a structured WAL log that allows keys to be searched and retrieved from the log without a time-consuming sequential scan. This technique enables (almost) instant resumption access to the SPTree index before a full recovery is completed.

In the design, the conventional WAL log is enhanced by organizing linked lists, each consisting of KV pairs that belong to the same write buffer, in the log. As shown in Figure 6, each record in the log contains a `'next'` pointer, which is the offset of the record in the log about the KV pair that belongs to the same the buffer and has been appended to the log immediately prior to this one. SPTree maintains a table of head pointers in the DRAM, each pointing to the most recently appended record in the log that belongs to a buffer. Each time a buffer's KV pair is appended, the buffer's current head pointer becomes its `'next'` pointer and the `'head'` pointer is updated to the new pair. In this way, without introducing additional writes to the log, the WAL log becomes a structured one containing multiple linked lists. After a crash, SPTree can immediately service a read request by following the corresponding buffer's head pointer and searching on the linked list.

To maintain the structured log for high access efficiency, there are two issues to address. The first one is about placement of head pointers. For efficiency, the table of head pointers must be in the DRAM. In the meantime, it must survive a system crash. To this end, SPTree periodically checkpoints the table to the PMem along with the offset of the WAL's current tail. This offset represents the checkpoint position, indicating that the checkpointed table is up to date until this position. After a crash, the table is reloaded into the DRAM. And SPTree only needs to scan the log from its tail to the checkpoint position to find out all newer header pointers and update the in-DRAM table. Each record contains a 8-byte checksum to determine if it is a valid one. Accordingly, we can identify the log's tail.

The second issue is about size of the log. Once KV pairs in a buffer have been merged into the in-Pmem Bnode, their corresponding records in the log become obsolete and can be removed from the log as well as the linked lists. Otherwise, they would make the log and search on the log unnecessarily long. Therefore, SPTree needs to identify the true end record in a linked list and flag it. To this end,

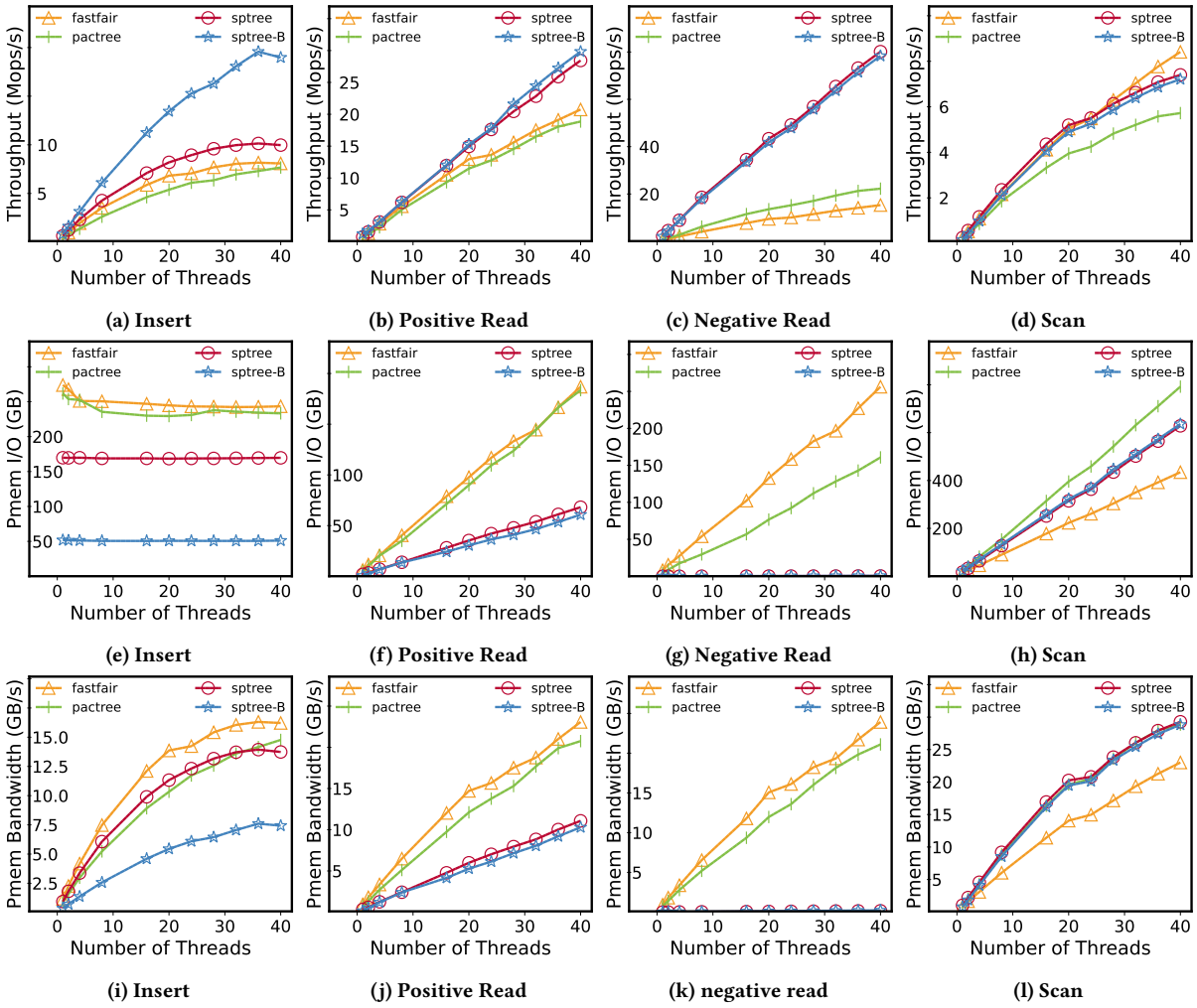


Figure 7: Throughput, PMem I/O volume, and PMem bandwidth with different types of requests.

after persisting a buffer's KV pairs to the BNode, SPTree writes a special record to the WAL flagged (at the 'Type' field shown in Figure 6) as the end of the buffer's linked list. A key search in the list will stop at the end record. Furthermore, to know the offset in the WAL log beyond which all records in the log can be collected as garbage, SPTree keeps track of the oldest end record, whose up-to-date offset is maintained in the PMem. When a space reclamation via on-log garbage collection is required, all records from this offset to the head of the log can be removed.

Thanks to the linked lists in the WAL, access to the SPTree index can instantly become available. In the meantime, a background thread scans the log to recover the write buffers. When the buffers are fully restored, access to the log for reading KV pairs is no longer required.

5 EVALUATION

In this section, we experimentally evaluate SPTree by comparing it with several state-of-the-art B+-Tree for persistent memory, including FastFair [10] and PACTree [13]. As a sorted index that supports range search, we do not compare SPTree with the hash-based indexes, such as CCEH.

5.1 Experiment Setup

In the experiments, we use 16-byte KV pairs. All the threads in an experiment are pinned to one socket using `numactl`. SPTree's DRAM footprint is about 12% of the total size of the PMem bottom layer when the write buffers are not used. If all of the Mnodes in the SPTree have been assigned with write buffers (denoted *sptree-B*), the DRAM footprint is about 33% of the PMem bottom layer. All the experiments are run on a server with an Intel Xeon Gold 6230 20-core processor, 64GB DRAM and 6 × 128GB Intel Optane DC.

5.2 The Throughput

To evaluate the performance of the trees, we conduct extensive experiments, including insertions of new KV pairs (Insert), reading keys in the indexes (Positive Read), reading keys not in the indexes (Negative Read), and range queries (Scan). Experiment results are shown in Figure 7. In each experiment, different number of threads

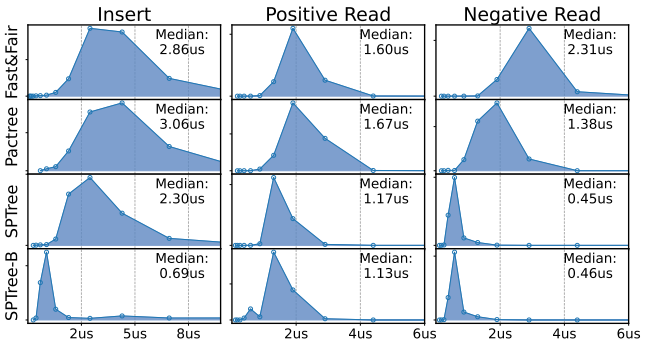


Figure 8: Latency comparison

(from 1 to 40 threads) are used. For *Insert* each thread sends 120 millions/number-of-threads requests. For *Positive Read* and *Negative Read*, and *Scan* each thread sends 10 million requests. Figure 7 reports throughput of the trees (number of requests serviced per second) and the corresponding raw PMem media access amount (I/O volume) . The I/O volume represents all read/write data amount on the Optane PMem's media, including amplified I/O due to existence of its 256B access unit. It is measured with `ipmwatch`, available in the Intel VTune Amplifier tool. It also shows the PMem's bandwidth, which is the raw I/O volume per second.

Insert. As shown in Figure 7a, SPTree consistently outperforms the others for insert performance. This is mainly because in the other two trees the pointer chasing in the internal nodes causes large read amplifications, resulting in reduced effective memory bandwidth. As shown in Figures 7e and 7i, though the bandwidth of FastFair and PACTree during the insertion is equal to or even higher than that of SPTree, their high I/O bandwidth actually results in the lower throughput. When the write buffers are enabled for SPTree (sptree-B), the total I/O volume is reduced by 3X. This is mainly because of the reduced write amplification as well as the read amplification as explained in Section 3.4.

Positive Read. We see up to 25% throughput improvement for SPTree over the others. This advantage mainly comes from the reduced I/O during search in the internal nodes. As shown in Figure 7f, SPTree's I/O volume is only about 1/3 of the others. When write buffers are used, we see slightly performance improvements over the one without buffers. This is because the write buffers also function as read caches for `Positive Read`. Hence, a small portion of the request does not reach the Bnodes in the PMem. That's why I/O volume of sptree-B is smaller, as shown in Figure 7f.

Negative Read. SPTree has up to 4X throughput improvement compared with the other two trees, as shown in Figure 7c. This is because it caches the existing keys in the Mnodes' Bloom filter, which filters out most of the unnecessary PMem accesses during the negative read. As shown in Figures 7g and 7k, there is almost no PMem I/O and we barely read from the PMem. All the saved bandwidth can be used to service other requests.

Scan. Both SPTree and PACTree use an indirection array for sorting keys in the leaf node. This strategy comes with a cost of higher read amplification compared with the physically sorted KV pairs in FastFair. As shown in Figure 7h, SPTree and PACTree have higher I/O volume during the `Scan`. However, thanks to the low

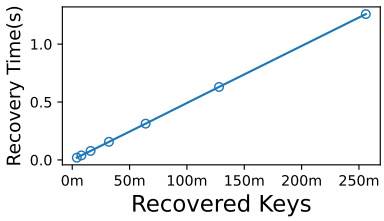


Figure 9: Recovery time to rebuild the DRAM top and middle layers.

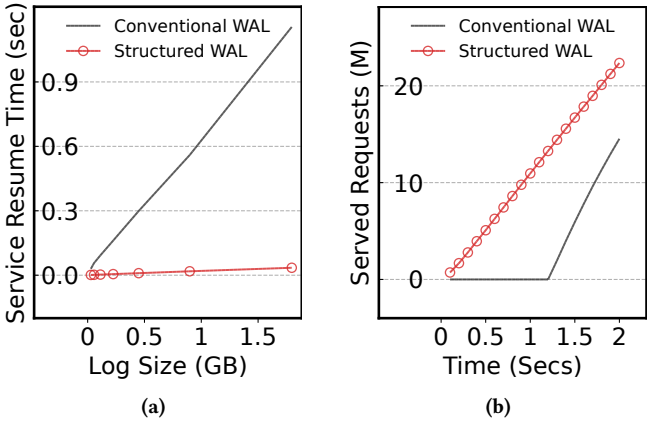


Figure 10: (a) Service resumption time for varying size of records in the log. (b) Number of lookup requests served within a certain time period right after a restart with a log of 1.8GB.

overhead of the DRAM top layer, SPTree's scan performance is only 10% lower than FastFair, while PACTree is 30% lower.

5.3 The Latency

In this section, we evaluate the read/write latency of the three trees. We use 20 threads to write 120 million KV pairs. Each thread sends 4 million read requests.

As shown in Figure 8, SPTree always has the lowest latency among the trees in all of the workloads (`Insert, Positive Read,` and `Negative Read`). By caching the internal nodes in the DRAM, the random pointer chasing cost in the PMem is greatly reduced. Meanwhile, using the Bloom filters also helps SPTree to avoid the access to the PMem for non-existing keys. In the experiment, after 120 million KV pairs are inserted, the false positive rate is around 5%. It is worth noting that when write buffers are used, the insert latency for SPTree (SPTree-B) reduces by 3X compared with the one without using write buffer, which is at the cost of DRAM for only around the 30% of the PMem footprint.

5.4 Recovery

If the write buffers for SPTree are not used, SPTree can start servicing the requests without the DRAM top and middle layers as the PMem top layer can be used. In the meantime, it rebuilds the DRAM layers in the background and then puts it into service. As shown in Figure 9, the time to rebuild the DRAM top and middle layers are

negligible (1 second for 250 million keys) because it simultaneously rebuilds the DRAM layers based on the PMem top layer.

If the write buffers are used and an unexpected crash takes place, the structured WAL log is employed to quickly resume request service. Figure 10a shows the time it takes to resume its service (for the first lookup request to be served after a crash) with different size of the log (from the oldest end record to the log tail). As shown, using the structural log, SPTree reduces the resumption time to almost 0, much smaller than that using the regular WAL log. Figure 10b shows the number of lookup requests that can be served within a given period of time right after a restart on a log of 1.8GB. As shown, without the structured WAL log no requests can be served until after about 1.4 seconds from the restart. While instant-on resumption is expected for any persistent memory, only SPTree makes it possible with its unique structured WAL design.

6 CONCLUSIONS

In this paper, we propose to use the spot-on DRAM caching and buffering techniques to efficiently address the performance issues at the critical places in an index structure where performance is compromised due to the PMem's performance characteristics. We systematically studied the benefits of the techniques to understand their individual and combined impacts on optimization of in-PMem index structures. Empowered by this understanding, we further introduce SPTree, a persistent ordered tree designed for high-performance systems. Adopting a holistic approach, SPTree leverages the techniques in its design supported by a novel structured WAL log to deliver the instant-on user experience. Experiments show that SPTree minimizes the PMem I/O traffic and achieves 2X to 4X improvement of access performance over the state-of-the-art PMem tree index designs in terms of both throughput and latency.

The source code of the SPTree is available at *https://github.com/hansonzhao007/buflog*.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by the US National Science Foundation under grant CCF-2313146.

REFERENCES

[1] [n. d.]. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (jan 2018), 553–565. https://doi.org/10.1145/3164135.3164147

[3] Hanyeoreum Bae, Miryeong Kwon, Donghyun Gouk, Sanghyun Han, Sungjoon Koh, Changrim Lee, Dongchul Park, and Myoungsoo Jung. 2021. Empirical Guide to Use of Persistent Memory for Large-Scale In-Memory Graph Analysis. In *39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021*. IEEE, 316–320. https://doi.org/10.1109/ICCD53106.2021.00057

[4] Renhai Chen, Zili Shao, and Tao Li. 2016. Bridging the I/O performance gap for big data workloads: A new NVDIMM-based approach. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783712

[5] Renhai Chen, Zili Shao, Duo Liu, Zhiyong Feng, and Tao Li. 2019. Towards Efficient NVDIMM-Based Heterogeneous Storage Hierarchy Management for Big Data Workloads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 849–860. https://doi.org/10.1145/3352460.3358266

[6] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. 2015. Revisiting Hash Table Design for Phase Change Memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (Monterey, California) *(INFLOW '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/2819001.2819002

[7] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[8] S. Ghemawat and J. Dean. 2011. LevelDB. https://github.com/google/leveldb

[9] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *ACM sigmod record* 29, 2 (2000), 1–12.

[10] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang

[11] intel. [n. d.]. ipmctl. https://github.com/intel/ipmctlïż¿.

[12] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[13] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 424–439.

[14] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[15] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2933349.2933352

[16] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302* (2020).

[17] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 243–262. https://doi.org/10.1145/3477132.3483568

[18] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. {Write-Optimized} Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.

[19] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[20] William W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676. https://doi.org/10.1145/78973.78977

[21] Facebook RocksDB Team. 2021. A Persistent Key-value Store for Fast Storage Environments. http://rocksdb.org

[22] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2021. Characterizing and Modeling Nonvolatile Memory Systems. *IEEE Micro* 41, 3 (2021), 63–70. https://doi.org/10.1109/MM.2021.3065305

[23] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 488–505. https://doi.org/10.1145/3492321.3519556

[24] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a key-value store for optane persistent memory. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 194–209. https://doi.org/10.1145/3447786.3456237

[25] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo

What does Performance Mean for Large Language Models?

Jane Hillston
jane.hillston@ed.ac.uk
University of Edinburgh
Edinburgh, UK

ABSTRACT

In the last decade there has been a significant leap in the capability of foundation AI models, largely driven by the introduction and refinement of transformer-based machine learning architectures. The most visible consequence of this has been the explosion of interest and application of large language models such as ChatGPT. This is one exemplar of how a foundation model trained on a huge amount of data can be specialised for particular task, often by a phase of reinforcement learning with human feedback.

Within the AI community "performance" of such systems is generally taken to mean how well they respond to their users on characteristics such as accuracy, verifiability, and bias. Performance analysis usually considers both the responsiveness of a system to its user and the efficiency and equity of resource use. These foundation models rely on massive amounts of resource but there appears to have been little work considering how to understand the resource use or the trade-offs that exist between how the system responds to users and the amount of resource used.

In this talk I will present initial ideas of what it could mean to develop a framework of performance evaluation for foundation models such as large language models. Such a framework would need to take into consideration the distinct phases of operation for these models, which broadly speaking can be categorised as *training*, *generating* and *fine-tuning*. Evaluating the trade-off between user interests and resource management will require the identification of suitable metrics. The *resources* in these systems can be more than simply compute, storage, bandwidth; data and even human resources also play crucial roles in training and fine-tuning. I will discuss all these topics.

CCS CONCEPTS

• **Hardware** → *Power estimation and optimization*; • **Software and its engineering** → **System modeling languages**; • **Information systems** → **Retrieval efficiency**; **Information extraction**; • **Computing methodologies** → **Natural language processing**; **Natural language generation**; **Machine learning algorithms**; **Model development and analysis**.

KEYWORDS

Performance evaluation, large language models, efficient use of resources, user responsiveness

ACM Reference Format:
Jane Hillston. 2024. What does Performance Mean for Large Language Models?. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3629526.3649130

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3649130

InstantOps: A Joint Approach to System Failure Prediction and Root Cause Identification in Microservices Cloud-Native Applications

Raphael Rouf
raphaelr@my.yorku.ca
York University
Toronto, Ontario, Canada

Mohammadreza Rasolroveicy
roveicy@ibm.com
IBM Canada Lab
Markham, Ontario, Canada

Marin Litoiu
mlitoiu@yorku.ca
York University
Toronto, Ontario, Canada

Seema Nagar
senagar3@in.ibm.com
IBM Research
Bangalore, India

Prateeti Mohapatra
pramoh01@in.ibm.com
IBM Research
Bangalore, India

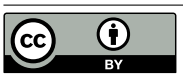
Pranjal Gupta
Pranjal.Gupta2@ibm.com
IBM Research
Bangalore, India

Ian Watts
ifwatts@ca.ibm.com
IBM Canada Lab
Markham, Ontario, Canada

ABSTRACT

As microservice and cloud computing operations increasingly adopt automation, the importance of models for fostering resilient and efficient adaptive architectures becomes paramount. This paper presents `InstantOps`, a novel approach to system failure prediction and root cause analysis leveraging a three-fold modality of IT observability data: logs, metrics, and traces. The proposed methodology integrates Graph Neural Networks (GNN) to capture spatial information and Gated Recurrent Units (GRU) to encapsulate the temporal aspects within the data. A key emphasis lies in utilizing a stitched representation derived from logs, microservices events(e.g. Image Pull Back Off, PVC Pending), and resource metrics to predict system failures proactively. The traces are aggregated to construct a comprehensive service call flow graph and represented as a dynamic graph. Furthermore, permutation testing is applied to harness node scores, aiding in the identification of root causes behind these failures.

To evaluate the efficiency of `InstantOps`, we utilized in-house data from the open-source application Quote of the Day (QoTD) as well as two publicly available datasets, MicroSS and Train Ticket. The F1 scores obtained in predicting the system failures from these data sets were 0.96, 0.98, and 0.97, respectively, beating the state-of-the-art. Additionally, we further evaluated the efficiency of root cause analysis using MAR and MFR. These results also outperform the state of the art.



This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, UK
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645047

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; **Feature selection**; *Regularization*; • **Information systems** → **Clustering**; • **Networks** → Network performance analysis.

KEYWORDS

Cloud Computing, Anomaly Prediction, Resource Overload, Machine Learning, Root Cause Analysis

ACM Reference Format:
Raphael Rouf, Mohammadreza Rasolroveicy, Marin Litoiu, Seema Nagar, Prateeti Mohapatra, Pranjal Gupta, and Ian Watts. 2024. InstantOps: A Joint Approach to System Failure Prediction and Root Cause Identification in Microservices Cloud-Native Applications. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645047

1 INTRODUCTION

With the growing use of cloud computing, enhanced system stability and reliability have become critical due to increased dependence on cloud servers for data storage and processing. Anomalies–unusual deviations from expected system behavior, can signify potential security breaches or malfunctions, posing risks to system availability and integrity [2, 13]. Anomaly Detection and prediction are crucial in server and cloud environments for the immediate identification of abnormal activities, thereby preventing these issues from escalating into serious problems [7, 35].

The automatic detection and prediction of anomalies and failures in microservice applications have garnered significant interest among researchers in recent years. Historically, anomaly detection has been a focus of extensive study. Techniques such as statistical methods, clustering, and rule-based systems have been employed for this purpose [1, 5, 9, 11, 12, 24, 25, 28]. However, as modern systems have become more interconnected and intricate, there has

been a palpable shift towards leveraging machine learning models to achieve enhanced efficiency and accuracy in anomaly detection [6, 7, 18, 26].

To gain a holistic perspective of the system's health, research has particularly emphasized various single-modal data sources:

Traces in [16, 32, 38]: Traces offer insights into the flow of requests and the interactions among different microservices. Anomalies in traces, such as unexpected latencies or failed requests, can be revealing. For instance, a deviation from a usual 10-millisecond response time to a full second indicates potential concerns.

Logs in [4, 8, 23, 29]: Logs, the textual records generated by applications and infrastructure components, provide a comprehensive context about events, errors, and warnings. A surge in error messages or emergent warning patterns that deviate from the standard can pinpoint anomalies.

Resource and Performance Metrics in [19, 21, 27, 33]: Monitoring various metrics, including CPU usage, memory allocation, network activity, response time, and throughput, offers pivotal insights into the system's performance and health. Unforeseen spikes, drops, or inconsistencies in these metrics might be indicative of performance bottlenecks, inefficiencies, or other health-related concerns within a service.

However, recent studies [14, 15, 36] suggest that relying solely on single-modal data for failure localization may not be adequately efficient. One primary reason is that a single failure can have cascading effects on multiple facets of microservices. Such a failure might manifest itself in various modalities, leading to multiple anomaly patterns. For instance, a database slowdown might result in both extended response times (observable in traces) and a surge in error logs.

Furthermore, there exist certain failures that might not be evident in specific modalities. If detection methods rely solely on one modality, they risk missing out on these anomalies. A classic example might be an internal logic error in a service that doesn't necessarily result in increased resource usage or evident trace anomalies but could still produce erroneous outputs.

Given these complexities, there's a growing consensus in the research community about the necessity of a multi-modal approach, combining data from various sources to create a holistic and more accurate picture of the application's health.

In this paper, through the introduction of `InstantOps`, we augment the existing state-of-the-art research [14–16, 34, 36] by fusing multimodal data source including logs, traces, resource metrics, and microservices events (e.g. Image Pull Back Off, PVC pending) in a time series format to predict system failures at the service (node) level. In this paper, we characterize a 'node' as an individual service within the micro-service architecture. Specifically, our objective is to predict, identify, and localize the nodes in the system that might be responsible for imminent system failures. By pinpointing and localizing the problematic node within the microservice architecture, we can strategize appropriate remediation measures, such as node scaling or resource configuration.

We stitch the multi-modal data for a time window in a novel fashion, where traces act as a thread. We serialize logs, traces, and metrics for each time window. In this process, we intertwine traces as the stitching thread, serializing logs, and metrics for each time window. This serialized data aids in constructing a dynamic dependency graph that delineates the interconnections among services—wherein nodes represent services and their interactions are depicted as edges.

Moreover, we enrich this dependency graph by merging both logs and resource metrics as attributes assigned to its nodes. This augmentation enhances the representation of the system's spatial features at specific points in time. To analyze and comprehend these spatial features, we employ Graph Neural Networks (GNN), recognizing that a system's failure isn't a single-point occurrence but a lifecycle that gradually progresses toward a system crash. Understanding the temporal evolution of these spatial features is imperative. To address this, we utilize Gated Recurrent Unit (GRU) models, acknowledging the importance of capturing the system's changing dynamics over time.

Our focus lies not only in training a multi-modal GNN-GRU model for predicting system failures but also in utilizing the learned node scores to localize the root cause. This approach allows us to repurpose the trained model for the specific task of root cause analysis. By leveraging the insights gained from failure prediction, particularly the node scores, we aim to guide the root cause analysis process. We believe that the predictive learning captured in failure prediction can significantly aid in root cause localization, thereby aligning our methodologies for a more comprehensive understanding of system behavior and failure analysis.

The contributions of this paper can be summarized as follows:

- Novel System Failure Prediction: We introduce a unique approach to predict system failures by fusing multimodal observability data in a novel fashion wherein we overlay the logs and metrics data as node attributes in the graph constructed using the traces.
- We propose to capture both temporal and spatial aspects in effective failure prediction.
- Innovative Root Cause Analysis Method: We use the multimodal system failure prediction model for pinpointing the root causes of failures at the node level.
- Comprehensive Experimental Study: We conduct an extensive experimental analysis to evaluate the efficiency of our model in terms of both prediction accuracy and root cause analysis. This assessment is based on two open-source and one proprietary dataset derived from open-source microservice systems.

The remainder of the paper is organized as follows: Section 2 describes the motivation behind our experiments and study. In Section 3, we discuss recent related works in failure predictions and anomaly detection in microservices, specifically focusing on the use of multimodal datasets. Section 4 details our methodology and elaborates on our GNN-GRU based algorithm for detecting failures in microservices. Section 5 delves into our approach to temporal failure prediction and root cause analysis including an explanation of the algorithms. In Section 6, we discuss the evaluation of our experimental study. Section 7 addresses the potential threats to validity. Finally, in Section 8, we conclude our findings and present potential avenues for future research.

2 MOTIVATION:

The analysis of system failures necessitates a thorough examination of diverse data modalities. Prior research indicates that relying solely on single-modal data is inadequate for capturing the intricacies of failure patterns, especially in microservice applications. In this paper, we utilize three distinct datasets: Anafusion's MicroSS [36], TraceRCA's Train Ticket [16], and an in-house dataset for the "Quote of the Day (QOTD)" open-source application. These datasets comprise different combinations of logs, traces, events, and metrics, depending on the specific experiment. For instance, the QOTD dataset incorporates logs, traces, events, and resource metrics, while the Train Ticket dataset emphasizes traces and metrics. To ensure a comprehensive evaluation, all datasets encompass records of every failure injection. Our primary objective is to determine the precision of system failure predictions when leveraging supervised learning. Distinctly from previous models, we incorporate edges into the system to systematically assess the impact on specific nodes. Furthermore, in this paper, the edges are constructed by the communications among the services in the microservice application.

The analysis of these interactions offers critical insights into which nodes require interventions, such as scaling, restarting, or remediation of various failures.

For system failure prediction, multiple neural network models are tested to ascertain the most efficient predictive methodology. The effectiveness of `InstantOps` is subsequently quantified using metrics including accuracy, precision, recall, and the F1 Score. To benchmark its performance, `InstantOps` is compared against two established methods, Anafusion and TraceRCA.

3 RELATED WORKS

Recent works have significantly leaned into exploring methods for anomaly and failure detection within microservices and cloud applications, leveraging various data-oriented and machine-learning approaches.

Zhao et al. [36] proposed a novel approach called AnoFusion for unsupervised failure detection through multimodal data for microservice systems. AnoFusion uses GTN to learn the correlation of the heterogeneous multimodal data and constructs a heterogeneous graph structure. Then, GAT is utilized to capture significant features and update the heterogeneous graph. Finally, GRU is used to predict the data pattern at the next moment. However, while AnoFusion looks at the system level to predict the failures of the microservice, we focus our attention on the service level to localise which nodes are the leading cause in predicting the system failures and take that into account in our system failure prediction. In Anafusion, the authors model one node, in this paper, we model the entire microservice application. Furthermore, our model is built with multiple data sources such as events from each microservices.

Zhang et al. [34] proposed an alternative method known as DiagFusion. This approach leverages multimodal data to enhance fault detection by employing advanced embedding techniques fastText and data augmentation. It constructs a dependency graph and employs a graph neural network to pinpoint the root cause and identify the type of failure.

Li et al. [16] proposed TraceRCA, a root cause microservice localization approach designed for trace anomaly detection and flagging abnormal traces to predict the root cause. Zhou et al. [38] presented MEPFL, a model designed to predict latent errors that possess the potential to precipitate failures, especially during the runtime in production environments of microservice applications. This approach is realized through the comprehensive analysis of system trace logs and the training of prediction models utilizing features distilled from these logs. In essence, MEPFL aims to empower developers by providing them with the capacity to identify and rectify faults before their manifestation as failures in a production setting. The model specifically addresses three predominant types of faults in microservice applications: system overload, memory leak, and sudden node crash. These fault types constitute nearly half of all microservice application faults, substantiated by existing empirical studies, underscoring the pivotal role and applicability of MEPFL in fortifying the reliability of microservice applications. The TraceRCA and MEPFL approaches are designed to take not only account traces but metrics, logs, and events using edge interactions of the nodes to localize the faults at the service level.

Lee et al. [15] introduced a novel approach named Hades, designed for detecting system anomalies in software systems. Hades seamlessly integrates heterogeneous data sources, including logs and metrics, to proficiently identify system anomalies. These anomalies, which encompass system failures, performance degradation, and other unanticipated behaviors, are detected promptly, thereby enabling system administrators to enact corrective actions swiftly. To facilitate the prediction of system anomalies, the authors employ a binary classification approach, wherein each data chunk is labeled as either 'normal' or 'anomalous.' Lee et al. [14] proposed Eadro, an approach for anomaly detection within microservices. Eadro operates by modeling the standard behavior of microservices and identifying deviations from this established normalcy. Specifically, it employs a deep neural network to derive discriminative representations of microservice statuses through multi-modal learning, compelling the model to apprehend fundamental features indicative of anomalies through multi-task learning. The model, which ingests multi-source monitoring data—including traces, logs, and Key Performance Indicators (KPIs)—generates a score for each microservice, reflecting the likelihood of an anomaly. A higher score indicates a greater probability of the microservice encountering an anomaly. Eadro's anomaly detection module is capable of identifying various types of anomalies, such as network-related issues, resource exhaustion, and software bugs. By pinpointing the root causes of anomalies, Eadro assists system administrators and developers in promptly troubleshooting and addressing issues. Research works similar to Hades and Eardo aim to detect system anomalies and failure, our intention is not to detect system anomalies but to predict the anomalies at the next minute based on the last 5 minute time window.

4 METHODOLOGY

In this section, we provide a systematic approach designed to predict system failures and facilitate subsequent root cause analysis using Graph Neural Networks (GNNs) in a Microservice application. We initiate with the standardization of features and then proceed

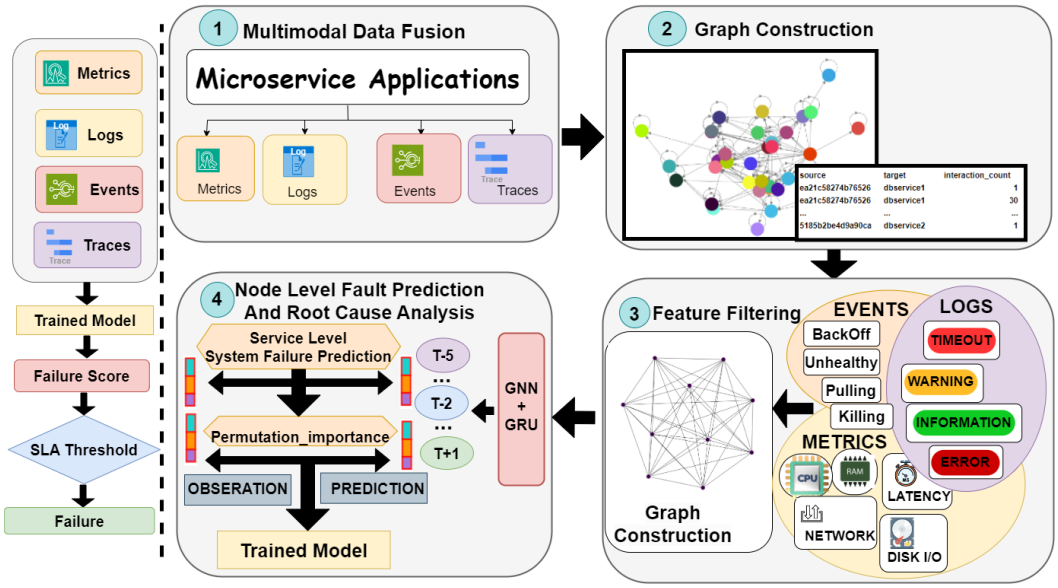


Figure 1: InstantOps Offline Training Model

to the systematic construction of a graph. In this representation, nodes symbolize system components, while edges correspond to the interactions between these components, quantified on a per-minute basis. To quantify the frequency of these interactions, we incorporate the concept of 'weights'.

When an interaction between or services is observed for the first time within a specified duration, we create an edge between the two corresponding nodes in GNN, and the weight of the edges is 1. If an edge between these nodes already exists within this timeframe, its weight is incremented by 1, marking an additional recorded interaction. Essentially, this weight serves as a metric, representing the number of interactions between two nodes within the designated interval. For example, if the weight of an edge between node A and node B is 10, it indicates that Service A interacted with Service B 10 times within the specified observation period.

Further, a GNN model is developed, integrating node feature information and graph topology to generate node embeddings. The mathematical framework defines the forward propagation and convolution operations within the GNN and Graph Convolutional Network (GCN) layers. Additionally, the model is extended to incorporate edge features, enhancing the representation of interactions. Temporal aspects and dependencies of the system are incorporated by using a Gated Recurrent Unit (GRU) model. During the training phase, the cross-entropy loss function is employed to optimize model parameters to minimize the difference between predicted outputs and actual labels. We predict system failure when the entire application crashes or if the application receives 500 errors more than 99% of the time in the $t+1$ time windows. Finally, the model's predictive accuracy, a quantifiable metric that measures the proportion of correct predictions relative to the total, is used to evaluate its ability to predict system failures and enable root cause analysis.

As shown in Figure 1, the workflow of `InstantOps` is divided into multiple stages:

First, we preprocess and serialize the multi-modal data-set within a controlled time-frame. For each data source, we define time windows with the same size. The sequence of the time windows defines the stream of graphs for our model. Next, we construct the Graph Neural Networks using edges and nodes. For each time window, we add the features to the nodes in the graph. The layers of the GNN aggregate information from neighbors in a graph. We then filter features by selecting important keywords from logs and events that serve as features and correlating them with resource metrics. The first layer takes the features, while the second layer accepts the output from the first and produces additional graph neural network features.

After processing the node features with the GNN layers, the node embeddings are further refined through a GRU cell. For failure prediction, we consider both the features and temporal aspects across different timestamps to predict system failures in the subsequent time moment, which in our case is $t+1$. This layer can capture temporal dependencies in the node embeddings produced by the GNN layers. While GRU models are traditionally used for sequence data, in our approach, we use it as an additional transformation for node embeddings.

We further elaborate on our steps below:

4.1 Data Preprocessing and Graph Construction

Data normalization is conducted to maintain consistency among the dataset features, expressed mathematically as:

$$x_{\text{std}} = \frac{x - \mu}{\sigma}$$

where x is the raw feature, μ is the mean, and σ is the standard deviation of the feature across all data points.

A graph G is then defined as $G = (V, E)$, where V is a set of nodes representing microservices, and E is a set of edges representing

interactions between services. The adjacency matrix $A \in \mathbb{R}^{n \times n}$ encapsulates the number of interactions, where $A_{ij} = 1$ if an interaction exists between services represented by nodes v_i and v_j, and 0 otherwise. In other paper, authors define Edges $e_{ij} \in E$ can also be defined based on metrics between nodes:

$$e_{ij} = f(\text{logs}_{ij}, \text{metrics}_{ij})$$

where f represents a function determining interactions between nodes based on logs and metrics.

4.2 Log Parsing

To predict system failure and their respective causes, serialization of the logs is an integral part of our work. The architectural design of this project is inspired by our previous work, BERTOps [8], which utilizes the Drain [10] to extract structured information from raw log data by clustering log lines into templates based on their structural similarity and BERT [3] for building an encoder based Large Language Model (LLMs) for the log data. By fine-tuning the pre-trained BERTOps model on labeled data from downstream tasks such as log classification and fault category prediction, BERTOps can learn to accurately represent log data and perform various log analysis tasks with high accuracy. While BertOps aims to classify a log line, in this paper, we classify each node and extract vital features from the logs, such as the number of errors a node receives within the specific time window. For instance, given the log structure "[2023-08-22T17:20:12.083] [Error] default - [418241] Quote request timeout", we extract the following features: timestamp, node id, number of errors. These features will be added to the nodes for a corresponding time window.

4.3 GNN Model Formulation

Leveraging the ability of GNNs to capture localized graph structures and enable accurate predictions, the formulation and forward propagation within a GNN layer include the transformation and aggregation of node features across successive layers, adhering to:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

where:

- $H^{(l)}$ is the matrix of node features at layer l,
- $\tilde{A} = A + I$ includes the adjacency matrix A fortified with self-loops I,
- $\tilde{D}$ represents the degree matrix of $\tilde{A}$,
- $W^{(l)}$ denotes the weight matrix at layer l,
- σ embodies a non-linear activation function.

This structural formulation of GNN simultaneously ensures the preservation of spatial relations between nodes and facilitates an iterative enhancement of node representations through the aggregation of neighboring information. This mechanism is instrumental in decoding intricate patterns, which are crucial for the predictive analysis of system failures, and provides a robust foundation for subsequent root cause analysis.

4.4 Graph Neural Network Model Development

The GNN model leverages both node feature information and topological structure. The forward propagation of a GNN model is often expressed as:

$$h_{v_i}^{(l+1)} = \sigma\left(\sum_{v_j \in N(v_i)} W^{(l)} \cdot h_{v_j}^{(l)}\right)$$

where $h_{v_i}^{(l)}$ represents the feature vector of node v_i at layer l and $N(v_i)$ is the set of neighbors of node v_i.

The operation in a Graph Convolutional Network (GCN) layer can be expressed as:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

where $\tilde{A} = A + I$ and $\tilde{D}$ is the degree matrix of $\tilde{A}$.

Enhancing the GCN model to include edge features, the node representation becomes:

$$h_{v_i}^{(l+1)} = \sigma\left(\sum_{v_j \in N(v_i)} W^{(l)} \cdot h_{v_j}^{(l)} + U^{(l)} \cdot e_{ij}\right)$$

where $U^{(l)}$ is a trainable weight matrix for edge representations at layer l.

5 TEMPORAL FAILURE PREDICTION AND ROOT CAUSE ANALYSIS

In this section, we discuss the approaches for failure prediction and root cause analysis.

5.1 Temporal Failure Prediction

In our approach, we integrate the power of Graph Neural Networks (GNN) with Gated Recurrent Units (GRU) to predict temporal failures. The GNN captures the spatial structure of the data by operating on a graph, encapsulating local neighborhood information of each node through iterative feature aggregation from its neighbors. The propagation in GNN is steered by the adjacency matrix $\tilde{\mathbf{A}}$ and its diagonal degree matrix $\tilde{\mathbf{D}}$, formalized as:

$$\mathbf{X}^{(l+1)} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}^{(l)}\mathbf{W}^{(l)}\right)$$

The Gated Recurrent Unit (GRU) captures the temporal dynamics across sequences. The GRU discerns sequential patterns using its intrinsic update and reset gates. The vital computations within the GRU include:

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \quad \text{(Reset gate)}$$

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \quad \text{(Update gate)}$$

$$\hat{h}_t = \tanh(W x_t + U(r_t \odot h_{t-1}) + b) \quad \text{(New potential hidden state)}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t \quad \text{(Actual hidden state update)}$$

For each sequence and its respective time step, the GNN first imbibes spatial information derived from the graph. The conjoined features from the GNN for every time step are subsequently funneled into the GRU, which modifies its hidden state according to its preceding state and the contemporary input. The culminating output of the GRU forms the basis of the prediction. The associated loss is computed relative to the true labels, followed by a backward

pass to refine the model parameters. Upon training, this model can be employed to prognosticate failures for imminent time steps.

Algorithm 1 Temporal Failure Prediction using GNN with GRU

procedure GNN_TemporalFailurePrediction(nodes, edges, y)
  Standardize *nodes*
  Map node names to integers in *edges* and *nodes*
  Convert *nodes*, *edges*, *y* to *data*
  Initialize GNN-GRU model with 2 GCN layers and a GRU layer
  for each epoch **do**
    for each *time window* in train_loader **do**
      Get temporal sequence of *nodes* for the *time window*
      for each time-step t **do**
        $\mathbf{X}^{(l+1)} \leftarrow \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}^{(l)}\mathbf{W}^{(l)}\right)$
        $r_t \leftarrow \sigma(W_r x_t + U_r h_{t-1} + b_r)$
        $z_t \leftarrow \sigma(W_z x_t + U_z h_{t-1} + b_z)$
        $\hat{h}_t \leftarrow \tanh(W x_t + U(r_t \odot h_{t-1}) + b)$
        $h_t \leftarrow (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t$
      Use h_t for prediction
      Compute loss with CrossEntropy
      Backward pass
      Update model parameters
    Evaluate on the validation set
    Compute metrics (Accuracy, Precision, Recall, F1 Score)
    Predict failure for $t + 1$ using current model state
  return Trained GNN-GRU model

Table 1: Conjoined features from the GNN

Features	Traces	Metrics	Logs	Events
# of Node Interactions	✓			
CPU Usage		✓		
Memory Usage		✓		
Disk I/O		✓		
Network I/O		✓		
5XX Errors			✓	
2XX Requests			✓	
4XX Errors			✓	
API Latency			✓	
CrashLoopBackOff				✓
ImagePullBackOff				✓
NodeNotReady				✓
PodScheduled				✓
NodeReady				✓
Unhealthy				✓
VolumeMount				✓
Failed (Image Pull)				✓
Resource Constraints				✓

Figure 2 depicts the model architecture of a system designed for predicting system failure at the node level in a microservice application. This system employs a hybrid approach, integrating a Graph Neural Network (GNN) with a Gated Recurrent Unit (GRU) model. The GNN component is responsible for capturing the interactions among microservice nodes, effectively learning from the topological structure of the microservices network. Each node in the GNN represents a microservice, and the edges reflect the interactions between these services.

The GRU part of the model handles the temporal aspects of the system's features, such as resource utilization and error rates, which are critical for understanding the state of the system over time. The GRU's ability to maintain information across time steps makes it particularly suitable for this task, as it can recognize patterns that precede a system failure.

The figure illustrates nodes representing different layers and operations within the neural network, such as convolutional layers (conv1.lin.weight, conv2.lin.weight), which are used in processing nodes interactions, and GRU components (gru_cell.weight_ih, gru_cell.weight_hh), which are adept at handling temporal aspects. The AccumulateGrad nodes suggest the accumulation of gradient values for each parameter across multiple batches or time steps.

In the computational graph shown, we see the backward propagation flow, which is part of the training phase where the model's parameters are adjusted. The backward nodes represent the derivatives of the loss function with respect to the model's parameters, and the arrows indicate the direction of the gradients' flow.

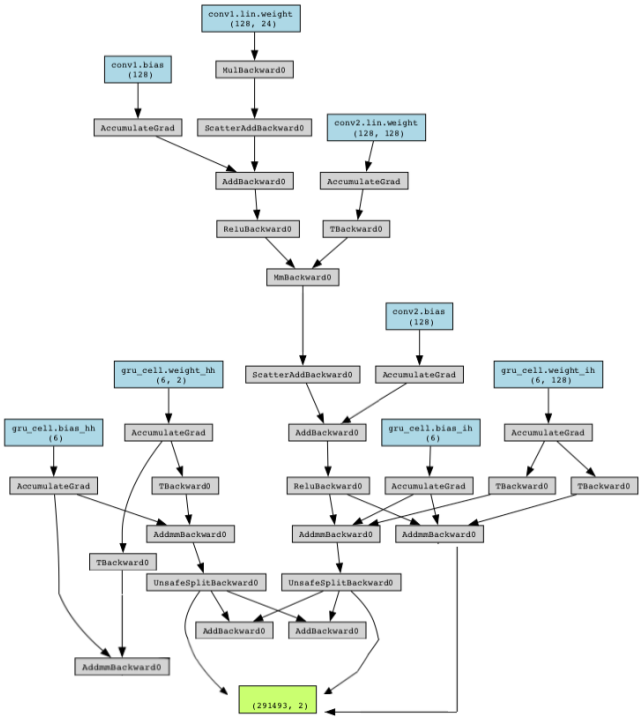


Figure 2: The Diagram for GNN+GRU Neural Network Model

5.2 Root Cause Analysis

In complex microservice applications, identifying the root causes of faults is critical for maintaining operational efficiency and reliability.

To systematically analyze and diagnose faults in our system, we employ a two-fold metric-based approach. This approach focuses on evaluating the effectiveness of InstantOps root cause localization processes at the node level. Specifically, we utilize Mean First Rank (MFR) and Mean Average Rank (MAR) as our primary metrics.

Mean First Rank (MFR): MFR focuses on the position where the first correct item (fault) appears in a ranked list. In the context of fault analysis, this means identifying the most likely root cause of a problem as quickly as possible. The quicker a primary fault is identified, the faster remedial actions can be taken. This is crucial in systems where prompt fault resolution is essential to minimize downtime or prevent cascading issues. In practice, MFR is calculated by averaging the ranks at which the first true fault appears across all instances in a dataset. A lower MFR value indicates higher efficiency in pinpointing the primary fault quickly.

Mean Average Rank (MAR): MAR extends the analysis to consider the average rank of all relevant items in the ranked list. This is particularly important in scenarios where multiple potential faults might contribute to a problem. MAR provides a broader view of the system's diagnostic accuracy. It is essential for comprehensive fault identification, especially in complex systems where multiple issues can coexist or be interrelated. MAR is calculated by averaging the ranks of all relevant faults across each instance in the dataset. It involves more intricate computations as it takes into account the position of each relevant item, not just the first one.

To implement these metrics, we utilized three different datasets, each comprising a ranked list of potential faults for each node generated upon the detection of a fault in the system. For instance, in the case of the QoTD datasets, we encountered three distinct faults: CPU, Memory, and DNS. The ranked lists from these datasets are then scrutinized using the MFR and MAR metrics, allowing us to quantify the accuracy and efficiency of our fault identification process. This methodology ensures a robust analysis of the diagnostic capabilities at the node level within our system.

6 EVALUATION

Here, we present the data for evaluation, evaluate our methods for failure prediction and root cause analysis, and present the performance of the models on various datasets and baselines.

6.1 Datasets

In this paper, we utilized two open-source microservice datasets: Train-Ticket and MicroSS, and Quote of the Day (QoTD) application that was deployed in-house on IBM OpenShift Clusters V4.12.36 with 16 CPU cores, and 32 GB Ram.

Train-Ticket [37] which comprises 41 microservices, is frequently used by researchers for root cause identification and localization. We accessed this application made available online by Li et al., [16] to enable comparative analysis with their findings. Three types of faults were introduced [16]: application bugs, CPU exhaustion, and network congestion. While these faults were introduced at various system levels, we specifically focused on those injected into the microservices, aligning them for comparison with the other two datasets.

MicroSS, also known as the Genetic AIOps Atlas (GAIA) dataset [1], contains 10 microservices, two databases (MYSQL and Redis), and is supported by five host machines. It is designed to cater to both mobile and PC users. The GAIA dataset encompasses five distinct faults: system hang-ups, process crashes, system failures like login issues, missing files, and access denials. A record detailing the injection of these failures is provided alongside the data. This dataset has been widely used for predicting system failure and root cause localization in [36] and [34].

The QoTD open-source application [3] consists of eight distinct microservices. We deployed QoTD on IBM OpenShift clusters and introduced a variety of faults using Chaos-Mesh [4]. These faults include disruptions in CPU, memory, and DNS.

For monitoring purposes, we employed Instana [5], which provided data points such as API response times, error codes, and various resource utilization metrics: CPU, memory, disk, and network. We utilized LogDNA [6] to extract application logs. Furthermore, we devised a custom script to capture events per minute at the node level. This script monitored events like "Scheduled", "Unscheduled", "Pulled", "Failed", "Started", etc. These events were retrieved by querying 'oc describe <resource> <resource-name>'.

6.2 Evaluation Metrics

6.2.1 Failure Prediction: The model is trained using CrossEntropy loss, formulated as:

$$L = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

where y_i and $\hat{y}_i$ are the true label and predicted probability for sample i, respectively.

To assess the model, accuracy and F1 Score are computed. Accuracy is given as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

F1 Score is calculated using precision and recall, which consider the model's performance regarding false positives and false negatives. F1 Score is expressed as:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

with

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

and

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

6.2.2 Root Cause Analysis: To validate and understand the impact of crucial nodes for the root cause, we employed a function for evaluating the accuracy without k nodes. This function deliberately nullifies the top k nodes (based on their importance) and evaluates how the model's accuracy is affected without their presence:

[1][2]

[3]https://gitlab.com/quote-of-the-day/quote-of-the-day/-/tree/master
[4]https://chaos-mesh.org/
[5]https://www.ibm.com/products/instana
[6]https://www.ibm.com/case-studies/logdna-cloud

- **Ranking Nodes:** Nodes are ranked in descending order based on their computed importance.
- **Nullification of Top Nodes:** Features of the top k nodes are set to zero, effectively removing their influence from the network.
- **Model Evaluation:** With these nodes nullified, the model's accuracy is gauged again, highlighting the impact of these crucial nodes on the network's overall performance. A pronounced drop in these accuracies compared to the baseline underscores the critical nature of these nodes within the network.

Top-k accuracy is a metric commonly used in retrieval and recommendation tasks. It measures how often the true item (or one of the true items) appears in the top k items of the ranked list and is defined as:

$$A@k = \frac{1}{|A|} \sum_{a \in A} \begin{cases} 1 & \text{if } RC_i^a \in RC_s^a[k] \\ 0 & \text{otherwise} \end{cases}$$

where,

- A is the set of test samples.
- RC_i^a is the true root cause instance for sample a.
- $RC_s^a[k]$ is the set of top-k predicted instances for sample a.

Mean First Rank (MFR) evaluates the average rank at which the first correct item is found in the ranked list and is defined as:

$$\text{MFR} = \frac{1}{N} \sum_{i=1}^{N} \text{rank}_i$$

where,

- N is the total number of ranked lists.
- rank_i is the rank of the first true item in the i^{th} ranked list.

Mean Average Rank (MAR) measures the average rank of all relevant items in the ranked list. It's a s metric when there are multiple relevant items per query, and you want to assess how well the system ranks all of them on average and is defined as:

$$\text{MAR} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{1}{|R_i|} \sum_{r \in R_i} \text{rank}_{i,r} \right)$$

where,

- N is the number of ranked lists.
- R_i is the set of relevant items for the i^{th} ranked list.
- $\text{rank}_{i,r}$ is the rank of relevant item r in the i^{th} ranked list.

6.3 Results and Discussions

In this paper, we focus on the resources' over-utilization (also known as overload) use case as the cause of anomalies. The assumption is that resource utilization happens due to causes external to the applications we monitor. Within this particular use case, our experiments are organized around several research questions, which are discussed below.

- **RQ1:** How efficient is `InstantOps` for system failure prediction compared to other neural network algorithms?
- **RQ2:** How does the efficiency of `InstantOps` for failure prediction compare to state-of-the-art methods?
- **RQ3:** How does `InstantOps` perform root cause analysis as compared to other neural network algorithms?
- **RQ4:** How does `InstantOps` perform in terms of root cause node localization compared to existing methods?

6.3.1 RQ1: How efficient is `InstantOps` *for system failure prediction compared to other neural network algorithms?* To assess the efficiency of `InstantOps`, three distinct datasets were utilized, as shown in Table 2. The first, MicroSS, is an open-source dataset comprising 419,959 data. This data was divided into training and testing sets with 84,026 and 335,933, respectively. The Train Ticket open-source dataset encompasses 24,492 data, which are further divided into 19,337 for testing and 5,085 for training. Lastly, our in-house dataset includes 450,000 data, with a distribution of 360,000 for testing and 90,000 for training.

Table 2: Data Overview for Applications

Application	Total Samples	Test_data	train_data
MicroSS	419959	335933	84026
Train Ticket	24492	19337	5085
QoTD	450000	360000	90000

For each dataset, several applications metrics were considered which we defined as features: application resource utilization, logs detailing errors for each node within specific timeframes (e.g., 1 minute), edge constructions that link nodes with traces within certain time windows, and the frequency of interactions between nodes. These metrics assist in identifying anomalies by observing deviations in standard node interactions. Specifically for the QoTD dataset, we also incorporated event metrics, capturing events such as out of memory, scheduled activities, pull events, failed creation events, and image pull back-offs on a per-node basis. Using this data, we aimed to predict systemic failures.

We adhered to a standard Service Level Agreement (SLA) that designates a system as "failed" if 99.9% of the total requests received on the server resulted in errors, such as a 503 error, within a specific timeframe (e.g., 1 minute). The labeled dataset was employed to validate the accuracy of our predictive efforts.

Further validation was sought by leveraging multiple neural network models, as delineated in Table 3. These models, previously utilized by other researchers [34, 36], were examined to compare the effectiveness of `InstantOps`. Notably, InstantOps employs a Graph Neural Network (GNN) to structure the relationships between features and nodes. Additionally, a Gated Recurrent Unit (GRU) captures temporal aspects of each node in the system, such as time lags of t-2 and t-5.

The performance metrics revealed that `InstantOps` achieved the highest F1 score for MicroSS at 0.98, with a recall of 0.98 and a precision of 0.97. Similar scores were observed for the Train Ticket dataset. For the QoTD dataset, the F1 score was 0.96, with a recall and precision both measuring 0.96. When comparing other neural network algorithms using the MicroSS dataset, the combination of GTN with GRU and LSTM yielded an F2 score of 0.94. The Train Ticket dataset registered a score of 0.92 for the integration of GTN and LSTM, while GNN and LSTM achieved 0.94.

Table 3: Comparison of different algorithms for MicroSS, Train Ticket, and QoTD datasets

	MicroSS				Train Ticket				QoTD			
Algorithm	Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1
InstantOps	**0.97**	**0.97**	**0.98**	**0.98**	**0.98**	**0.97**	**0.98**	**0.98**	**0.97**	**0.96**	**0.96**	**0.96**
GNN+LSTM	0.93	0.93	0.93	0.93	0.91	0.87	0.91	0.89	0.91	0.94	0.94	0.94
GNN	0.90	0.93	0.90	0.91	0.87	0.83	0.87	0.85	0.92	0.93	0.92	0.93
GTN	0.73	0.93	0.73	0.81	0.92	0.86	0.92	0.89	0.88	0.89	0.88	0.88
GTN+LSTM	0.95	0.93	0.96	0.94	0.93	0.92	0.96	0.92	0.91	0.90	0.94	0.93
GTN+GRU	0.96	0.92	0.96	0.94	0.93	0.86	0.93	0.90	0.92	0.94	0.89	0.92

RQ1: We showed that by fusing traces, logs, and metrics and capturing temporal aspects using GRUs to identify anomalies by observing deviations in the interactions between services, `InstantOps` obtained higher precision, recall and F1 scores in comparison to other neural network algorithms on three datasets: QoTD, MicroSS and Train Ticket.

6.3.2 RQ2: How does the efficiency of `InstantOps` *for failure prediction compare to state-of-the-art methods?* Table 4 illustrates a comparison study between `InstantOps`, JLT and Anafusion to establish a multimodal baseline.

The JLT method aggregates the results from JumpStarter [22], LogAnomaly [23], and Traceanomaly [20]. It employs majority voting, marking a failure if two or more modalities fail simultaneously. A notable observation about JLT is that it disregards the correlation among different modalities. For the MicroSS dataset, which integrates metrics, logs, and traces, JLT achieves an F1 score of 0.61, a recall of 0.94, and a precision of 0.46.

To further improve JLT's performance for fault prediction, Zhao et [36]. proposed Anafusion, an unsupervised failure detection technique that integrates multimodal data for microservices. It uses a graph neural network to learn the correlations in the heterogeneous multimodal dataset. For the same MicroSS dataset, Anafusion achieved an F1 score of 0.85, a recall of 0.94, and a precision of 0.79.

In `InstantOps`, we have further refined the Anafusion model by constructing a graph based on traces, which displays interactions among the nodes in real time. `InstantOps` employs both GNN and GRU to account for the system's temporal aspects. Our experimental results indicate that `InstantOps`, when applied to the MicroSS dataset, achieves an F1 score of 0.98, a precision of 0.97, and a recall of 0.98.

Table 4: The Average Percentage Among Precision, Recall, and F1-Score of Different Approaches on MicroSS Dataset

	Modality			MicroSS Dataset		
Approach	Metric	Log	Trace	Prec.	Rec.	F1
InstantOps	✓	✓	✓	**0.970**	**0.980**	**0.980**
AnoFusion	✓	✓	✓	0.795	0.945	0.857
JLT	✓	✓	✓	0.461	0.940	0.618

RQ2: We showed that by constructing a dependency graph based on traces to depict the service interactions of the faulty service using GNN and GRU to incorporate temporal aspects, `InstantOps` achieves higher precision, recall and F1 score than the state-of-the-art methods.

6.3.3 RQ3: How does `InstantOps` *perform root cause analysis as compared to other neural network algorithms?* To assess the efficiency of `InstantOps` in root cause localization at the node level, we used two metrics: MFR (Mean First Rank) and MAR (Mean Average Rank) described earlier on three datasets: MicroSS, Train Ticket, and QoTD as shown in Table 5.

As can be seen from the table, `InstantOps` achieves an MFR score of 1.49 and a MAR score of 1.51 on the QoTD dataset. On the MicroSS dataset, `InstantOps` achieves a score of 1.51 for both MFR and MAR. On the Train Ticket dataset, `InstantOps` achieves a score of 1.06 for both MAR and MFR. We also observed that the combination of GNN and LSTM exhibited similar strong performance. For the QoTD dataset, GNN+LSTM achieved a score of 1.51 for both MAR and MFR. On the MicroSS dataset, GNN+LSTM achieved a score of 1.06 for MAR and a score of 1.07 for MFR. On the Train Ticket dataset, GNN+LSTM achieved a score of 1.6 for both MAR and MFR.

Table 5: Effectiveness of failure type determination at the node level

Algorithm	QoTD		MicroSS		Train Ticket	
	MAR	MFR	MAR	MFR	MAR	MFR
InstantOps	**1.51**	**1.49**	**1.06**	**1.06**	**1.06**	**1.06**
GNN+LSTM	1.51	1.51	1.06	1.07	1.06	1.06
GNN	1.51	1.51	1.59	1.59	1.06	1.19
GTN	1.67	1.62	1.15	1.24	1.47	1.47
GTN+LSTM	1.89	1.89	1.18	1.11	1.37	1.47
GTN+GRU	1.89	1.56	1.06	1.14	1.90	1.95

RQ3: We showed by localizing the root causes at the node level, the `InstantOps` approach performs better in terms of MAR and MFR in comparison to other neural network algorithms in its effectiveness of determining failure type at the node level on three datasets: QoTD, MicroSS and Train Ticket.

6.3.4 RQ4: How does `InstantOps` *perform in terms of root cause node localization compared to existing methods?* In this experiment, we compare `InstantOps` with seven algorithms. Microscope and MEPFL are microservice anomaly detection approaches where Microscope collects network and SLO metrics to infer root causes during SLO violations while MEPFL predicts latent errors and faulty microservices by integrating trace logs and injecting faults. TraceAnomaly, an unsupervised approach, learns trace patterns to detect abnormal traces and localise root causes. MonitorRank

employs Random Walk, which combines historical and real-time metrics for root cause ranking in service-oriented web architectures. RCSF, designed for enterprise systems, analyses performance logs and dependency models to identify fault propagation sequences. `InstantOps` is designed to fuse features such as resource utilization, events and logs and construct graph neural network based on the interations among the nodes in microservice. and it localize the faulty node that corresponds to system failure.

Table 6 provides a quantitative comparison of `InstantOps` with seven algorithms such as TraceRCA [16], MicroScope [17], MEPFL (RF) [38], TraceAnomaly[20], Random-Walk [31], and RCSF [30] benchmarked against other methods sourced from [16]. The metrics $A@1$, $A@2$, and $A@3$ are utilized as evaluative standards.

As is evident from the table, `InstantOps` demonstrates a high degree of effectiveness. With an $A@1$ score of 0.92, it surpasses the majority of the algorithms in the list and maintains consistent performance across $A@2$ and $A@3$. This consistent high performance across metrics suggests the reliability and robustness of the `InstantOps` algorithm.

Algorithms such as `Random Walk` and `RCSF`, although displaying commendable values in $A@2$ and $A@3$, have relatively lower $A@1$ values. This difference could indicate potential variability in their performance across different stages or conditions.

Conversely, `TraceAnomaly` and `MicroScope` consistently perform worse, further delineating the performance gap between these methods and `InstantOps`.

Table 6: Comparison of root cause localization on faults of different levels on A

Algorithm	A@1	A@2	A@3
InstantOps	**0.92**	**0.95**	**0.98**
TraceRCA	0.83	0.93	0.97
MicroScope	0.56	0.62	0.7
MEPFL (RF)	0.94	0.97	0.97
Random Walk	0.51	0.86	0.94
RCSF	0.52	0.86	0.93
TraceAnomaly	0.49	0.59	0.63

RQ4: We showed by localizing the root causes at the node level, `InstantOps` outperforms in terms of the evaluative standards A@1, A@2 and A@3 in comparison to other existing methods in root cause localization on faults of different levels on A, demonstrating higher effectiveness than existing methods.

7 THREATS TO VALIDITY

In our research, we've identified several threats to validity that warrant careful consideration. The first threat pertains to the accuracy of failure labeling within our microservices study. Specifically, while examining the 'Quote of the Day' (QoTD) application, we established a performance baseline with JMeter and subsequently annotated failure events using Chaos Mesh for fault injection. This process was supplemented by observations of microservice crashes in OpenShift clusters. Nevertheless, any potential mislabeling or misinterpretation of these events could compromise the integrity of our findings.

Further complicating our validity is the diversity of our data sources. We've utilized three datasets, including two open-source ones, to support the generalizability of our results. However, the variance in size and scope between our experimental data and the real-world complexity of microservice operations could limit the applicability of our conclusions.

Another significant validity threat arises from the granularity of our data collection. By capturing a wide array of metrics—from resource usage to node interactions—on a one-minute interval, we assume that this level of granularity is sufficient for predicting imminent system failures. Yet, there is a risk that more nuanced or granular data could yield different insights, which means our current approach may overlook certain subtleties.

Lastly, the scope of our datasets, which are smaller in comparison to those used in extensive industrial microservice systems, could undermine the scalability of our algorithm. While we believe our algorithm should function effectively even with coarser-grained datasets, the true test of its applicability will come when it is applied to the larger and more complex datasets that we plan to obtain from our clients in future work. This step is crucial for us to validate the efficiency of our model and ensure that it can withstand the demands of a full-scale industrial environment.

8 CONCLUSION

In this work, we proposed `InstantOps`, an approach that takes in multi-modal data to construct a graph using traces with logs and metric data as node attributes. Through this, we use our multi-modal system failure prediction approach and capture temporal and spatial aspects to precisely and effectively predict failures and determine the root causes of failures at the node level. In addition to using our in-house data set: QoTD, we used two open source datasets: MicroSS and Train-Ticket. In our experimental studies, we have shown how `InstantOps` can identify and localize the faulty node in microservice which can facilitate the root cause analysis of the system failure. We believe that the use of a Graph Neural Network to construct topology and interaction among the nodes and incorporating temporal information using the GRU model improves the prediction of system failure. We aim to evolve `InstantOps` into an online learning system capable of updating its models in real time. This continuous learning approach would allow for immediate adjustments based on the latest system behavior, thereby enhancing predictive accuracy over time.

9 ACKNOWLEDGMENT

We would like to thank IBM Center for Advanced Studies and Ontario ORF-RE for their support.

REFERENCES

[1] Rafiul Ahad, Eric Chan, and Adriano Santos. 2015. Toward autonomic cloud: Automatic anomaly detection and resolution. In *2015 International Conference on Cloud and Autonomic Computing*. IEEE, 200–203.

[2] Vilde Christiansen, Moutaz Haddara, and Marius Langseth. 2022. Factors affecting cloud ERP adoption decisions in organizations. *Procedia Computer Science* 196 (2022), 255–262.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[4] Ying Fu, Meng Yan, Jian Xu, Jianguo Li, Zhongxin Liu, Xiaohong Zhang, and Dan Yang. 2022. Investigating and improving log parsing in practice. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1566–1577.

[5] Sahil Garg, Kuljeet Kaur, Neeraj Kumar, Shalini Batra, and Mohammad S Obaidat. 2018. HyClass: Hybrid classification model for anomaly detection in cloud environment. (2018).

[6] Sahil Garg, Kuljeet Kaur, Neeraj Kumar, Georges Kaddoum, Albert Y Zomaya, and Rajiv Ranjan. 2019. A hybrid deep learning-based model for anomaly detection in cloud datacenter networks. *IEEE Transactions on Network and Service Management* 16, 3 (2019), 924–935.

[7] L Girish and Sridhar KN Rao. 2021. Anomaly detection in cloud environment using artificial intelligence techniques. *Computing* (2021), 1–14.

[8] Pranjal Gupta, Harshit Kumar, Debanjana Kar, Karan Bhukar, Pooja Aggarwal, and Prateeti Mohapatra. 2023. Learning Representations on Logs for AIOps. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 155–166.

[9] Tanja Hagemann and Katerina Katsarou. 2020. A systematic review on anomaly detection for cloud computing environments. In *2020 3rd Artificial Intelligence and Cloud Computing Conference*. 83–96.

[10] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*. IEEE, 33–40.

[11] Jordan Hochenbaum, Owen S Vallis, and Arun Kejariwal. 2017. Automatic anomaly detection in the cloud via statistical learning. *arXiv preprint arXiv:1704.07706* (2017).

[12] Mohammad S Islam, William Pourmajidi, Lei Zhang, John Steinbacher, Tony Erwin, and Andriy Miranskyy. 2021. Anomaly detection in a large-scale cloud platform. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 150–159.

[13] Abul Khayer, Nusrat Jahan, Md Nahin Hossain, and Md Yahin Hossain. 2021. The adoption of cloud computing in small and medium enterprises: a developing country perspective. *VINE Journal of Information and Knowledge Management Systems* 51, 1 (2021), 64–91.

[14] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R Lyu. 2023. Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-Source Data. *arXiv preprint arXiv:2302.05092* (2023).

[15] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, Yongqiang Yang, and Michael R Lyu. 2023. Heterogeneous anomaly detection for software systems via semi-supervised cross-modal attention. *arXiv preprint arXiv:2302.06914* (2023).

[16] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. 2021. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 1–10.

[17] JinJin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, Vol. 16. Springer, 3–20.

[18] Marin Litoiu, Ian Watts, and Joe Wigglesworth. 2021. The 13th cascon workshop on cloud computing: engineering aiops. In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*. 280–281.

[19] Dapeng Liu, Youjian Zhao, Haowen Xu, Yongqian Sun, Dan Pei, Jiao Luo, Xiaowei Jing, and Mei Feng. 2015. Opprentice: Towards practical and automatic anomaly detection through machine learning. In *Proceedings of the 2015 internet measurement conference*. 211–224.

[20] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 48–58.

[21] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. 2020. Self-adaptive root cause diagnosis for large-scale microservice architecture. *IEEE Transactions on Services Computing* 15, 3 (2020), 1399–1410.

[22] Minghua Ma, Shenglin Zhang, Junjie Chen, Jim Xu, Haozhe Li, Yongliang Lin, Xiaohui Nie, Bo Zhou, Yong Wang, Dan Pei, et al. 2021. Jump-Starting multivariate time series anomaly detection for online service systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 413–426.

[23] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI*, Vol. 19. 4739–4745.

[24] Joydeep Mukherjee, Alexandru Baluta, Marin Litoiu, and Diwakar Krishnamurthy. 2020. RAD: Detecting performance anomalies in cloud-based web services. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 493–501.

[25] N Pandeeswari and Ganesh Kumar. 2016. Anomaly detection system in cloud environment using fuzzy clustering based ANN. *Mobile Networks and Applications* 21 (2016), 494–505.

[26] Daniel Pop. 2016. Machine learning and cloud computing: Survey of distributed and saas solutions. *arXiv preprint arXiv:1603.08767* (2016).

[27] Y Rouf, J Mukherjee, and M Litoiu. 2023. Towards a Robust On-line Performance Model Identification for Change Impact Prediction. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Melbourne, Australia, 68–78. https://doi.org/10.1109/SEAMS59076.2023.00018

[28] Jacopo Soldani and Antonio Brogi. 2022. Anomaly detection and failure root cause analysis in (micro)service-based cloud applications: A survey. *ACM Computing Surveys (CSUR)* 55, 3 (2022), 1–39.

[29] Arthur Vervaet. 2021. MoniLog: An Automated Log-Based Anomaly Detection System for Cloud Computing Infrastructures. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2739–2743.

[30] Kui Wang, Carol Fung, Chao Ding, Polo Pei, Shaohan Huang, Zhongzhi Luan, and Depei Qian. 2015. A methodology for root-cause analysis in component based systems. In *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*. IEEE, 243–248.

[31] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. 2018. Cloudranger: Root cause identification for cloud native systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 492–502.

[32] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*. 3087–3098.

[33] Hongwei Zhang, Yuanqing Xia, Tijin Yan, and Guiyang Liu. 2021. Unsupervised anomaly detection in multivariate time series through transformer-based variational autoencoder. In *2021 33rd Chinese Control and Decision Conference (CCDC)*. IEEE, 281–286.

[34] Shenglin Zhang, Pengxiang Jin, Zihan Lin, Yongqian Sun, Bicheng Zhang, Sibo Xia, Zhengdan Li, Zhenyu Zhong, Minghua Ma, Wa Jin, et al. 2023. Robust Failure Diagnosis of Microservice System through Multimodal Data. *arXiv preprint arXiv:2302.10512* (2023).

[35] Xu Zhang, Qingwei Lin, Yong Xu, Si Qin, Hongyu Zhang, Bo Qiao, Yingnong Dang, Xinsheng Yang, Qian Cheng, Murali Chintalapati, et al. 2019. Cross-dataset Time Series Anomaly Detection for Cloud Systems. In *USENIX Annual Technical Conference*. 1063–1076.

[36] Chenyu Zhao, Minghua Ma, Zhenyu Zhong, Shenglin Zhang, Zhiyuan Tan, Xiao Xiong, LuLu Yu, Jiayi Feng, Yongqian Sun, Yuzhi Zhang, et al. 2023. Robust Multi-Modal Failure Detection for Microservice Systems. In *arXiv preprint arXiv:2305.18985*.

[37] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.

[38] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 683–694.

Daedalus: Self-Adaptive Horizontal Autoscaling for Resource Efficiency of Distributed Stream Processing Systems

Benjamin J. J. Pfister
Technische Universität Berlin
Berlin, Germany
benjamin.pfister@campus.tu-berlin.de

Dominik Scheinert
Technische Universität Berlin
Berlin, Germany
dominik.scheinert@tu-berlin.de

Morgan K. Geldenhuys
Technische Universität Berlin
Berlin, Germany
morgan.geldenhuys@tu-berlin.de

Odej Kao
Technische Universität Berlin
Berlin, Germany
odej.kao@tu-berlin.de

ABSTRACT

Distributed Stream Processing (DSP) systems are capable of processing large streams of unbounded data, offering high throughput and low latencies. To maintain a stable Quality of Service (QoS), these systems require a sufficient allocation of resources. At the same time, over-provisioning can result in wasted energy and high operating costs. Therefore, to maximize resource utilization, autoscaling methods have been proposed that aim to efficiently match the resource allocation with the incoming workload. However, determining when and by how much to scale remains a significant challenge. Given the long-running nature of DSP jobs, scaling actions need to be executed at runtime, and to maintain a good QoS, they should be both accurate and infrequent. To address the challenges of autoscaling, the concept of self-adaptive systems is particularly fitting. These systems monitor themselves and their environment, adapting to changes with minimal need for expert involvement.

This paper introduces *Daedalus*, a self-adaptive manager for autoscaling in DSP systems, which draws on the principles of self-adaption to address the challenge of efficient autoscaling. Daedalus monitors a running DSP job and builds performance models, aiming to predict the maximum processing capacity at different scaleouts. When combined with time series forecasting to predict future workloads, Daedalus proactively scales DSP jobs, optimizing for maximum throughput and minimizing both latencies and resource usage. We conducted experiments using Apache Flink and Kafka Streams to evaluate the performance of Daedalus against two state-of-the-art approaches. Daedalus was able to achieve comparable latencies while reducing resource usage by up to 71%.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Availability**; • **Computing methodologies → Massively parallel algorithms**.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645042

KEYWORDS

Distributed Stream Processing, Autoscaling, System Tuning, Performance Modeling, Resource Management, Cloud Computing

ACM Reference Format:
Benjamin J. J. Pfister, Dominik Scheinert, Morgan K. Geldenhuys, and Odej Kao. 2024. Daedalus: Self-Adaptive Horizontal Autoscaling for Resource Efficiency of Distributed Stream Processing Systems. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645042

1 INTRODUCTION

Distributed Stream Processing (DSP) is an important paradigm that enables quick extraction of insights from unbounded data streams with high throughput and low latencies. The generation of streaming data is continually increasing and this trend is evident across a range of contexts, including online advertising, financial transactions, and IoT sensor networks [4, 20, 21]. In order to provide a good Quality of Service (QoS), DSP systems need to be properly configured. Insufficient allocation of resources can lead to unstable service delivery, while over-provisioning results in wasted energy and higher operational costs. Achieving the right balance in resource allocation is therefore crucial to optimize both performance and cost-effectiveness. However, due to the dynamic nature of streaming workloads, configurations can quickly become obsolete. Likewise, because the manual tuning of configurations is infeasible over the course of a long-running job, resources can only be adjusted through an automated approach. Therefore, it is prudent to provide DSP systems with self-adaptive capabilities, enabling them to monitor themselves and respond to environmental changes by autonomously tuning their configurations during runtime.

Automated scaling of computational resources, commonly known as autoscaling, serves as the key method for aligning resources with dynamic workloads in DSP systems. The scaleout, defined by the number of worker nodes and processing slots per node, directly determines the system's level of parallelism and influences the overall resource allocation. This, in turn, affects each worker node's processing potential. Popular DSP frameworks like Apache Flink [3], Kafka Streams [19], and Apache Spark [23], support dynamic adjustments in scaleout during runtime, which impacts the job's parallelism and subsequently its resource utilization. Typically, these systems are deployed in cloud environments, where resources

are provisioned elastically and can be scaled in or out as required, directly affecting the processing capabilities of the DSP job.

Autoscaling approaches are optimized towards various adaptation goals, with most research primarily focusing on enhancing DSP system performance by maximizing throughput or minimizing latency [4, 18]. Additionally, secondary goals like minimizing resource usage, reducing monetary costs, lowering energy consumption, or shortening recovery times are also considered [4, 21]. Despite active research in this area, there is still potential for further advancements. Many existing methods demand in-depth system or job knowledge, such as setting scaling thresholds or altering DSP source code [13, 14, 24]. Often, scaling decisions lead to downtime for initializing new workers or recalculating data distribution among operators, yet few strategies account for this overhead. Additionally, many presume an even distribution of data across parallel operators, which is not always the case in practical scenarios.

This paper approaches the challenges of autoscaling DSP systems from the perspective of self-adaptation. It proposes a self-adaptive manager called Daedalus that targets a running DSP job and horizontally scales its parallelism to adapt to the incoming workload while minimizing resource usage. In order to meet QoS requirements, Daedalus can optimize towards a target recovery time and ensure that a job will recover between scaling actions. Because scaling decisions incur an overhead cost, Daedalus employs Time Series Forecasting (TSF) to predict the future workload in order to reduce the frequency of scaling actions. By ensuring that the incoming workload can be processed and minimizing system downtime, Daedalus can achieve reasonable latencies. Scaling decisions are realized through a combination of monitoring, performance modeling, and TSF. Unlike most existing approaches that ignore how data is split among parallel workers, Daedalus explicitly incorporates data skew in its capacity models. It is a general approach, applicable to containerized DSP systems running in cloud environments. Daedalus has been evaluated using three benchmark DSP jobs, employing two DSP frameworks, Apache Flink and Kafka Streams, and is compared against two state-of-the-art approaches.

This paper is structured as follows: Section 2 reviews related work on autoscaling, recovery time, and time series forecasting. Section 3 describes the approach taken to realize self-adaptive autoscaling. In Section 4, the approach is evaluated, and the results are discussed. Lastly, the paper is concluded in Section 5.

2 RELATED WORK

Adaptive autoscaling for DSP systems is an active field of research. This section describes the most relevant state-of-the-art solutions that Daedalus builds upon and work relevant to recovery time and TSF.

Heinze et al. [12] assess the viability of both threshold-based and reinforcement learning autoscaling techniques using their approach called FUGU. The authors find that reinforcement learning produces scaling decisions that best maximize system utilization, and that global thresholds are not well-suited for autoscaling. However, it should also be noted that reinforcement learning techniques can take a long time to adequately train and need to make ill-suited scaling decisions to learn.

Dhalion [8] is a self-adaptive system developed for Heron that employs user-defined policies to enable autoscaling. By monitoring metrics such as backpressure, tuples waiting in buffers, and data skew across operators, Dhalion determines if resources are over-provisioned or under-provisioned and scales in or out accordingly. Although Dhalion aims to be a comprehensive self-adaptive DSP solution, defining suitable policies requires expert knowledge of the job and system. Additionally, its autoscaling capabilities have been found to over-provision resources. Because individual operators are scaled one at a time after detecting backpressure, Dhalion needs a long time to converge to an optimal configuration [13].

DS2 is a reactive autoscaling approach that monitors a running DSP job to calculate true processing rates in order to determine the proportional processing relationships among operators [13]. By scaling in response to the workload, DS2 can accurately adjust the parallelism of operators. However, DS2 assumes data skew is not present and that workloads remain stable during scaling operations. It also requires manual implementation of the true processing rate metric, as this metric is not readily available in all DSP systems.

AuTraScale [24] utilizes Bayesian Optimization for operator autoscaling to minimize latency while maximizing total throughput. Like DS2, it uses the true processing rates to find the minimum parallelism needed to process a static workload. When the system is over-provisioned or latency exceeds a target threshold, AuTraScale reactively rescales. Although designed to find the optimal configuration for a static workload, AuTraScale includes a transfer learning algorithm to more quickly find optimal configurations if the input rate would change. However, the authors do not evaluate their approach on a dynamic workload and assume no data skew.

Caladrius [14] is a performance modeling tool that uses TSF to anticipate future workloads and predict the throughput and CPU usage for both the current and other scale-outs to enable proactive scaling decisions. Like DS2, Caladrius models the operator topology and relationship between operator input and output rates. The authors make the important observation that throughput rates and CPU utilization are linearly related. Therefore, by predicting the throughput, the authors can predict CPU utilization. Daedaulus is heavily influenced by this and uses the linear relationship between throughput and CPU to estimate capacity. However, whereas Daedalus uses a CPU-throughput regression model to estimate capacity, Caladrius relies heavily on backpressure to observe maximum throughput rates, which is an unreliable metric in presence of data skew or slow nodes.

Relatively few DSP autoscaling approaches incorporate the overhead cost of scaling decisions [21]. Phoebe chooses scale-outs that can guarantee a target recovery time [10]. Martin et al. [15] provide a self-adaptive approach for a DSP system to adjust its fault tolerance mechanism during runtime. While not an autoscaling approach, their approach allows users to provide high level constraints such as a target recovery time. Lastly, Borkowski, Hochreiner, and Schulte [2] note the downtime caused by autoscaling and aim to reduce the number of rescaling actions for a threshold-based autoscaler. Their approach uses an extended Kalman filter to estimate trends in the workload, similar to ARIMA. By ignoring short-term variations in the workload, they are able to reduce the number of scaling decisions, thereby reducing overall recovery time over the course of the DSP job.

Gontarska et al. [11] compare commonly used TSF methods to assess their use in predicting DSP workloads. They compare seven methods, including ARIMA and two deep learning methods. Although deep learning methods produced the best predictions overall, they also required much longer training times. The ARIMA model, on the other hand, was trained faster and yielded good results for making short-term predictions in the tested 5 and 15 minute forecasts. Caladrius and Phoebe both use TSF to predict future workloads [10, 14]. Caladrius uses Facebook's Prophet while Phoebe also uses ARIMA.

3 APPROACH

In this paper, the challenges of autoscaling a DSP system for performance optimization are approached from the perspective of self-adaptation [5, 7, 16, 17]. Using self-adaptive autoscaling enables a DSP system to monitor itself and make scaling decisions to meet DSP requirements and adaptation goals while processing dynamic workloads. This paper proposes a self-adaptive DSP autoscaler called Daedalus. Its main objectives are to ensure enough resources are allocated to process the incoming workload while minimizing resource usage, meet QoS requirements by targeting a recovery time, and providing a stable level of service by enacting long-lived scaling decisions over the course of a long-running DSP job. On a high level, Daedalus uses the self-adaptive MAPE-K control loop to continuously monitor a running DSP job by collecting metrics stored in a time series database. It analyzes the data, builds models to estimate the capacity across all potential scale-outs, and predicts the future workload using TSF. Using both the historical and predicted workload, it decides if a scaling action is necessary and determines how to scale based on the described adaptation goals and estimated recovery time. Lastly, it executes a scaling action if necessary. A high-level overview of the approach, its architecture, and components can be seen in Figure 1.

In order to determine when and how to scale, Daedalus builds capacity models at the worker level using throughput and CPU utilization metrics. These metrics are typically already exposed by DSP systems so that their performance can be monitored. Monitoring provides accurate, up-to-date insights for a job running in a cloud environment. Previous observations, including a series of profiling runs conducted at the beginning of a deployment as in [10], have the potential to become less reliable over time. Since the underlying resources or placement of operators can change over the course of a long-running job, continually collecting metrics and monitoring the system performance is prudent.

Developing separate self-adaptive autoscaling solutions for each individual DSP system and job is complex and time-intensive. Therefore, Daedalus aims to provide a general solution for containerized DSP systems by requiring only a few commonly exposed metrics. In this paper, Daedalus is tested with Flink and Kafka Streams, though it is applicable to other systems as well.

Our approach makes the assumption that the DSP system is running in a cloud environment where homogeneous resources can be elastically scaled. However, it is also taken into account that these homogeneous resources may not provide identical performance by monitoring each worker individually. This approach also relies on a few metrics being available. These include the throughput, CPU utilization, and consumer lag for each worker, collected from the DSP system, and the workload, collected from the data source.

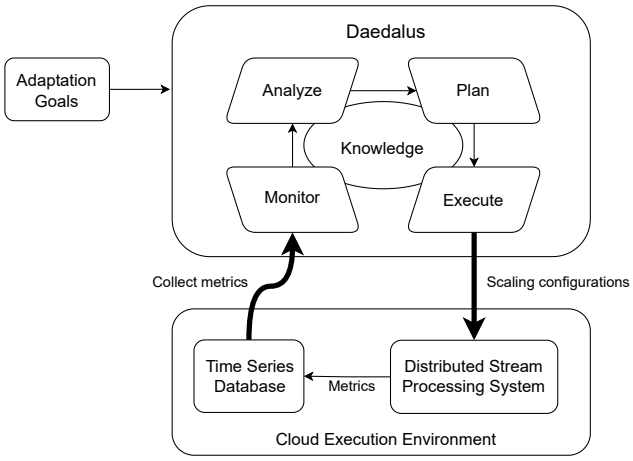


Figure 1: Daedalus architecture overview

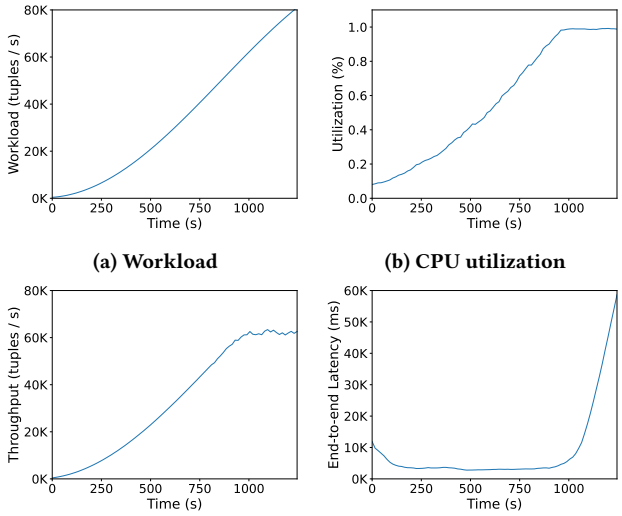
3.1 Performance Modeling

At the core of our approach is a capacity model, which provides a basis for all scaling decisions. It estimates the maximum number of tuples that can be processed per second at a given scale-out. Knowing this information along with the current and future workload informs if scaling is necessary and if so, how to scale. The capacity model is based on the observable relationship between metrics, especially, throughput and CPU utilization.

Figure 2 shows the relationship between the incoming workload, CPU utilization, throughput, and end-to-end latency at a fixed parallelism. The metrics were taken from a running job and are therefore not influenced by a warm-up period. As long as sufficient resources are available, the throughput of the DSP system will match the workload. When there are no longer resources to keep up with the rate of incoming tuples, the DSP system reaches its maximum capacity, and CPU utilization is 100%. In this example, throughput is capped at 60,000 tuples per second. As can be seen, the relationship between throughput and CPU utilization is linear.

End-to-end latency is impacted by several factors and is often job dependent. In jobs that use windowing, for example, end-to-end latency can increase when not enough tuples exist to trigger the end of the window. In Figure 2d, the latency is slightly influenced by the workload while processing capacity exists. However, this effect is minor when compared to the sharp increase when the workload exceeds maximum processing capacity. When the system cannot keep up with the incoming workload, either due to insufficient resources or downtime, tuples accumulate and end-to-end latency increases. It is therefore a main objective of Daedalus to ensure that sufficient capacity exists to process the incoming workload and to minimize downtime in order to limit spikes in end-to-end latency.

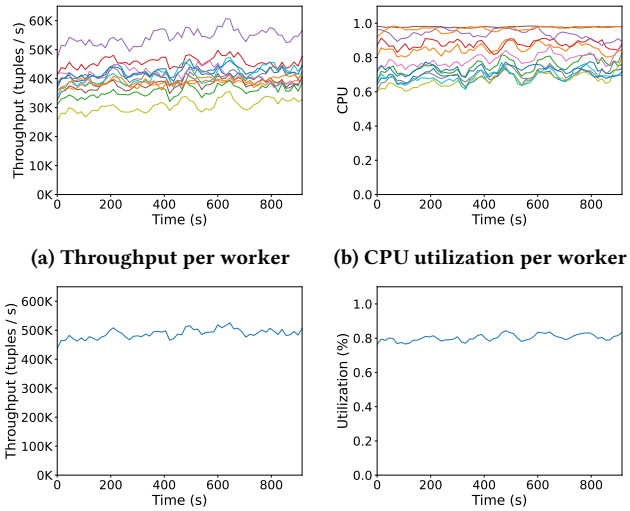
Much existing research on DSP autoscaling assumes that data is split equally among parallel operators, i.e. that data skew is not present. However, ignoring data skew can be a major weakness in these approaches as it is often present in reality.



(a) Workload (b) CPU utilization

(c) Throughput (d) End-to-end latency

Figure 2: Relationships between metrics

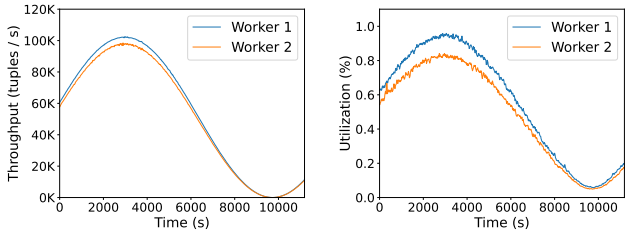


(a) Throughput per worker (b) CPU utilization per worker

(c) Total throughput (d) Average CPU utilization

Figure 3: Maximum throughput at a parallelism of 12 showing data skew and an average CPU utilization of 0.8.

Figure 3 shows metrics from a DSP job with stateful operators processing tuples at maximum capacity with a parallelism of 12. In this example, the data is generated randomly across 100 keys, and each worker reads from its own Kafka partition. In theory, the keys could be almost evenly distributed among parallel operators, with each worker handling eight or nine keys. However, as can be seen in Figure 3a and Figure 3b, data skew is apparent and the workers display a spectrum of throughput and CPU utilization. Although a worker using only 75% CPU is theoretically capable of processing more tuples, it cannot receive more tuples due to how the keys are distributed. Its maximum capacity is thus capped at its throughput at 75% CPU utilization. As seen in Figure 4, data skew across workers remains proportional at different levels of throughput and is most prominent at high CPU utilization. Following these observations, in general, the maximum capacity of a worker is limited by its proportion to the worker with the highest CPU utilization.



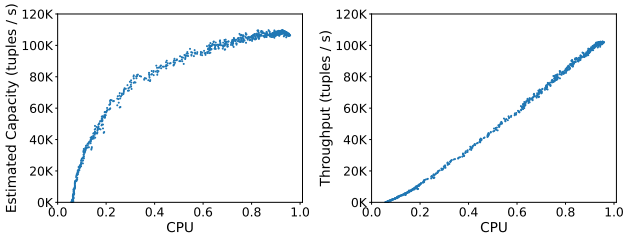
(a) Throughput per worker (b) CPU utilization per worker

Figure 4: Proportional data skew over CPU utilization

A worker's theoretical maximum capacity corresponds to how many tuples it can process at 100% CPU utilization. The calculation

$$\text{Capacity} = \frac{\text{Throughput}}{\text{CPU Utilization}}$$

yields a quick estimation of a worker's maximum capacity. However, as can be seen in Figure 5a, the accuracy of this estimation is highly dependent on the level of CPU utilization. Although this simple capacity calculation provides a reasonable estimation when CPU utilization is greater than 70%, and the maximum capacity estimation at a given CPU utilization is higher than the observed throughput, a more accurate capacity estimation is needed to inform stable scaling decisions. Given the linear relationship between throughput and CPU utilization, seen in Figure 5b, linear regression lends itself well to this task. Because only one explanatory variable is required to predict a worker's maximum throughput, it is possible to build a simple regression model using an efficient online analytical calculation. Using throughput as the dependent variable y and CPU utilization as the independent variable x, the throughput at a given CPU utilization can be predicted using the simple linear regression formula $y = \alpha + \beta x$, where α is the y-intercept and β is the slope. The slope can be calculated by dividing the covariance of CPU (X) and throughput (Y) observations by the variance of CPU observations. The y-intercept is given by subtracting the slope multiplied by the mean of all CPU observations ($\overline{X}$) from the mean



(a) Estimated capacity (b) Observed throughput

Figure 5: Capacity over CPU utilization

of all throughput observations ($\overline{Y}$). Put together, this yields the following equation to predict the capacity at a desired CPU utilization $\text{CPU}_{\text{desired}}$:

$$\text{Capacity} = \overline{Y} - \frac{\text{cov}(X,Y)}{\text{var}(X)} \cdot \overline{X} + \frac{\text{cov}(X,Y)}{\text{var}(X)} \cdot \text{CPU}_{\text{desired}}$$

To update such a model with new observations, the running covariance, variance, and means can be computed using an adaptation of Welford's online algorithm for calculating variance [22]. The algorithm passes over new observations once, updating the count of observations and the delta for the new mean CPU and throughput observations. These values are then used to update the variance, covariance, CPU, and throughput means. Welford's algorithm is numerically stable and all required values can be computed on one pass of the data, meaning that there is no need to save observations, which could require much storage space over the course of time.

A linear regression model is computed for each worker individually to increase the accuracy of the overall capacity estimation at a given scale-out. To find the capacity for each worker while accounting for data skew, the linear regression model can predict the value at the expected maximum CPU utilization. As described previously, the expected maximum CPU utilization of a worker is proportional to the worker with the maximum CPU.

Daedalus differentiates capacity estimations at seen and unseen scale-outs. The capacity at the current scale-out is calculated by summing the estimated capacity across all workers. The estimation for a current scale-out can accurately assess how data is distributed among workers. The capacity at other scale-outs is estimated using the average capacity multiplied by the scale-out. While this can not guarantee how data will be distributed at that scale-out, it provides an adequate heuristic. When possible, Daedalus uses previously observed capacity estimations over purely predicted estimates for seen scale-outs. Ideally, the regression models would have a range of CPU observations in order to be more robust and accurate. However, due to the low variance present in the CPU-throughput regression (as seen in Figure 5b), relatively few data points are needed to accurately estimate capacity. From experimentation, the regression model is able to accurately estimate capacity in as little as 60 seconds, the time of a single loop. This observation holds across different jobs and various scale-outs.

3.2 Scaling Decisions

Central to any autoscaling approach is how it makes scaling decisions. While reactive approaches benefit from being able to use real observations to inform scaling decisions, the point at which this data is available can lead to QoS violations until an acceptable scale-out is deployed. On the other hand, proactive approaches need to deal with the uncertainty of predicting the future. Daedalus uses a hybrid of reactive and proactive approaches, using both observed data and future forecasts in order to reactively scale in and proactively scale out.

In a reactive manner, Daedalus uses historical workload data since the last iteration of the MAPE-K loop to find the minimum scale-out needed to process the workload. This helps to offset uncertainty from inaccurate future predictions, such as when future forecasts are lower than the actual workload. Using TSF the workload can be anticipated to enable proactive scaling decisions. TSF grants the ability to scale out before capacity is exceeded, helping to reduce QoS violations and minimize end-to-end latency. It also enables making long-lived scaling decisions, as a scale-out can be chosen that can handle the current and future workload. It also allows to more accurately calculate recovery time, instead of needing to assume that the workload will remain constant.

In each iteration of the MAPE-K loop, the workload since the last iteration is collected, capacities are estimated for all scale-outs, and the future workload is predicted. With this information, it can be determined if rescaling is necessary and if so, how to rescale.

The pseudocode for determining the appropriate scale-out is shown in Algorithm 1. Because Daedalus aims to make long-lived scaling decisions, the algorithm first checks if rescaling is absolutely necessary in case a rescale recently occurred. If a rescale was done in the last ten minutes, it checks that the current capacity can handle the average observed workload and maximum future workload until the next loop iteration. The average workload is used instead of the maximum in order to remove noise from the actual workload, such as any spikes that may have occurred. In case both of these conditions are true, the algorithm returns and no rescaling is necessary.

Algorithm 1: An algorithm to determine the scale-out

```
Data: C := Capacities, W := Workload, TSF
if time since last rescale < 600s then
    if C_current > W_avg and TSF_max until next loop then
        return current parallelism
for i = 1 to MaxScaleout do
    if C_i > W_avg then
        RT_i ← predict_recovery_time(i)
        if RT_i > RT_target then
            continue
        if C_i < TSF_max until RT_i then
            continue
        if i = current parallelism then
            return i
        if i < current parallelism and C_i < consumer lag then
            continue
        if C_i > TSF_max then
            return i
return MaxScaleout
```

The algorithm then iterates over all possible scale-outs to find the lowest number of workers that can process the incoming workload for the next 15 minutes and ensures that recovery is possible within a target time. In the first step, it is checked that the scale-out is capable of processing the average observed workload, similar to reactive autoscaling approaches. This prevents assessing the validity of smaller scale-outs that cannot handle the incoming workload, which would produce an inaccurate scaling decision.

Next, the recovery time is predicted for the scale-out. The recovery time estimates the time needed to process accumulated tuples

while the system is down until it can catch up to a normal state. This handles cases for scaling in and out (when the examined scale-out is different from the current scale-out) as well as failure (when the scale-out is the same as the current scale-out). If the estimated recovery time is greater than the specified target recovery time, the currently investigated scale-out is invalid and the next scale-out is examined. It is also checked that the scale-out can handle the future workload while recovering. Otherwise, a rescale would be necessary while the system is recovering. If this is the case, the next scale-out is examined.

At this point in the algorithm, the examined scale-out is valid. In case the current parallelism is the same as the examined scale-out, the algorithm returns and no rescaling is necessary. Otherwise, the examined scale-out requires scaling in or out. To prevent scaling in too early, the consumer lag is investigated. In case the consumer lag is larger than the examined capacity, it is a good indication that the system is recovering or potentially overloaded. Even though the target recovery time would be met when scaling in, it is beneficial to wait until the system has caught up to provide better end-to-end latencies. Therefore, the next scale-out is examined, potentially pushing the decision to scale in until the next iteration.

Lastly, to ensure that the scale-out will be long-lived, it is checked that the capacity is greater than the maximum of the full TSF prediction of 15 minutes. Doing so also helps to minimize the need to scale again after 10 minutes, making the first check of the algorithm a precautionary measure. After a scaling action has been initiated, the system is given three minutes to adjust to the new scale-out before another scaling action can occur. This allows the system to stabilize and helps prevent flapping, where autoscaling alternates between scaling in and out. Including a grace period is a common practice. For example, Kubernetes Horizontal Pod Autoscaler (HPA) uses a default stabilization time of five minutes. Since Daedalus anticipates the future workload and scaling decisions are designed to be long-lived, this arbitrary threshold should have little impact. However, because no scaling actions can be taken in this period, it increases the need to generate accurate scaling decisions.

3.3 Time Series Forecasting

TSF is used to anticipate future workloads and enables proactive scaling decisions that help to make long-lived scaling decisions, reduce the overhead cost of rescaling, and minimize latencies. Despite DSP workloads being dynamic and therefore difficult to predict, TSF has been shown to improve autoscaling decisions to reduce resource consumption and better meet QoS requirements [10, 11]. Multiple TSF methods exist including ARIMA, Holt-Winters, FB Prophet, and deep learning methods. TSF models often need to be configured themselves, which can be challenging and require expert knowledge. Fortunately, models such as auto-ARIMA exist that automatically find optimal parameters based on provided workload data. Using the pmdarima[1] library, an initial model is trained with the available workload, and the model is updated with the latest workload observations in every iteration of the MAPE-K loop. A new forecast is generated in each loop to predict the next 15 minutes of the workload at second-level granularity.

[1] https://alkaline-ml.com/pmdarima/index.html, Accessed: March 2024

Though ARIMA has been extensively evaluated and has been found to produce good forecasts for up to 15 minutes [11], it is essential to evaluate the quality of the forecasts and include a mechanism to retrain models that consistently deliver poor predictions. In each iteration of the MAPE-K loop, the latest workload metrics are collected and a new forecast is generated. At this point, it is possible to compare the previous forecast with the latest workload observations. To gain an insight into the overall accuracy of the forecast, the weighted absolute percentage error (WAPE) is used, which is calculated by weighting the error between the actual values and forecasts with the sum of the actual values over all units of time. A lower WAPE indicates a better forecast. In case the previous forecast was inaccurate with respect to the workload, it is possible that the next forecast will be similarly inaccurate. If this happens, the ARIMA forecast is replaced with a forecast that applies a simple regression on the workload. This heuristic uses the slope from the latest workload observations and projects the workload 15 minutes into the future. This linear approximation only provides a fallback forecast when the previous TSF prediction was poor. If the TSF predictions are consistently poor for 15 consecutive iterations, the TSF model is retrained in a background thread to not interrupt the execution of the MAPE-K loop. Once training is completed, the newly trained model replaces the existing one.

3.4 Recovery Time

Recovery time is a vital, yet often overlooked, aspect of autoscaling DSP systems. DSP systems must be fault tolerant and continue to operate despite failures that are likely to occur over the course of a long-running job. In addition to failures, when rescaling, processing must often be stopped temporarily in order to start new workers and recompute data parallelism among stateful operators. To ensure exactly-once processing, the DSP system must reprocess tuples that occurred after the last completed checkpoint as well as process tuples that arrived while the system was down. In addition to this accumulated backlog, tuples will continue to arrive while the system is catching up. As depicted in Figure 6, recovery time describes the time from when processing stops, due to rescaling or failure, until the system has caught up and processed the accumulated backlog. Only then can normal processing resume.

Recovery time has a direct impact on end-to-end latency. While the DSP system is down tuples cannot be processed, increasing their end-to-end latency. While the system is recovering, the accumulated backlog must be processed before the tuples that continue to arrive can be processed, creating a cascading effect. Therefore, by incorporating a target recovery time into autoscaling decisions, latency can be better minimized by avoiding long recovery times.

Recovery time can be predicted by calculating the accumulated backlog and estimating how long it will take to process the backlog in addition to the incoming workload using the extra processing capacity of the targeted scale-out.

The accumulated backlog of tuples is given by the tuples that need to be reprocessed since the last completed checkpoint and tuples that arrive while the system is down. For calculating the number of tuples since the last checkpoint, the worst case is assumed, for example, 10 seconds for a 10 second checkpoint interval. The number of tuples to be reprocessed can therefore be calculated

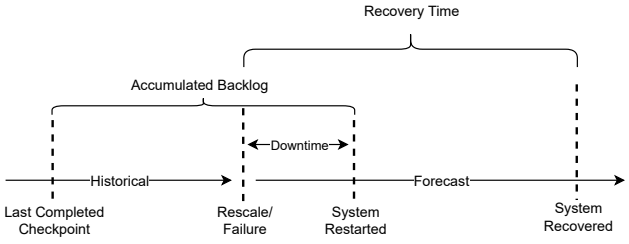


Figure 6: Predicting recovery time

by taking the number of tuples that occurred in the last checkpoint interval seconds using the historical workload. The worst case is assumed in order to provide a comparative baseline regardless of when the last checkpoint actually occurred with respect to the prediction. Failures and rescaling can occur at any time. Assuming the worst case also results in a larger recovery time prediction than would be expected on average, which provides a buffer to better achieve the target recovery time.

To estimate the number of tuples that arrive while the system is down, it is necessary to anticipate the time the system is down and use the workload forecast. The anticipated downtime is initially set to 30 seconds for scaling out and 15 seconds for scaling in. However, this value can be adaptively updated by monitoring the actual recovery time, discussed in the next section. This generally yields more accurate recovery time predictions over time.

After the DSP system restarts, it can begin processing the accumulated backlog using the maximum capacity of the target scale-out. While doing so, tuples will continue to arrive. Though the system will process the backlog first, when determining the point at which the system is caught up, the order tuples are processed is irrelevant. The end of the recovery time can thus be determined using the extra capacity, forecast, and accumulated backlog. The extra capacity available at the target scale-out can be obtained by subtracting the forecast from the capacity. For each future step, it is then checked if the cumulative extra capacity exceeds the accumulated backlog. When this is the case, the system has recovered, and the predicted recovery time can be returned.

3.5 Monitoring with Anomaly Detection

To improve the accuracy of recovery time predictions, the actual recovery time after a scaling action is observed using statistical anomaly detection. When monitoring recovery time, the goal is to identify when processing returns back to normal after rescaling. Anomaly detection is therefore a fitting paradigm, as it provides a means for classifying normal and abnormal behavior. Because the objective is to find points when the system's throughput deviates from the incoming workload, it is sufficient to use statistical anomaly detection on the difference between the workload and throughput. Statistical anomaly detection classifies observations as anomalous if the distance of an observation to the mean is above a certain threshold. Daedalus uses the threshold of one standard deviation. The anomaly detection model keeps track of the job's running mean and variance of the difference between workload and throughput using Welford's previously mentioned online algorithm. After a rescaling action, Daedalus checks for anomalies until the system has recovered. Because it takes time for a system to recover, the anomaly detection monitoring is run in a background thread to prevent interfering with the MAPE-K loop.

3.6 Implementation

Daedalus uses the proven MAPE-K control loop for self-adaptive systems to provide structured execution. The monitor, analyze, plan, and execute phases are structured as follows:

- **Monitor**: Daedalus collects metrics from Prometheus. From the DSP system, it collects the throughput for each worker measured by the number of records consumed by the source operator, CPU utilization of each worker using a moving average of one minute to reduce noise, and the overall consumer lag representing available, but not processed, tuples. For convenience, the job up-time and current parallelism are also collected. From the data source, the incoming workload rate is collected and measured in tuples per second.
- **Analyze**: The maximum capacity of each worker is calculated using CPU-throughput regression models, and the capacity for each scale-out is estimated. The ARIMA TSF and anomaly detection models are updated, and the future workload is predicted using the TSF model.
- **Plan**: The optimal scale-out is determined using the algorithm described in Algorithm 1. The chosen scale-out must be able to process the incoming workload and recover within the target recovery time. If rescaling is necessary, the scale-out must also be able to process the predicted workload.
- **Execute**: Any planned scaling action is executed by the Kubernetes client. The actual recovery time is then monitored with anomaly detection in a background thread.
- **Knowledge**: Knowledge represents the shared information between models. This is the collected metrics, capacity models, forecasts, anomaly detection, scaling actions, and recovery time information.

The MAPE-K loop runs every 60 seconds and takes on average one second to execute because of its low computational complexity.

4 EVALUATION

To demonstrate the effectiveness of Daedalus, it is evaluated with three DSP jobs and two DSP systems, Flink and Kafka Streams. The next sections describe the DSP jobs, the comparison systems, experimental setup, and results. This chapter concludes with a discussion that evaluates the overall performance of Daedalus.

4.1 DSP Jobs

Daedalus is evaluated with three representative DSP jobs: WordCount, Yahoo Streaming Benchmark, and Traffic Monitoring. All relevant code can be found in the Daedalus GitHub repository[2].

4.1.1 WordCount. WordCount is a popular DSP job that is readily available for different DSP systems, as it frequently serves as the exemplary tutorial to illustrate stream processing. Because of its simplicity and ubiquity, WordCount is often used to compare DSP systems [1]. WordCount computes a running total of word occurrences in a given text corpus. It takes lines of text as input, splits

[2]https://github.com/dos-group/daedalus

the line into words, and returns each word along with its cumulative word count. In order to test the job with dynamic workloads, WordCount has been modified to read input from a Kafka source. The output of words and their count are written to a console sink.

4.1.2 Yahoo Streaming Benchmark. The Yahoo Streaming Benchmark was one of the first benchmarks to evaluate major modern DSP systems including Apache Storm, Apache Spark, and Apache Flink [6]. Though created in 2016, it continues to be used to compare DSP systems and the pipeline serves as a baseline for further benchmarking jobs [20]. The pipeline features representative operations of stream processing jobs such as filtering, windowing, aggregation, and joining data with a database. The job is an advertising analytics use case that consists of deserializing JSON ad events from a Kafka source, filtering ads based on an event type and removing unnecessary fields, matching the ad to a campaign ID stored in Redis, and counting the number of times an ad was viewed within a ten second tumbling window. In the original benchmark, read and write operations to Redis became a bottleneck when operating at larger scales. For this reason, the job has been modified so that campaign IDs from Redis are cached in the DSP job and the resulting ad counts are written to a Kafka sink instead of Redis.

4.1.3 Traffic Monitoring. The Traffic Monitoring job is an IoT use case that calculates the average speed of moving vehicles in a particular radius adapted from the IoT Vehicles Experiment [9]. The job reads JSON vehicle events from a Kafka source, filters out events not contained within a radius of interest, calculates the average speed of vehicles in a ten second tumbling window, and enriches the vehicle information before outputting to a Kafka sink.

4.2 Workload Generation

In order to test the effect of dynamic workloads on autoscaling approaches, each job is run with a generator that produces a configurable amount of tuples per second to a Kafka topic. The data generators use the Akka actor system to simulate tuples in a highly scalable way. Generators are run inside the Kubernetes cluster to increase scalability and reduce the impact of network latency.

Each job is tested with a representative workload. Given its artificial nature, the workload for the WordCount job is a sine wave with two periods. The Yahoo Streaming Benchmark workload is taken from realistic online advertising click-through rate data[3]. Lastly, the traffic monitoring workload was generated based on the TAPASCologne scenario and SUMO to simulate realistic traffic patterns in the city of Berlin [9]. Each job was benchmarked to determine the maximum throughput achievable with 12 workers. All workloads have been scaled so that the maximum number of tuples is less than this throughput in order to more fairly compare autoscaling approaches to a static scale-out with 12 workers. Additionally, workloads are scaled to a duration of 6 hours. With these parameters, the workloads allow for a range of scaling decisions to test autoscaling approaches.

[3]https://www.kaggle.com/competitions/avazu-ctr-prediction, Accessed: March 2024

4.3 Comparison Systems

In order to demonstrate its usefulness, Daedalus is compared to a static deployment capable of processing the peak workload, HPA native to Kubernetes, and Phoebe, a recent state-of-the-art approach conceptually comparable to Daedalus.

4.3.1 Static Deployment. To serve as a baseline, a static deployment with 12 workers is deployed. As previously stated, this scale-out is capable of processing the peak workload for each job. It can therefore indicate if autoscaling approaches over-provision resources. While it is likely to have the highest resource consumption, it should also provide stable latencies because it will not rescale. It therefore provides a baseline latency comparison as well as showing the potential reduction of resource usage achieved through autoscaling.

4.3.2 Horizontal Pod Autoscaler. Kubernetes provides a built-in method for automatically scaling resources called a Horizontal Pod Autoscaler (HPA). A HPA monitors one or more metrics, such as CPU utilization or memory, and horizontally scales the target deployment in accordance with a user-defined policy. By default, the HPA monitors whether metrics violate the defined thresholds every 15 seconds. The HPA ignores instances that have not started yet in its calculation and waits for a default of five minutes between performing scaling actions to avoid flapping. While HPAs are commonly used and intuitive to understand, choosing a reasonable threshold to fairly compare with Daedalus is not simple. In line with Daedalus's objectives, the DSP system should target a high utilization to process the incoming workload with minimal resources while leaving extra processing capacity to recover from failure or rescaling actions. As shown in Figure 3, a system operating at full capacity does not necessarily use 100% CPU. Though Daedalus does not use CPU utilization thresholds to trigger scaling decisions, most scaling decisions occurred between 80% and 85% when testing with Flink. Therefore, two HPA deployments are tested. One targeting 80% utilization, and one targeting 85% utilization. With Kafka Streams, the HPA deployments target 60% and 80% utilization.

4.3.3 Phoebe. Phoebe is an approach fairly similar to Daedalus, and unlike many other DSP autoscaling approaches, its source code is publicly available[4]. Phoebe uses initial profiling runs to build QoS models and TSF to better meet QoS requirements. Similarly to Daedalus, it is capable of choosing worker parallelism in accordance with a target recovery time. Unlike Daedalus though, Phoebe explicitly models and accounts for latency. It also injects failures into profiling runs, measures the resulting recovery times, and incorporates them into its QoS models. Phoebe was implemented using Flink and also evaluated using the Yahoo Streaming Benchmark.

4.4 Experimental Setup

Experiments were run on a five-node Kubernetes cluster with ample CPU and memory, totaling 160 cores and 640 GB RAM. Details can be seen in Table 1. The experiments make use of Apache Kafka as a data source and sink. Kafka topics have been created with the same number of partitions as the maximum scale-out so that each worker can consume from its own partition. Prometheus is used as a time series database and periodically scrapes metrics from Kafka and

[4]https://github.com/dos-group/phoebe, Accessed: March 2024

Resource	Details
OS	Ubuntu 20.04.3
CPU	AMD64 Processor, 32 cores, 2.8GHz
Memory	128 GB RAM
Storage	3TB RAID0 (3x1TB, Linux software RAID)
Network	10 GBit Ethernet NIC
Software	Kubernetes v1.24.3, Docker v19.3, Java v1.11, Flink v1.16.0, Kafka v3.2, ZooKeeper v3.8, HDFS v2.8, Redis v6.2.7, Prometheus v2.39.1

Table 1: Cluster specifications

the target DSP system. Redis is used as a data store for the Yahoo Streaming Benchmark. Lastly, HDFS is used by Flink for saving checkpoints to storage.

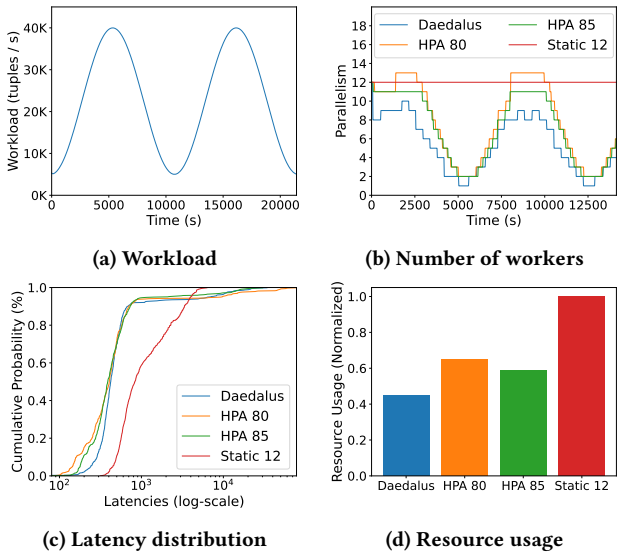
All approaches are deployed at the same time and read from the same Kafka source topic. For Flink deployments, each approach is deployed in application mode, ensuring resource isolation as there is no shared JobManager. The Flink deployments also make use of Flink's recently introduced reactive-mode,[5] a built-in method that allows elastic scaling. With reactive mode, Flink will automatically scale to the number of desired replicas and rescale the job from the last completed checkpoint. Defining the desired number of replicas is done using Daedalus or HPA. For both Flink and Kafka jobs, a custom metric was added to measure end-to-end latency. This metric measures the time from tuple generation until the end of processing, ignoring windowing periods. The 95th percentile latency is reported in the results. Across all our setups, we provided workers with significant memory access to preemptively ensure that memory constraints did not become a bottleneck. This was validated through empirical observations.

Approaches are evaluated by the quality of their scaling decisions. All jobs use exactly-once processing semantics and the DSP systems process all tuples. Therefore, end-to-end latency is a more suitable metric to indicate the processing performance. End-to-end latencies are reported by an empirical cumulative distribution function. Average latencies are also reported, though they can be misleading, since peaks of high latency during rescaling can skew an average upwards. While some papers exclude rescaling downtime in result calculations, this is misleading as recovery time is a crucial factor. Therefore, no metrics are excluded in the results.

4.5 Flink Experiment Results

This section contains the results for the Wordcount, Yahoo Streaming Benchmark, and Traffic Monitoring experiments run using Flink as the target DSP system. For these experiments, Daedalus used a recovery time of 600 seconds. Each experiment was executed five times to ensure consistency of the results. As the form of presentation of the results is repeated, we briefly explain the individual subplots on the basis of Figure 7: Figure 7a shows the workload over time; Figure 7b illustrates the number of workers over time as a result of the respective scaling method; Figure 7c presents the cumulative probability of observed latencies, which allows for statements

[5]https://flink.apache.org/2021/05/06/reactive-mode.html, Accessed: March 2024



(a) Workload **(b) Number of workers**

(c) Latency distribution **(d) Resource usage**

Figure 7: Flink WordCount results

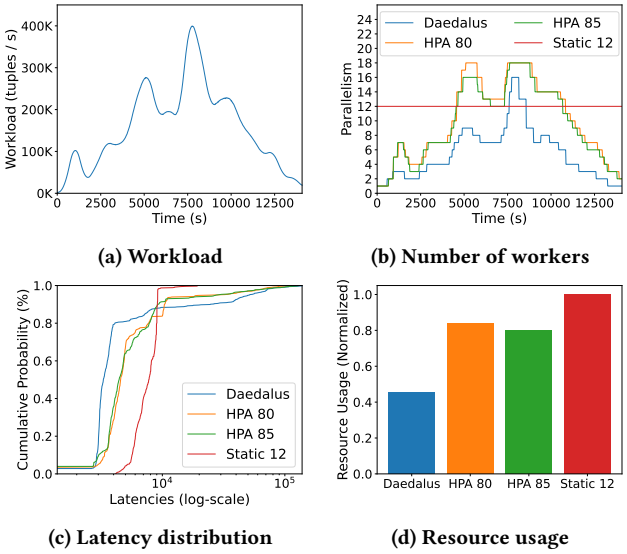
such as *"more than 80% of all latency values where smaller or equal* 10^3 *ms for Daedalus"*; Lastly, the total resource usage, normalized with respect to the static baseline, is displayed in Figure 7d.

4.5.1 WordCount Results. The results for the WordCount job are shown in Figure 7. Despite the job's simplicity, WordCount is highly susceptible to data skew, making it challenging to accurately estimate capacity for unseen workloads. In practice, this also means that the maximum observed capacity of workers at a specific scale-out can vary after rescaling to that scale-out again later in the job. Nevertheless, all autoscaling approaches were able to match the resources to the workload and process tuples in a timely manner. In general, the autoscaling approaches scale out around the same time, but Daedalus is able to scale in faster than the HPA methods.

As can be seen in Figure 7c, generally, all approaches perform very similarly with most latency measurements falling between 10^2ms and 10^3ms. Average latencies over the span of the job are as follows: Daedalus with 1,171 ms, HPA 80 with 1,791 ms, HPA 85 with 961 ms, and Static 12 with 1,408 ms. Notably, the static scale-out has proportionally slightly higher latencies. As found in previous research [24] and shown in Figure 2d, over-provisioning resources does not guarantee optimal latencies. Proportionally, the autoscaling approach latencies are quite similar. The larger increase in latencies above the 95% mark indicates when the systems are temporarily unavailable due to autoscaling.

When comparing resource utilization, Daedalus uses significantly fewer resources. On average, Daedalus used 5.4 workers, HPA 80 used 7.8, HPA 85 used 7.0, and the static scale-out naturally used 12. Overall, Daedalus used 55% less resources than the static scale-out, 31% less resources than the HPA 80, and 23% less resources than the HPA 85. So with fewer resources, Daedalus was able to achieve comparable latencies to HPA.

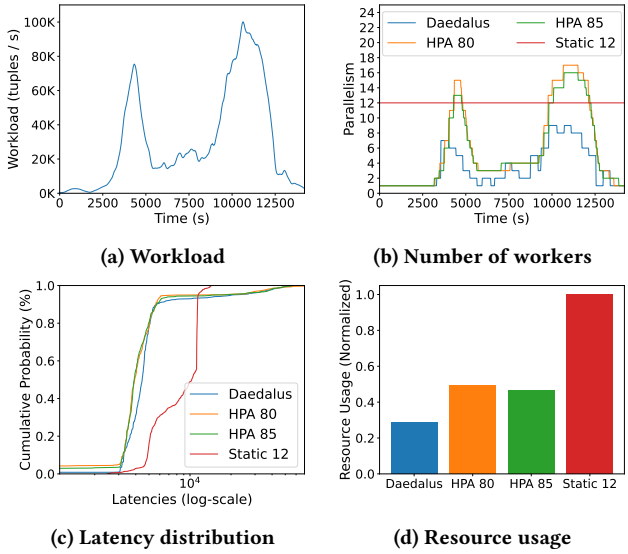
4.5.2 Yahoo Streaming Benchmark Results. The results for the Yahoo Streaming Benchmark are shown in Figure 8. Here, the HPA 80



(a) Workload (b) Number of workers

(c) Latency distribution (d) Resource usage

Figure 8: Yahoo Streaming Benchmark results



(a) Workload (b) Number of workers

(c) Latency distribution (d) Resource usage

Figure 9: Traffic Monitoring results

and 85 deployments scale very similarly. As can be seen in Figure 8b, they also allocate more workers than is necessary for a significant portion of the experiment. Both HPA deployments scale over 12 when the workload is around half of its maximum. Daedalus also over-provisions resources during the highest peak due to TSF predictions that the workload will continue to rapidly increase. As with the WordCount experiment, Daedalus scales in more quickly.

As seen in Figure 8c, Daedalus was proportionally able to achieve the lowest latencies while having slightly more downtime than the HPA deployments. Overall, average latencies for all approaches were similar, being within 1.5 seconds of each other. On average, Daedalus had 9,106 ms, HPA 80 had 7,862 ms, HPA 85 had 8,042 ms, and Static 12 had 7,576 ms. Also for this experiment, the latter did not proportionally achieve lower latencies. The highest latencies for the static scale-out come from when the workload is lowest.

On average, Daedalus used 5.5 workers, HPA 80 used 10, HPA 85 used 9.6, and the static scale-out naturally used 12. Daedalus used 54% less resources than the static scale-out, 45% less resources than the HPA 80, and 43% less resources than the HPA 85. Again, Daedalus was able to process all tuples with reasonable latencies using minimal resources.

4.5.3 Traffic Monitoring Results. The results of the Traffic Monitoring experiment can be seen in Figure 9. The major challenge of this workload comes from two large spikes where the workload rapidly increases and decreases. As with the Yahoo Streaming Benchmark, the HPA methods scaled similarly and allocated more workers than necessary. Again, Daedalus was able to scale to match the workload. It was able to react more quickly to the falling workload, scaling in faster than both the HPA approaches.

As seen in Figure 9c, for the majority of the job, Daedalus and the HPA methods had very similar proportional latencies. All autoscaling approaches had lower average latencies than the static scale-out: Daedalus with 6,176 ms, HPA 80 with 5,566 ms, HPA 85 with 5,671 ms, and Static 12 with 8,778 ms. As with the Yahoo Streaming Benchmark, the lowest latencies for the static scale-out occurred during the highest peaks of the workload.

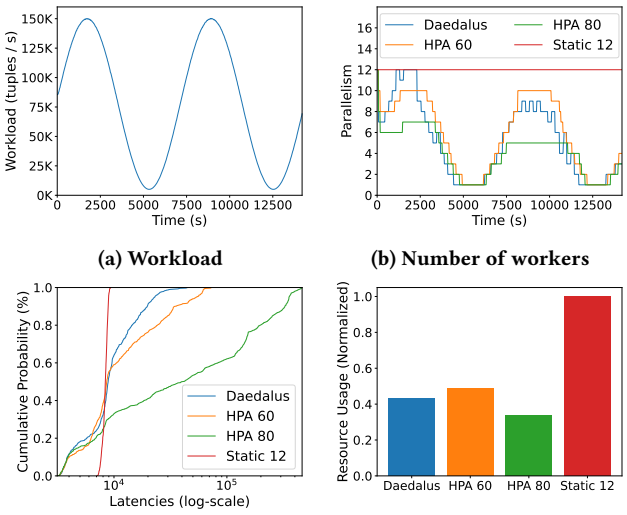
Daedalus also used fewer resources than the comparison approaches. On average, Daedalus used 3.5 workers, HPA 80 used 5.9, HPA 85 used 5.6, and the static scale-out naturally used 12. Percentually, Daedalus used 71% less resources than the static scale-out, 41% less resources than the HPA 80, and 38% less resources than the HPA 85. For all Flink experiments, Daedalus was able to achieve similar latencies with fewer resources.

4.6 Kafka Streams Experiment Results

To show that Daedalus is a general approach that can work with any DSP framework, it is tested with Kafka Streams using the WordCount job. The other two DSP jobs used so far, namely Yahoo Streaming Benchmark and Traffic Monitoring, do not qualify for this comparison as no implementations for Kafka Streams exist. The results are shown in Figure 10. As with the Flink WordCount job, the job is susceptible to data skew and the maximum capacity at a given parallelism is highly dependent on how data is split among workers. This is especially apparent when observing the peaks of the workload in Figure 10b. Unlike in the Flink experiments, HPA 80 was not able to process tuples in a timely manner and under-provisioned resources. This is also evident in the empirical cumulative distribution function.

The static workload had an almost constant latency and was able to achieve the best overall latencies with an average of 8,343 ms. Daedalus performed next best with an average of 10,566 ms. HPA 60 was slightly worse with an average latency of 15,453 ms. Lastly, the HPA 80, which was not able to match the workload had an average of 102,153 ms.

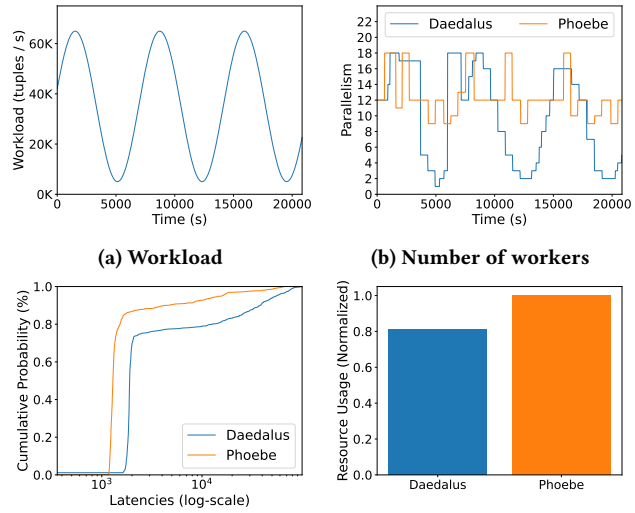
Compared to the deployments that were able to process the workload, Daedalus used fewer resources. On average, Daedalus used 5.2 workers, HPA 60 used 5.8, HPA 80 used 4, and the static



(a) Workload (b) Number of workers

(c) Latency distribution (d) Resource usage

Figure 10: Kafka Streams WordCount results



(a) Workload (b) Number of workers

(c) Latency distribution (d) Resource usage

Figure 11: Comparison with Phoebe

scale-out naturally used 12. Daedalus used 57% less resources than the static scale-out and 11% less resources than the HPA 60. From these results, one can infer that Daedalus is a generally applicable solution. With Kafka Streams, it was able to provide a stable level of service using minimal resources.

4.7 Comparison with Phoebe

The results of comparing Daedalus to Phoebe using the Yahoo Streaming Benchmark can be seen in Figure 11. For this experiment, a sine workload was chosen to compare Phoebe's scaling decisions to those in its paper. In addition, the recovery time target of 600 seconds was chosen, since lower recovery time targets (e.g. 180 seconds as in the original publication) caused Phoebe to stay primarily at the maximum scale-out of 18 with the tested workload.

When looking at the parallelism in Figure 11b, the scaling decisions of Phoebe do not appear to mirror the workload. However, when examining the logs, the scaling decisions are reasonable and balance achieving a minimum latency with a recovery time below the 600 second target. It should also be noted that these results do not match those from the initial paper, where the number of workers were more in line with the workload. In contrast, Daedalus scaled more frequently, but also more in line with the workload.

When examining latencies, Phoebe outperformed Daedalus by achieving proportionally lower latencies and faster recovery times. Daedalus achieved an average latency of 9,624 ms and a maximum latency of 88 seconds, while Phoebe achieved an average latency of 3,340 ms and a maximum latency of 65 seconds. The maximum latencies indicate the longest time that the system was unavailable, and both maximum latencies were under the target recovery time.

When comparing resources used during the autoscaling part of the experiment, Daedalus used 19% less resources, using an average of 10.1 workers while Phoebe used an average of 12.4 workers. However, Phoebe also requires initial profiling runs to build performance models. When incorporating profiling time, Daedalus used 53% less resources.

4.8 Discussion

The conducted experiments show that Daedalus achieves its goals: It allocates sufficient resources to process the incoming workload, processes tuples in a timely manner to achieve reasonable latencies, minimizes resource usage, and makes long-lived scaling decisions.

For methods such as HPA, adequate thresholds must first be determined. Even then, these fixed thresholds do not guarantee optimal resource usage or meeting QoS requirements. As an example, for Flink, the HPA 80 over-provisioned resources (i.e. competitive latencies in exchange for higher resource usage), while for Kafka Streams, it under-provisioned resources (i.e. lower resource usage in exchange for undesirable latencies). Seemingly, technical variations in DSP systems and the implementation of DSP jobs pose challenges to the HPA methods to achieve generalization. In addition, the HPA methods do not incorporate data skew, which occurred in these experiments, leading to suboptimal scaling decisions.

Comparing Daedalus to an approach like Phoebe highlights the trade-offs that can be made in autoscaling. By using initial profiling runs to build latency models, Phoebe can target scale-outs that result in minimal latencies. However, this comes at the cost of increased resource usage as a result of the required initial profiling runs. In contrast, Daedalus focuses on minimizing resource usage and builds its capacity models by monitoring a running job. It focuses on ensuring throughput and does not explicitly model latency. The autoscaling approach to use ultimately depends on the requirements of the DSP job. An approach similar to Phoebe is more appropriate when the primary goal is to minimize latencies as much as possible, while Daedalus is more suitable for optimizing resource efficiency. To further attain lower latencies, Phoebe manually creates a checkpoint before rescaling, minimizing the amount of tuples that need to be reprocessed. In contrast, Daedalus uses

Flink's reactive mode, which restarts the job from the last completed checkpoint. While this adaptation could improve achievable latencies, the implementation to configure a manual checkpoint before rescaling is dependent on the DSP system and would make Daedalus less generalizable.

The effectiveness of Daedalus relies on its ability to accurately ascertain worker capacity across all scale-outs. While it is difficult to truly determine the maximum capacity given factors like data-skew, it can be approximated by ensuring ample tuples in the data source to saturate the system and observing the throughput at different scale-outs. Comparing the observed capacities to the estimates from Daedalus gives insights into the estimation accuracy. Generally, the estimated capacities typically differ less than 5% from the observed capacities, with the majority between 0% and 3%. Therefore, it can be concluded that Daedalus can accurately estimate capacity.

Since the quality of scaling decisions are also impacted by TSF, the accuracy of TSF predictions was evaluated. TSF predictions were generally accurate with errors typically falling below 5%. In fact, the threshold for poor predictions at 25% was never reached.

In this evaluation, the recovery time heuristic was tested using the time needed to recover from a rescale. However, real failures typically incur longer recovery times since the DSP system must first detect the failure. While Daedalus accounts for failure by using a worst-case recovery time calculation, an evaluation that injects failures is left for future work. In general, a lower desired recovery time will lead to higher resource utilization, making recovery time the primary factor influencing autoscaling decisions. On the contrary, a higher desired recovery time will have less impact on scaling decisions, with actual processing capacity becoming the key determinant. To align with HPAs for comparison purposes, we opted for a higher recovery time of 600 seconds without exploring the boundaries or quantifying the precise influence of the recovery time parameter. In our experiments, the predicted recovery time was almost always greater than the measured recovery time. However, due to the worst-case calculation, the accuracy ranges wildly from a 1% difference to a 140% difference when comparing the actual and predicted recovery times. This lack of precision is a limitation to this approach and could be improved in future work.

5 CONCLUSION

This paper presents Daedalus, a self-adaptive autoscaling approach for DSP systems. Daedalus monitors running DSP jobs, builds worker-level capacity models using readily available CPU and throughput metrics, and scales resources to meet adaptation goals. Its primary objectives are to process incoming workloads efficiently while minimizing resource usage and to make long-lived scaling decisions to reduce system downtime. After evaluation with three representative DSP jobs in two DSP systems, Flink and Kafka Streams, Daedalus proves to be an effective autoscaling method. It accurately estimates the maximum processing capacity across different scale-outs and matches resources to the incoming workload. Using up to 71% fewer resources, Daedalus achieves latencies comparable to those of a static scale-out and HPA. When compared to Phoebe, a state-of-the-art approach that explicitly models latency, Daedalus does not quite achieve the same low latencies, nonetheless, it still provides a stable level of service with fewer resources than Phoebe. A current limitation of Daedalus is its inability to target scale-outs for minimal latencies, and incorporating a fitting latency model is considered a direction for our future work.

REFERENCES

[1] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Cláudio Fernando Resin Geyer, and Luiz Gustavo Fernandes. 2020. DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems. *IEEE Access* 8 (2020).

[2] Michael Borkowski, Christoph Hochreiner, and Stefan Schulte. 2019. Minimizing Cost by Reducing Scaling Operations in Distributed Stream Processing. *Proc. VLDB Endow.* 12 (2019).

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Data Engineering Bulletin* (2015).

[4] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Run-time Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Computing Surveys (CSUR)* (2022).

[5] Tao Chen, Rami Bahsoon, Shuo Wang, and Xin Yao. 2018. To Adapt or Not to Adapt?: Technical Debt and Learning Driven Self-Adaptation for Managing Runtime Performance. In *ICPE*. ACM.

[6] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *IPDPS*. IEEE Computer Society.

[7] John M. Ewing and Daniel A. Menascé. 2014. A meta-controller method for improving run-time self-architecting in SOA systems. In *ICPE*. ACM.

[8] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthikeyan Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. *Proc. VLDB Endow.* 10 (2017).

[9] Morgan Geldenhuys, Benjamin J. J. Pfister, Dominik Scheinert, Lauritz Thamsen, and Odej Kao. 2022. Khaos: Dynamically Optimizing Checkpointing for Dependable Distributed Stream Processing. In *FedCSIS*.

[10] Morgan K. Geldenhuys, Dominik Scheinert, Odej Kao, and Lauritz Thamsen. 2022. Phoebe: QoS-Aware Distributed Stream Processing through Anticipating Dynamic Workloads. In *ICWS*. IEEE.

[11] Kordian Gontarska, Morgan K. Geldenhuys, Dominik Scheinert, Philipp Wiesner, Andreas Polze, and Lauritz Thamsen. 2021. Evaluation of Load Prediction Techniques for Distributed Stream Processing. In *IC2E*. IEEE.

[12] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In *DEBS*. ACM.

[13] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava C. Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*. USENIX Association.

[14] Faria Kalim et al. 2019. Caladrius: A Performance Modelling Service for Distributed Stream Processing Systems. In *ICDE*. IEEE.

[15] André Martin, Tiaraju Smaneoto, Tobias Dietze, Andrey Brito, and Christof Fetzer. 2015. User-Constraint and Self-Adaptive Fault Tolerance for Event Stream Processing Systems. In *DSN*. IEEE Computer Society.

[16] Diego Perez-Palacin and Raffaela Mirandola. 2014. Uncertainties in the modeling of self-adaptive systems: a taxonomy and an example of availability evaluation. In *ICPE*. ACM.

[17] Jason Porter, Daniel A. Menascé, Hassan Gomaa, and Emad Albassam. 2018. TESS: Automated Performance Evaluation of Self-Healing and Self-Adaptive Distributed Software Systems. In *ICPE*. ACM.

[18] Henriette Röger and Ruben Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Computing Surveys (CSUR)* 52 (2019).

[19] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *BIRTE*. ACM.

[20] G. van Dongen and D. Van den Poel. 2020. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020).

[21] Adriano Vogel, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo Fernandes. 2022. Self-adaptation on parallel stream processing: A systematic review. *Concurrency and Computation: Practice and Experience* 34 (2022).

[22] B. P. Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* 4 (1962).

[23] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* (2016).

[24] Liang Zhang, Wenli Zheng, Chao Li, Yao Shen, and Minyi Guo. 2021. AuTraScale: An Automated and Transfer Learning Solution for Streaming System Auto-Scaling. In *IPDPS*. IEEE.

Demeter: Resource-Efficient Distributed Stream Processing under Dynamic Loads with Multi-Configuration Optimization

Morgan K. Geldenhuys
Technische Universität Berlin
Berlin, Germany
morgan.geldenhuys@tu-berlin.de

Dominik Scheinert
Technische Universität Berlin
Berlin, Germany
dominik.scheinert@tu-berlin.de

Odej Kao
Technische Universität Berlin
Berlin, Germany
odej.kao@tu-berlin.de

Lauritz Thamsen
University of Glasgow
Glasgow, United Kingdom
lauritz.thamsen@glasgow.ac.uk

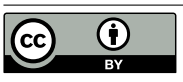
ABSTRACT

Distributed Stream Processing (DSP) focuses on the near real-time processing of large streams of unbounded data. To increase processing capacities, DSP systems are able to dynamically scale across a cluster of commodity nodes, ensuring a good Quality of Service despite variable workloads. However, selecting scaleout configurations which maximize resource utilization remains a challenge. This is especially true in environments where workloads change over time and node failures are all but inevitable. Furthermore, configuration parameters such as memory allocation and checkpointing intervals impact performance and resource usage as well. Sub-optimal configurations easily lead to high operational costs, poor performance, or unacceptable loss of service.

In this paper, we present Demeter, a method for dynamically optimizing key DSP system configuration parameters for resource efficiency. Demeter uses Time Series Forecasting to predict future workloads and Multi-Objective Bayesian Optimization to model runtime behaviors in relation to parameter settings and workload rates. Together, these techniques allow us to determine whether or not enough is known about the predicted workload rate to proactively initiate short-lived parallel profiling runs for data gathering. Once trained, the models guide the adjustment of multiple, potentially dependent system configuration parameters ensuring optimized performance and resource usage in response to changing workload rates. Our experiments on a commodity cluster using Apache Flink demonstrate that Demeter significantly improves the operational efficiency of long-running benchmark jobs.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Availability**; • **Computing methodologies → Massively parallel algorithms**.



This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645048

KEYWORDS

Distributed Stream Processing, Cluster Resource Management, Performance Profiling, Performance Modeling, Runtime Optimization, Time Series Forecasting, Cloud Computing

ACM Reference Format:
Morgan K. Geldenhuys, Dominik Scheinert, Odej Kao, and Lauritz Thamsen. 2024. Demeter: Resource-Efficient Distributed Stream Processing under Dynamic Loads with Multi-Configuration Optimization. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645048

1 INTRODUCTION

Distributed Stream Processing (DSP) is a paradigm in the big data domain that focuses on processing large streams of unbounded data in near real-time. As streaming workloads are typically dynamic in nature, DSP systems such as Apache Flink [7], Spark Streaming [37], and Storm [33] are designed to scale horizontally, distributing the load across multiple nodes within a cluster. When configured correctly, this enables high throughput, low latency, and fault tolerance in cloud-based environments. This is essential in areas such as real-time analytics, IoT data processing, click stream analysis, network monitoring, and more, where data needs to be analyzed on-the-fly continuously [18, 25, 26]. In these areas, ensuring minimal latencies is vital as the value of results is often greatest at the time of data arrival. Therefore, by maintaining the fastest response times, the maximum value of these results is captured, enabling timely decision-making. Likewise, in the event of failures, the ability to both recover quickly and maintain the consistency of results is important for preserving system reliability.

However, the manner in which these systems are configured has a significant impact on the Quality of Service (QoS) they are able to deliver as well as the resources they consume. Given the dynamic nature of cluster environments and streaming workloads, ensuring near-optimal configurations becomes inherently challenging. Relying on static configurations can lead to over-provisioning, wasting energy and resources, or under-provisioning. The manual fine-tuning of configuration parameters for individual streaming jobs is likewise impractical, involving considerable trial and error, and necessitates continuous adjustments to align with the ever-changing workload. As a result, configurations would quickly become outdated, leading to a diminished QoS and significantly

higher operational costs. It is evident that adaptive configuration optimization strategies are essential, as they can dynamically respond to changing workloads, ensuring both reliability and efficiency.

A number of methods have been proposed that optimize the configuration of DSP jobs adaptively. The underlying principle is to identify a configuration parameter that, when adjusted, can improve the performance of the DSP job. For instance, some methods focus on optimizing the scaleout [1, 2, 11–13, 17, 21, 27], while others concentrate on adjusting the checkpoint interval [8, 9, 20, 36]. They can broadly be categorized as either reactive or proactive. Reactive strategies [11, 12] typically rely on threshold monitoring, adjusting the parameter when the established upper or lower bounds are violated. However, these approaches are imprecise and prone to initiating re-configurations in response to transient conditions such as load spikes or failures. This can lead to unnecessary changes that disrupt the service. Alternatively, proactive strategies [1, 2, 13, 17, 21, 27] often employ modeling techniques, using historical data to predict a near-optimal configuration setting for the workload. Nonetheless, these methods are not without their challenges, especially when historical data is limited, complicating accurate predictions in dynamic environments.

Although these methods have shown improvements in certain aspects of system performance, the focus on a single parameter does not fully capture the complex interdependencies among various key configuration parameters that are common in real-world DSP systems. Take, for example, the concepts of scaleout and local parallelism. In a DSP system, the scaleout for each job is reflected by the total number of processing slots, and the ratio of processing slots to each worker node determines the level of local parallelism. Generally, previous methods have kept this ratio at one-to-one, resulting in the total processing slots equalling the number of workers. They do not explore how local parallelism might boost efficiency beyond identifying the optimal scaleout for a particular workload rate. Despite these interdependencies, the simultaneous optimization of multiple configuration parameters for enhancing resource efficiency in DSP systems remains largely unexplored.

In this paper, we introduce *Demeter*, a novel approach intended for cloud-based environments that dynamically optimizes multiple key configuration parameters for targeted DSP jobs. We focus on the following critical parameters:

- **No. of Workers:** Workers are responsible for orchestrating and supervising the execution of processing tasks. Their configuration affects the overall capacity of the system to handle parallel tasks and its resilience against failures.
- **CPU Cores:** These refer to the computational resources assigned to each worker. Workers receive an equal share of computation resources, ensuring homogeneous processing potential throughout the system.
- **Memory Allocation:** Memory is important for buffering incoming data streams, storing intermediate results, and facilitating various in-memory operations. As with the CPU cores, each worker is assigned an equal share of memory.
- **Processing Slots:** A worker can manage multiple processing slots, determining the number of concurrent processing tasks it can oversee. These slots utilize CPU and memory resources from the collective resource pool assigned to each worker.
- **Checkpoint Interval:** This involves periodic snapshots of the system's current state as a part of its fault-tolerance mechanism. The checkpoint frequency is important; while more frequent snapshots can facilitate faster recovery from failures, they also introduce overheads.

Demeter aims to find the best combination of settings for the aforementioned parameters, ensuring both near-optimal performance and resource efficiency in response to changing workloads. To achieve this, Demeter makes use of two statistical modeling techniques: Time Series Forecasting (TSF) for predicting workload rates, and Multi-Objective Bayesian Optimization (MOBO) to model runtime behaviors in relation to parameter settings and workload rates. From a high-level perspective, Demeter operates through two independent processes: *profiling* and *optimization*. Both processes execute iteratively and begin with a prediction of the upcoming workload rate. In profiling, when the MOBO models lack sufficient data for confidently selecting a near-optimal configuration at the predicted rate, Demeter initiates short-lived parallel profiling runs to gather data to enhance the models. On the other hand, during optimization, once the models are sufficiently informed about the predicted workload rates, Demeter selects appropriate configurations. The objective is to minimize resource usage such that processing latencies and recovery times remain within the bounds of a runtime-derived latency constraint and a user-defined recovery time constraint. With continued iterations, Demeter's understanding of how configuration impacts on performance at specific workload rates increases along with resource efficiency. Over time, as the models become more accurate, the need for profiling decreases, further decreasing overheads. We implement Demeter prototypically with Apache Flink and conduct experiments on a commodity cluster with established benchmark jobs to demonstrate its usefulness in comparison to two state-of-the-art methods.

2 APPROACH

In this section, we provide a detailed description of Demeter, explaining the general idea and its processes.

2.1 General Idea

Demeter's goal is the runtime optimization of a target DSP job, ensuring near-optimal performance and resource efficiency across dynamic workloads. We measure resource efficiency in terms of *resource usage* (U) and performance in terms of the *average end-to-end latency* (L_{avg}) and *recovery time* (R). The process is guided by two constraints: an *average end-to-end latency constraint* (LC) which is determined at runtime based on observed latencies; and a user-defined *recovery time constraint* (RC). Therefore, the objective is to minimize U while ensuring L_{avg} and R are always kept below LC and RC, respectively. The decision variables consist of a set of key configuration parameters: number of workers, CPU cores, memory allocation, processing slots, and checkpoint interval.

To achieve its goal, Demeter employs a proactive modeling strategy, gathering performance data through short-lived parallel profiling runs of identical jobs with varying configuration sets. This strategy is enabled by using two fundamental statistical modeling techniques: Time Series Forecasting (TSF) to predict upcoming workload rates and Multi-Objective Bayesian Optimization (MOBO)

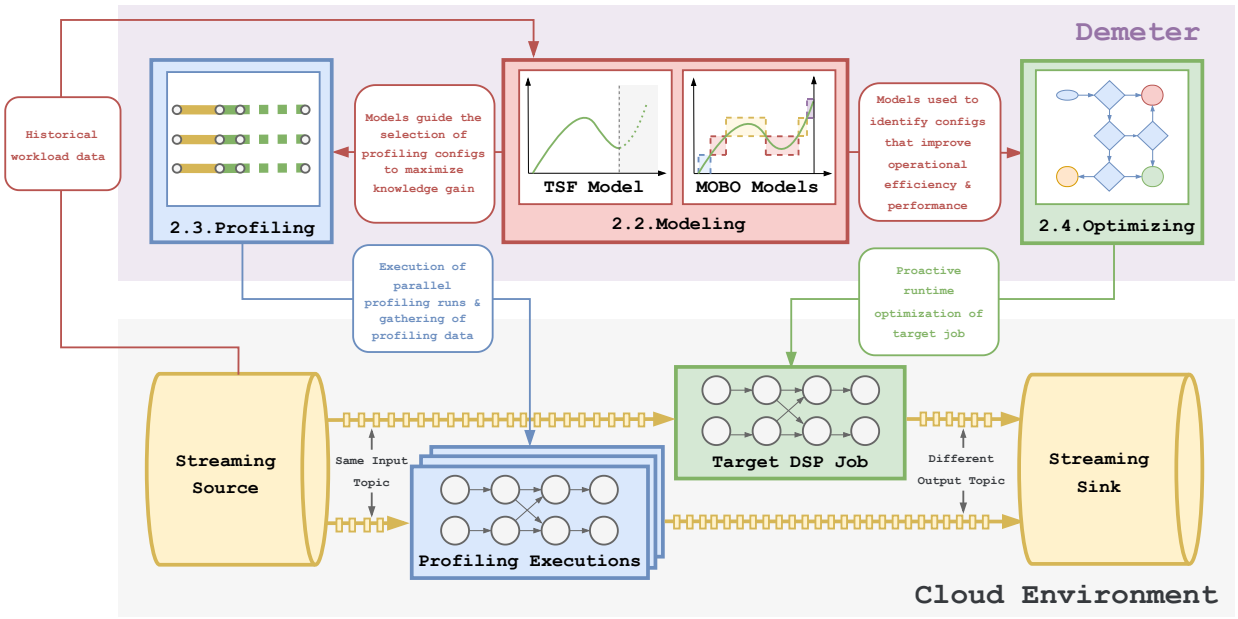


Figure 1: High-level representation of Demeter, illustrating the interplay between internal processes and external systems.

to model runtime behaviors in relation to parameter settings and workload rates. TSF grants insight into upcoming workloads, facilitating informed decision-making. This is important for preventing performance from degrading beyond a critical point before mitigating actions can be taken and ensures the longevity of configuration changes, thereby reducing the frequency of restarts. When working with exactly-once processing guarantees, restarts are expensive and introduce interruptions to the service. MOBO complements this by not only providing a means of simultaneously modeling multiple possibly competing criteria, but also a mechanism for guiding the exploration of the configuration search space. Together, TSF and MOBO establish a comprehensive approach for well-informed multi-configuration optimization under dynamic workloads.

Demeter is designed as a standalone client, interfacing with systems within a cloud-based environment. Our methodology is built on three foundational processes: Modeling, Profiling, and Optimizing. Next, we present an overview of Demeter's approach, accompanied by a graphical representation in Fig. 1. For any target job, a *maximum configuration* ($C_{\max}$) is defined, where parameters are set to allocate a large amount of resources, thereby guaranteeing consistent high performance in terms of L_{avg} and R across any reasonable workload[1]. After initiating the target job with $C_{\max}$, two iterative processes begin executing asynchronously. The first process focuses on profiling. Based on the predicted workload rate, the need for profiling is evaluated through a series of MOBO models, with each model dedicated to a particular configuration parameter. When required, they suggest configurations that maximize information gain, which are then applied in parallel profiling runs. This approach enables efficient exploration of the large configuration space. After these runs, data is collected to update the models. If profiling is deemed unnecessary, it is skipped. The second process focuses on optimization. It uses the TSF prediction and the MOBO models to check whether sufficient information is available for job optimization at that workload rate. If a more efficient configuration is found, a reconfiguration is initiated. If no better configuration is available according to our models but the existing setup can manage the upcoming workload, it remains unchanged. In situations where the current configuration is inadequate, or if there is insufficient information, such as encountering a new workload rate, the system reverts to the $C_{\max}$ configuration, unless it is already in use.

[1]Here, 'reasonable' refers to workloads within an upper bound determined by expert knowledge.

2.2 Modeling

In this section, we provide a description of the modeling techniques used in our approach. As previously stated, Demeter follows a proactive strategy, using predictions of future workload rates to inform and guide the profiling and optimization processes. To achieve this, we firstly use a multistep-ahead TSF model, trained on historical data. This model not only provides insights into the expected workload rate at a specific time horizon but also reveals the behavior of the rate over time leading up to this point. Understanding both the expected future workload rate and its behavior over time is important for our method's effectiveness. Whenever a forecast is generated, it is partitioned into separate averaging bins and the bin with the highest average value is calculated and selected. The value of this bin will be used for all subsequent profiling and optimization processes. This ensures that if an increase in workload is anticipated, the system will select the furthest bin for analysis, guiding the profiling and optimization efforts to address higher workload rates expected in the future, rather than focusing on the current point in time. Conversely, if the workload is anticipated to decrease, the model selects the nearest bin, focusing the profiling and optimization efforts on lower workload rates that are expected

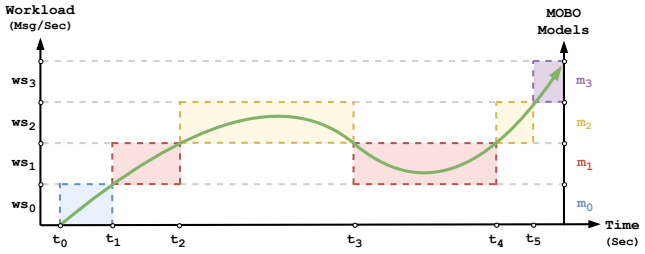


Figure 2: Model generation and selection based on observed workload rates, categorized by specific workload segments.

to occur closer to the current point in time. This is done to ensure that reconfigurations are valid at least up until the forecast horizon, thereby increasing their longevity. For our method, we use an online ARIMA model for workload predictions.

Addressing the challenge of optimizing multiple parameters, our approach shifts focus towards exploring a discrete search space, represented as $\mathbf{x}$. This space consists of the Cartesian product of all relevant parameters, each with its discrete set of potential values. Initially, we have no prior knowledge of this space. In anticipation of future streaming workloads, our primary goal is to identify an optimal configuration within $\mathbf{x}$. Such a configuration should not only comply with predefined constraints, including *LC* and *RC*, but also demonstrate resource efficiency, in terms of CPU and memory utilization. To address this challenge, we adopt MOBO, where each objective and constraint is represented using an individual Gaussian process. The optimization procedure aims to maximize the sum of their marginal log likelihoods. In this context, we weight the expected improvement of one or more objectives by the probability of feasibility, considering all the modeled constraints. Ultimately, the MOBO model is proficient in identifying near-optimal configurations for scenarios characterized by low workload variance. However, its performance diminishes when dealing with streaming workloads that present a wider range of variance, highlighting the need for a more adaptable optimization strategy.

To manage workload variability, we apply the concept of Rank-Weighted Gaussian Process Ensembles (RGPE), a method proven effective in similar scenarios [28, 38]. All collected observations, comprising various configurations and their corresponding performance metrics, are organized into K segments based on workload rate, denoted as $WS = \{ws_i\}_{i=0}^{K}$. The size of each workload segment is defined by the *segment size* (*SS*) hyper-parameter, and segments are added dynamically as they are detected. We train a set of MOBO models m_i for each segment ws_i using the included observations D_i. Referring to Fig. 2, we present an example of a variable workload over time. As Demeter identifies new workload segments ws_0, ws_1, ws_2, and ws_3 at time instances t_0, t_1, t_2, and t_5, it correspondingly creates MOBO models m_0, m_1, m_2, and m_3 to model the configuration parameters for these segments. Following this, whenever the workload prediction falls within a segment's range, the system utilizes the associated MOBO models for both profiling and optimization purposes. These models then contribute to approximating the target MOBO model m_{tar} for the anticipated streaming workload in an ensemble manner:

$$m_{tar}(\mathbf{x}|D_{tar}) \sim \mathcal{N}\left(\sum_{ws_i \in WS} a_i \mu_i(\mathbf{x}), \sum_{ws_i \in WS} a_i^2 \sigma_i^2(\mathbf{x})\right)$$

In this formulation, μ_i and σ_i^2 are the mean and variance parameters of the BO model m_i, tailored for segment ws_i, while a_i represents the corresponding weight within the ensemble, defined by a ranking loss specific to RGPE. This approach offers several benefits, including the ability to leverage previously trained models, effectively addressing the cold-start issue by utilizing existing knowledge, even if partial, of the configuration search space. Concurrently, these *support* models play a crucial role in pinpointing the optimum of the desired function. Consequently, we expedite the identification of promising configurations by utilizing previously profiled configurations across varied workload rates. So, our outlined approach, combined with RGPE, allows us to leverage existing knowledge for more informed configuration recommendations. For simplicity, in this paper, the RGPE ensemble model, target MOBO models, and base MOBO models for a specific segment are collectively referred to as the MOBO models for that segment.

2.3 Profiling

The profiling process evaluates the need for profiling, selects fitting configurations, and supervises execution. Operating within a time-delay loop, the profiling algorithm begins each iteration by consulting the TSF model for a workload prediction. This prediction then guides the selection of configurations by referencing the MOBO models corresponding to the specific workload segment. For generating profiles, the method focuses on configurations with the highest expected hyper-volume improvement, essentially those with the best knowledge acquisition value. The number of configurations chosen for profiling aligns with the resources assigned for this task within the cluster. Within each workload segments, an annealing factor adjusts this number, leading to a reduced extent of exploration as more knowledge about that segment is accumulated.

Additionally, the system uses domain knowledge from past configurations to make strategic profiling decisions. For example, if during a previous re-configuration, a selection was found to be unsuitable for a workload rate similar to the prediction, and the system had reverted back to the $C_{\max}$, it would then give preference to profiling configurations that have greater resources than the previously unsuitable one. On the other hand, should historical data show an earlier decision to downscale, the system will prioritize configurations that use fewer resources for the projected workload. This approach allows the system to streamline its selection process, capitalizing on historical data and specialized expertise.

Upon determining suitable profiles for the predicted workload, jobs are deployed in parallel with their own unique configurations. A graphical representation of the profiling process can be seen in Fig. 3. Each profile is set to read from the same data source as the target job, yet directs its output to a separate temporary sink. Once deployed, a stabilization phase is allowed for metrics to achieve equilibrium. After stabilization, a fixed duration of standard execution is observed, after which the L_{avg} are computed. Next, timeout failures are injected into the profiling jobs to measure how long each job takes to recover. Demeter will then continuously

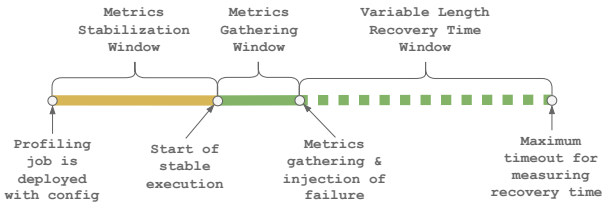


Figure 3: Profiling lifecycle from deployment to recovery to recovery time measurement.

re-evaluate the status of the jobs, ensuring they either attain a full recovery or exceed a designated maximum timeout constraint. In order to measure the recovery time, Demeter monitors specific metrics related to the recovery process:

- **Input Throughput:** This metric indicates the total number of events consumed by the source operators of the DSP job every second. It provides a measure of the system's capacity to handle incoming data.
- **Average Consumer Lag:** This metric represents the accumulated events in the messaging queue, still awaiting consumption by the source operators of the DSP job. It gives an indication of any potential backlog.

These metrics are used to train an anomaly detection algorithm on positive executions, i.e. let the function $s : X \to X$ perfectly represent the metrics data stream such that for any given data point $x \in X$ the prediction is always $s(x) = x$. Given that the majority of data collected within the standard execution period is expected to be normal, this approach allows the algorithm to recognize deviations from the norm. Should these deviations surpass a predefined threshold, derived from past errors, the system is deemed to be in an anomalous state. The length of time spent in this anomalous state therefore is equivalent to the recovery time. To implement this, we utilize an online ARIMA method as proposed in [30].

Importantly, in this context the recovery time encompasses more than just the period during which the system is in an inconsistent state before processing resumes. For systems using checkpoint and rollback recovery strategies, processing restarts from a previously saved offset. It then works to catch up to the latest offset, even as new events keep arriving. We aim to measure the entire duration – from the moment the failure begins until processing has once again caught up to processing events at the latest offset. This provides a more accurate measure of system availability for steam processing.

Upon the completion of profiling, the associated jobs are terminated, and metrics relating to workload, throughput, and latency are subsequently used to update the model.

2.4 Optimizing

The optimization process is tasked with tuning the configuration parameters of the target job in relation to changing workloads. Central to this process is the optimization algorithm, which operates within a time-delayed loop to periodically assess the state of the target job. Intervals between evaluations are essential, particularly

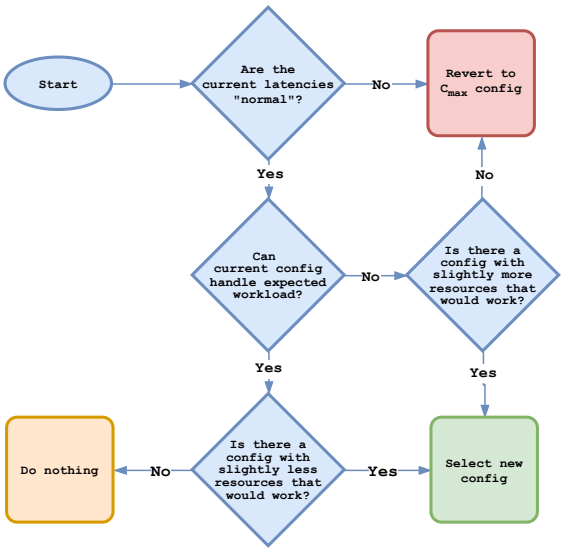


Figure 4: Flow chart depicting the optimization algorithm of the Optimizer component.

following events like re-configurations or failures that both require restarts. They ensure that metrics stabilize between evaluations and increase the longevity of changes. A simplified overview of this algorithm can be seen in Figure 4. After each interval, metrics from the target job are collected and the current L_{avg} is evaluated. Unlike our other constraints and objectives, determining whether the L_{avg} aligns with typical expectations presents a challenge, primarily because 'normal' latencies vary significantly, changing from one job and environment to another. To address this, we utilize the MOBO models, which are trained on the specific current workload rate segment, in combination with a clustering technique to establish a benchmark for near-optimal latencies.

Firstly, we need to identify two clusters among the latencies observed so far: those considered normal and those considered abnormal, and hence invalid. If a configuration resulted in the job being able to keep up with the current workload, its latencies would stabilize around the shortest possible lengths. Thus, the cluster with the smallest centroid represents configurations that yield near-optimal latencies. In order to do this, we start by normalizing the values according to the first percentile. Then, we apply a transformation to the values so that they fall within the $[0, 1]$ interval. Values smaller than 0.5 are considered normal, while those equal to or greater than 0.5 are considered abnormal. This technique allows us to define the latency constraint LC. If the current L_{avg} falls outside the range of LC, the job is assumed to be unstable and reverts to the C_{max} configuration, ensuring a rapid return to near-optimal operation.

Alternatively, if L_{avg} falls within acceptable limits, there exists an opportunity to further improve resource efficiency. In both cases, we transform all latency values according to the observed clusters, establishing a notion of normal and invalid latencies. The next step involves retrieving a workload prediction from the TSF model. We now aim to identify the near-optimal configuration that aligns most closely with the predicted workload. Using this workload

prediction, the algorithm references the MOBO models for the corresponding workload rate segment. We then use the models to retrieve predictions L_{avg} and R for a list of possible configurations.

We first filter out configurations with invalid L_{avg} as well as those where the R is greater than the RC, ensuring we retain only valid configurations that do not violate either constraint. Next, we sort the list of valid configurations based on their resource usage, where the smallest is predicted to result in the highest operation efficiency. At this point, we introduce a *safety buffer* (*SB*) hyper-parameter. Instead of directly selecting the most efficient configuration, *SB* is applied to scale our selection, effectively moving us up the sorted list by a certain percentile. For instance, if *SB* is set to 30%, we would skip the bottom 30% of configurations and choose the one at the 30th percentile mark, or with at least a 30% increase in resource usage. This provides a margin of safety, ensuring that our selected configuration is not too close to the lower bound of the system's requirements. Furthermore, this buffer approach allows us to account for and negate intermittent fluctuations in the cluster, which could otherwise trigger unwarranted re-configurations, optimizing for both efficiency and system stability.

Following our selection process informed by *SB*, the need for re-configuration is evaluated against the *efficiency threshold* (*ET*) hyper-parameter. Reconfiguration is triggered only if the expected improvement in resource efficiency exceeds *ET*. For example, setting *ET* at 5% means reconfiguration is only done when the resource saving is equal to or greater than 5%. Therefore, if the proposed configuration fails to meet this threshold, the current configuration remains unchanged, and the current iteration concludes. Conversely, if the threshold is exceeded, signifying a potential reduction in resource usage, reconfiguration is initiated. In instances where a predicted configuration is not available, potentially due to a lack of sufficient observations for modeling, a re-configuration to the C_{max} configuration is prompted, provided it is not already in place.

3 EVALUATION

In this section, we assess the effectiveness of Demeter. We detail our experimental cluster setup, outline the methods evaluated, and specify how they were configured. Two distinct experiments are presented, followed by a comparative analysis. All materials related to our prototype, datasets, and tools are available in our repository[2]

3.1 Experimental Setup

Our experimental setup was based on a co-located 5-node Kubernetes [35] and HDFS [31] cluster with all servers interconnected by a single switch. We developed and implemented a prototype to work with Apache Flink. Additionally, we use the Flink Kubernetes Operator[3] to automate deployments and upgrades of streaming jobs. We configured an Apache Kafka [22] cluster, serving as both the sources and sinks for the streaming jobs, with 24 partitions and a replication factor of 3. All sources and sinks of the experimental processing pipelines were configured to use *exactly-once* processing thereby guaranteeing the consistency of results [4]. For all experiments and methods, a maximum parallelism of 24 was set, meaning that the number of processing slots, or task managers, could not exceed 24 at any given time. Additionally, we set a 20s timeout interval for Flink task managers. For end-to-end latencies, measurements were taken over a 1-minute averaging window, with a focus on the 95th percentile to minimize the impact of outliers during periods of stable operation. Each experiment was conducted three times with the median result being selected for further analysis and discussion. Chaos Mesh[5] was used for injecting failures into the Kubernetes pods. During each experiment, Chaos Mesh injected 23 timeout failures at regular 45-minute intervals ensuring a uniform distribution of failures across a broad range of workload rates. Prometheus[6] was used for metrics collection. Cluster node specifications and software versions are summarized in Table 1.

Table 1: Cluster Node Specifications

Resource	Details
OS	Ubuntu 20.04.1
CPU	AMD EPYC 7282 16-Core Processor, 32 cores, 2.8 GHz
Memory	128 GB RAM
Storage	2TB RAID0 (2x1TB SSD, software RAID)
Network	10 GBit Ethernet NIC
Software	Java v11, Flink v1.17, Flink Operator v1.6, Kafka v3.4, Docker v19.3, Kubernetes v1.26, HDFS v2.8, Redis v5.0, Prometheus v2.25, Chaos Mesh v2.1, pmdarima v2.0.4, BoTorch v0.6.0

3.2 Demeter Setup

In configuring Demeter, the C_{max} configuration was allocated a maximum scaleout of 24, dedicating a full CPU core and 4096 megabytes of memory to each taskmanager, as well as a single processing slot. The *segment size* (*SS*), *safety buffer* (*SB*), and *efficiency threshold* (*ET*) hyper-parameters were set to 10.000, 30%, and 5%, respectively. The configuration space for profiling was defined by setting lower and upper bounds on each parameter, yielding 2592 distinct parameter combinations for each workload segment, as shown in Table 2. We utilize BoTorch [5] for MOBO modeling as part of our prototype.

Table 2: Configuration Parameter Search Space

Parameter	Min	Max	Step
Workers	4	24	4
CPU Cores	1	3	1
Memory Allocation (mb)	1024	4096	1024
Processing Slots	1	4	1
Checkpoint Interval (s)	10	90	10

Profiling runs incorporated a 2-minute stabilization period and a 1-minute latency measurement window. We found that the best

[2] https://github.com/dos-group/demeter

[3] https://nightlies.apache.org/flink/flink-kubernetes-operator-docs-release-1.6, Accessed: March 2024

[4] https://flink.apache.org/2018/02/28/an-overview-of-end-to-end-exactly-once-processing-in-apache-flink-with-apache-kafka-too/, Accessed: March 2024

[5] https://chaos-mesh.org, Accessed: March 2024

[6] https://prometheus.io, Accessed: March 2024

possible recovery times with $C_{\max}$ generally span between 90 and 120 seconds, even in scenarios of low workloads. This insight led us to set the recovery time constraint (*RC*) at 180s for our experiments, establishing the upper limit for acceptable recovery times. To accommodate for potential deviations, we defined a maximum timeout of 360s. For the optimization phase, the system was configured to perform evaluations every 10 minutes, with a 10-minute time horizon set for the TSF model. To predict future workloads, we employ online ARIMA for our TSF model using the pmdarima[7] python library. ARIMA was selected for its proven accuracy and efficiency in forecasting streaming workloads, as detailed in our previous research [16], and is favored due to its low computational demand and minimal data requirements.

All of these default settings are designed to cover a wide range of execution scenarios, providing good performance and should suffice for the majority of users without requiring further adjustments.

3.3 Baselines Setup

For our comparative analysis, we included a static configuration alongside two state-of-the-art baselines, both of which are interoperable with Apache Flink. As these methods do not focus on optimizing CPU an memory allocations, we assign a full CPU core and 4096 MB of memory to all taskmanagers. Moreover, all baseline methods consistently used a 10s checkpoint interval.

3.3.1 Static Configuration. In our experiments, we employed a static configuration with 24 worker nodes which aligns with Demeter's $C_{\max}$ configuration. This representing the maximum available resources, guarantees the best latencies and recovery times due to the highest resource allocation. This baseline serves as the standard for comparing all other methods.

3.3.2 Flink Reactive. The first method we evaluated was Apache Flink's reactive mode scheduler [2]. This scheduler dynamically optimizes cluster resources by adjusting to workload variations. Monitoring each worker's performance, any deviation from a specified utilization threshold prompts the scheduler to restart the DSP job from its last successful checkpoint with an altered scaleout. Configured to work with the Kubernetes Horizontal Pod Autoscaler (HPA)[8], our experiments set the HPA to target a CPU utilization of 35%, aligning with the recommended setup from the reactive mode documentation. We experimented with several higher utilization targets, but they consistently yielded inferior results.

3.3.3 DS2 Autoscaler. The second method we evaluated was the DS2 Autoscaler [21], a solution designed for the dynamic scaling of DSP jobs. It integrates historical data with real-time metrics, employing forecasting techniques to proactively predict workload variations. The system then makes informed scaling decisions, considering both current system state and anticipated workloads. This is achieved by optimizing operator parallelism, which adjusts the number of vertices in the execution graph to ensure efficient data processing. For our evaluation, we used the implementation provided by the Flink Kubernetes Operator, configuring it in such a way as to align it with the other methods. The system used a stabilization interval of 2 minute and metrics were aggregated over 1-minute windows. The target utilization was set at 35% and a boundary of 15%. Following any scaling adjustments, a 1-minute restart period was observed, and a 5-minute catch-up duration is assumed to guarantee system equilibrium.

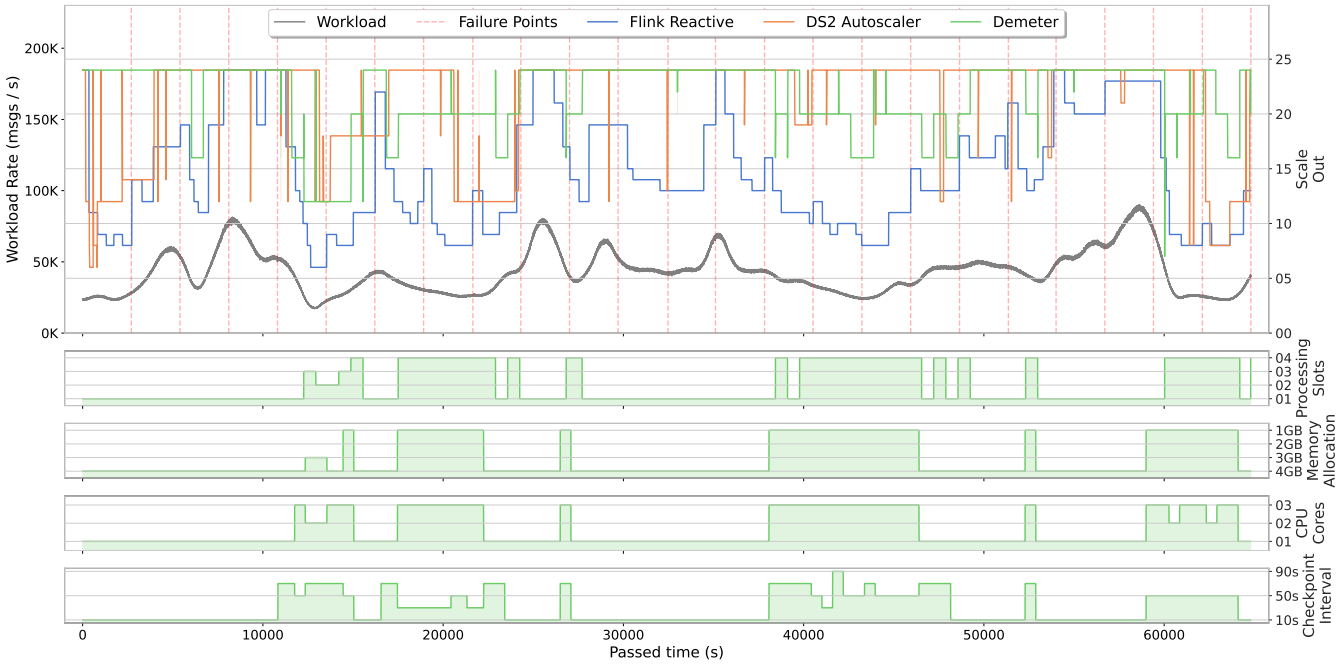
3.4 Experiments

For comparison, two experiments were conducted using established benchmark jobs and real-world workload simulations.

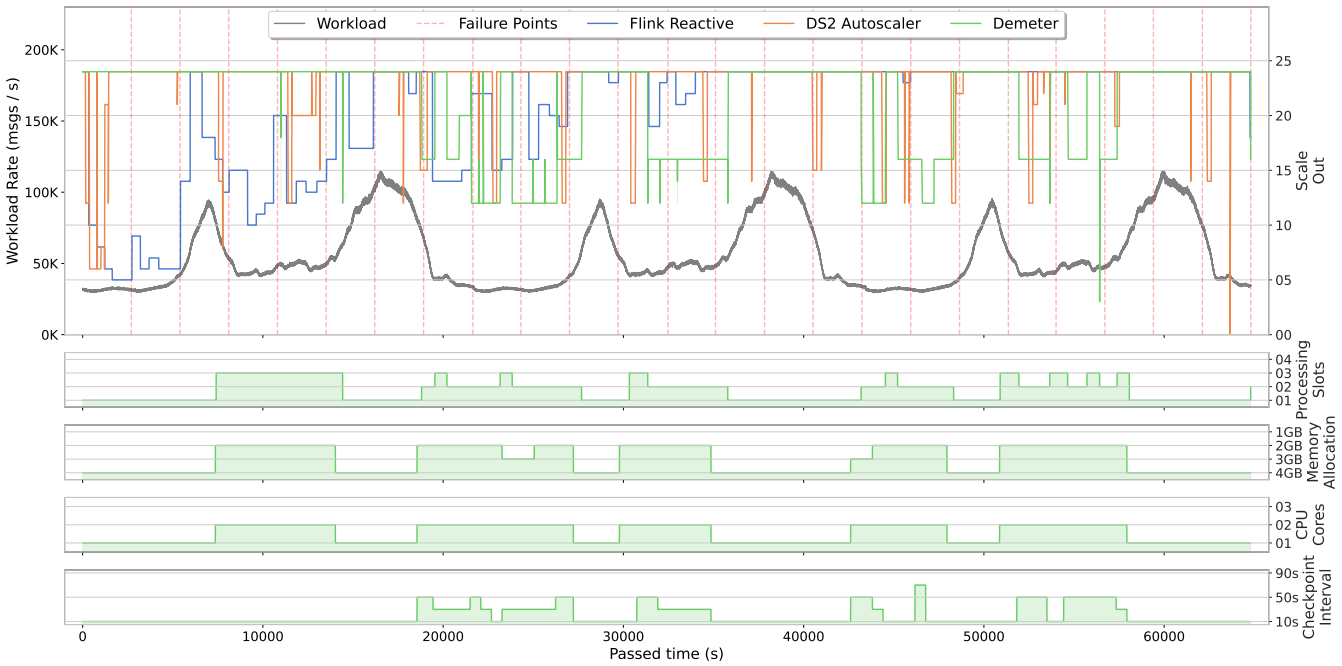
3.4.1 Yahoo Streaming Benchmark (YSB) Experiment. For our first experiment, we used the Yahoo Streaming Benchmark[9]. This benchmark simulates a streaming advertisement job, structured with multiple advertising campaigns, each containing several individual advertisements. Streaming sources retrieve events from a Kafka topic, identify relevant events, and aggregate a windowed count of these events, grouped by campaign. A key component of this setup was the deployment of a Redis cluster[10], which managed campaign and advertisement data, streamlining event generation and data aggregation processes. To align the benchmark with our objectives, we enabled checkpointing and replaced the native windowing functionality with the standard Flink implementation. We designed a data generator, using a click-through rate dataset[11], to emit events characterized by attributes such as *event_time*, *event_type*, and *ad_id*. From the initial 10-day dataset, we extracted a 3-day segment and sub-sampled every 4^{th} data point, resulting in a dataset that spans 18 hours. A graphical representation of the generated workload is provided in Figure 6(a). This dataset is characterized by high variability, covering a wide range of processing rates, and lacking a discernible long-term trend.

3.4.2 Top Speed Windowing (TSW) Experiment. For our second experiment we used a DSP job derived from the official Flink repository[12]. The primary focus of the job is on grouped stream windowing, enabling the application of diverse eviction and trigger policies. Each car-related event consists of attributes such as a unique number plate, the current speed (km/h), the total elapsed distance (meters), and an associated timestamp. The goal of the job is to determine the top speed of each car over a span of 50 meters, using only data from the immediate past 10 seconds. The job was modified to enable it to consume events from and publish results to separate Apache Kafka topics. To simulate workload variations, represented by the changing number of vehicles over time, we used the Sumo simulation tool to generate a 24-hour workload dataset, specifically employing the TAPASCologne scenario[13]. Similar to the YSB experiment, we reduced this to 18 hours by sub-sampling every 4^{th} data point and then repeating the resulting workload three times. This resulted in a dataset characterized by a clear seasonal pattern, with workload rates fluctuating within specific ranges, and a weak upward trend over time. A graphical representation can be seen in Figure 6(b). We created a generator program which would produce events constrained by this dataset.

[7] https://pypi.org/project/pmdarima/, Accessed: March 2024
[8] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale, Accessed: March 2024
[9] https://github.com/yahoo/streaming-benchmarks/, Accessed: March 2024
[10] https://redis.io/, Accessed: March 2024
[11] https://www.kaggle.com/c/avazu-ctr-prediction, Accessed: March 2024
[12] https://github.com/apache/flink/; Accessed: March 2024
[13] https://sumo.dlr.de/docs/; Accessed: March 2024

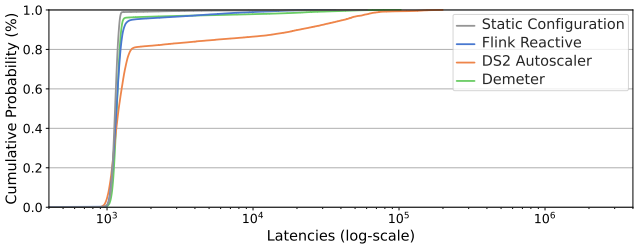


(a) YSB experiment: Workload rates & all configuration changes.

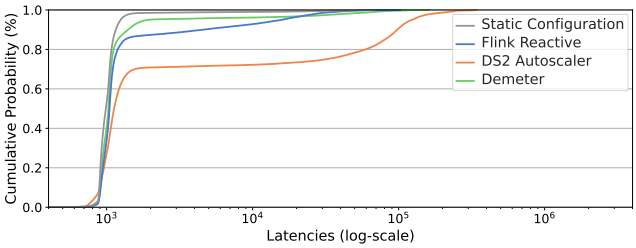


(b) TSW experiment: Workload rates & all configuration changes.

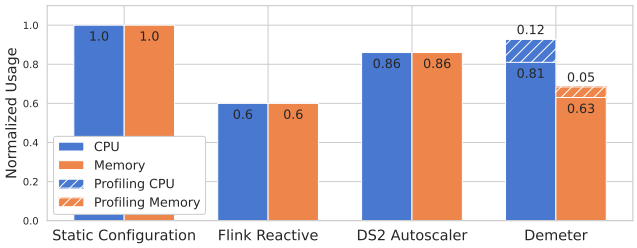
Figure 5: Workloads, failure injections, & configuration changes for Demeter & state-of-the-art approaches.



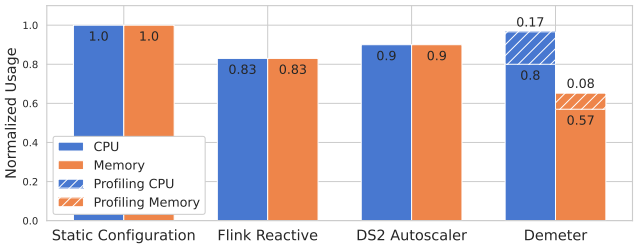
(a) YSB experiment: Average end-to-end latencies over time.



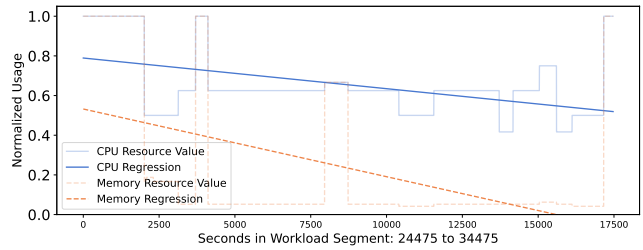
(b) TSW experiment: Average end-to-end latencies over time.



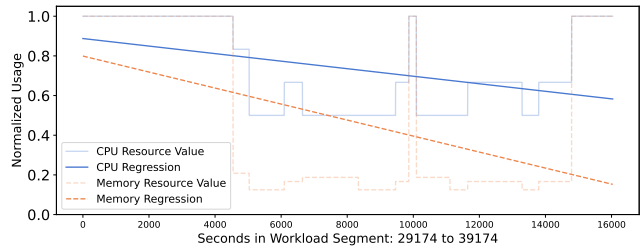
(c) YSB experiment: Total CPU & memory usage.



(d) TSW experiment: Total CPU & memory usage.



(e) YSB experiment: Resource usage for largest segment with regression.



(f) TSW experiment: Resource usage for largest segment with regression.

Figure 6: Overview of performance comparison results for Demeter & state-of-the-art approaches.

3.5 Experimental Results

After completing the experiments, we analyzed the performance metrics, with Figures 5(a) and 5(b) showing the workload rates over time and scaleout decisions for all methods. The data shows that Demeter tends to favor higher scale-out values, aligning with expectations as newly encountered workload rates trigger the use of C_{max}, and the safety buffer hyper-parameter increases the optimal amount of resources by 30%. Moreover, an increase in CPU usage was consistently observed in the TSW experiment across all executions, including static configurations. Further analysis identified a statistically significant weak upward trend within this dataset, likely explaining the observed increase in CPU usage.

3.5.1 Average End-to-End Latency Results: Our analysis first examines the L_{avg} of the static baseline, Demeter, and comparative methods, shown in Figures 6(a) and 6(b) using an empirical cumulative distribution function. The static configuration consistently had latencies near 1000ms, a high proportion of optimal latencies. Among the optimization methods, Demeter led with the most latencies in the optimal range, followed by Flink Reactive and then DS2. In the YSB experiment, Demeter and Flink Reactive achieved near-optimal latencies in about 95% of cases, compared to DS2's 80%. In the TSW experiment, Demeter maintained over 95% in optimal latencies, with Flink Reactive at about 85% and DS2 at 70%.

3.5.2 Recovery time Results: Recovery times were manually measured by analyzing consumer lag and throughput rate metrics (refer to Section 2.3). The results, detailed in Table 3, use color highlights to indicate performance: green for recovery times under the 180s recovery time constraint, yellow for exceeding this constraint, and red for surpassing the 6-minute maximum. The static configuration, with maximum resources, set a benchmark with the fastest recovery times, averaging 96s in the YSB experiment. Demeter showed a minimal deviation in recovery times of 3.21% compared to this benchmark, while Reactive and DS2 had deviations of 82.79% and 77.59%, respectively. In the TSW experiment, the static configuration's average recovery time was 107s, with Demeter at a 5.17% deviation, maintaining consistent performance. Reactive and DS2 had higher deviations of 62.78% and 51.62%. These findings underscore Demeter's ability to closely match the optimal recovery times of the static configuration, unlike the larger variances seen with other methods. Additionally, Demeter was the method which

Table 3: Recovery times & number of reconfigurations (Δ).

(a) YSB experiment results.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	#17	#18	#19	#20	#21	#22	#23	Δ
Workload	30K	50K	80K	55K	25K	45K	30K	50K	35K	50K	40K	65K	45K	33K	25K	35K	45K	45K	50K	60K	65K	25K	27K	–
Static	122s	95s	93s	94s	95s	92s	96s	96s	97s	96s	95s	95s	99s	94s	94s	95s	95s	96s	95s	94s	95s	96s	95s	–
Demeter	123s	97s	97s	98s	90s	95s	90s	89s	NR	120s	95s	96s	100s	96s	NR	90s	87s	NR	97s	96s	95s	96s	140s	33
Reactive	201s	200s	95s	145s	182s	255s	146s	252s	95s	165s	190s	NR	143s	136s	193s	164s	185s	190s	266s	135s	152s	NR	205s	87
DS2	187s	95s	125s	204s	330s	122s	6m+	125s	6m+	126s	95s	6m+	125s	129s	NR	95s	91s	120s	6m+	NR	155s	95s	NR	77

(b) TSW experiment results.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	#17	#18	#19	#20	#21	#22	#23	Δ
Workload	30K	50K	80K	55K	25K	45K	30K	50K	35K	50K	40K	65K	45K	33K	25K	35K	45K	45K	50K	60K	65K	25K	27K	–
Static	95s	95s	95s	94s	96s	155s	98s	96s	100s	97s	96s	152s	127s	126s	95s	106s	96s	96s	98s	93s	97s	166s	95s	–
Demeter	96s	95s	90s	90s	120s	127s	150s	NR	90s	120s	100s	90s	116s	160s	95s	NR	89s	95s	123s	120s	120s	160s	120s	30
Reactive	158s	210s	171s	175s	205s	200s	199s	170s	178s	95s	146s	198s	225s	150s	185s	220s	170s	192s	115s	170s	175s	146s	128s	49
DS2	125s	95s	95s	94s	125s	275s	6m+	226s	95s	6m+	127s	124s	125s	157s	NR	135s	NR	6m+	95s	126s	95s	6m+	97s	73

initiated the least number of reconfigurations (Δ). 'No Result' (NR) entries in our table reflect the dynamic nature of jobs; reconfigurations for exactly-once processing sometimes overlap with failure injections, leading to unsuccessful recovery attempts.

3.5.3 Resource Usage Results: Figures 6(c) and 6(d) display the cumulative CPU and memory usage for both experiments, normalized against the maximum resource usage benchmarked at 100%. For Demeter, both the target job's resource usage and the cost of profiling are included. As expected, the static configuration consistently showed the highest consumption. In the YSB experiment, Flink Reactive achieved a 40% reduction in CPU and memory usage relative to the static configuration. Demeter demonstrated a 19% reduction in CPU usage and 37% in memory, while DS2 recorded a 14% improvement. Including Demeter's profiling costs, the net resource savings were 7% in CPU and 32% in memory usage compared to the static configuration. Flink reactive was the best performer for this experiment. In the TSW experiment, Demeter achieved a 20% reduction in CPU usage and 43% in memory compared to the static configuration. Flink Reactive showed a 17% reduction, and DS2 had a 10% improvement. After factoring in profiling costs, Demeter's overall efficiency was a 3% saving in CPU and 35% in memory. In this case, Demeter had the most efficient memory usage, while Flink Reactive was the most efficient in terms of CPU usage.

3.6 Experimental Discussion

In both experiments, Demeter maintained latencies and recovery times close to those of the static configuration, while also enhancing resource efficiency. Flink Reactive showed good performance in terms of latencies in the YSB experiment; however, it was not able to achieve comparable results in the TSW experiment. Moreover, both Flink Reactive and DS2 had variable recovery times, often substantially longer than the static configuration, indicating inconsistent performance. For methods aimed at enhancing resource efficiency to be considered effective, maintaining a high Quality of Service is fundamental. If the service is not reliably available, the resulting benefits of resource savings are irrelevant. This is further evidenced by Demeter initiating significantly fewer reconfigurations across both experiments, thereby minimizing disruptions to the service. In the experiments, the only two methods that consistently met this requirement were the static configuration and Demeter.

While the initial analysis indicates Demeter's modest CPU usage improvement of 7% and 3%, and a substantial decrease in memory usage by 32% and 35% compared to the static configuration, continuing the experiment could reveal further enhancements. Extending execution beyond 18 hours would enable more profiling and the selection of increasingly optimized configurations, thus reducing the time spent at C_{max}. Assuming similar workload rates, we anticipate a further decrease in resource usage over time. This trend is illustrated in Figures 6(e) and 6(f), which show CPU and memory usage over time, concentrating on the workload segments with the highest data points in each experiment. We included regression lines to highlight the resource usage trends. The data shows a general decrease in resource utilization, with CPU usage potentially dropping by 20% to 30%, and memory usage by 50% to 70%.

Similarly, if the experiment were to be extended, the annealing factor applied to profiling would lead to a reduction in profiling executions as knowledge is accumulated. Consequently, the cost of profiling is expected to progressively decrease, further enhancing Demeter's resource efficiency. Considering this, it is possible that Demeter would not only maintain its high performance but also improve upon the resource usage results from the other methods.

4 RELATED WORK

In this section, we explore the work related to our own, examining the domain of automatic configuration optimization.

4.1 Batch Processing Optimization

In batch processing, configuring resources to meet runtime targets is a key challenge. In exploring methods similar to our own, we examined Bayesian Optimization in CherryPick [4], which efficiently identifies near-optimal configurations for batch jobs using adequately precise performance models. Karasu [28] likewise makes use of this, employing MOBO and RGPE modeling for resource optimization and introducing a strategy for profiling with shared user data to overcome the cold start problem. Other methods focus on modeling batch job scaleout behaviors, using historical data for resource allocation—similar to our data gathering through profiling runs. Ernest [34] predicts cloud resource needs by running jobs with subsets of inputs and different configurations. Bell [32] uses existing workload data from recurring jobs for its predictions, eliminating isolated training runs. In our previous work, we presented Enel [29] which adopts a context-aware, graph-based approach for more detailed insights within batch jobs' synchronization barriers by incorporating runtime metrics. Taking inspiration from these methods, Demeter applies this to the stream processing domain.

4.2 Stream Processing Optimization

In stream processing, optimization approaches primarily focus on elastic scaling of resources to adapt to runtime conditions. A number of these focus on threshold monitoring and re-configuring after a predefined threshold has been violated. Gedik et al. in [12] introduce a control algorithm for IBM Infosphere Streams [6] that reacts to congestion, while Dhalion [11] applies policy-driven strategies for Heron [23]. TWRES [17] uses TSF for predicting future workloads and adjusts resources based on a latency constraint. Prompt [3] is a data partitioning scheme specifically for micro-batch DSPs, focused on maintaining latency guarantees through a threshold-based elasticity technique that dynamically adjusts execution parallelism. Apache Flink's Reactive Mode [2] automates the scaling process, dynamically adjusting resources to workload variations. Elastic Spark Streaming [1] also adopts a similar automated scaling approach, enhancing resource management in Spark environments.

However, the reliability of threshold-based autoscalers can be affected by the transient nature of shared computing environments and DSP job behaviors, leading to unnecessary adjustments. As a result, stream processing research has more recently adopted data-driven approaches using performance modeling for scaling decisions. Petrov et al.[27] detail a model that bases scaling actions on latency measurements, and DS2[21] uses historical and real-time data for workload forecasting to dynamically scale streaming dataflows. In our previous work with Phoebe [13], initial profiling was conducted to establish models that map scaleout and workload rates to latency and recovery times. TSF was then employed to predict future workloads, allowing for dynamic rescaling of resources aimed at maintaining stable latencies and achieving optimal recovery times. However, these data-driven methods can be limited by the availability of historical data, challenging their accuracy in dynamic environments. In contrast, Demeter concurrently addresses multiple configuration parameters and incorporates an efficient profiling method for data gathering, enabling more comprehensive optimization in stream processing.

In the context of Bayesian Optimization, a number of methods have been proposed for the optimization of configuration parameters in DSP systems. Fischer et al., in [10], suggest a technique for tuning multiple parameters that, while effective for Apache Storm—an older DSP system—is not widely applicable, hence not offering a generalizable solution. In [19], Jamshidi et al. present a method for tuning configuration parameters to reduce latency and maintain throughput. Their results are positive, but the paper does not discuss workload rates, leading to an implicit assumption that their optimization is designed for static workloads. ContTune [24] focuses on a single configuration parameter which employs Conservative Bayesian Optimization to fine-tune DSP job parallelism while ensuring SLA adherence. However, these methods overlook fault tolerance and SLA-specific recovery time considerations.

4.3 Checkpoint Optimization

Another area of related work is fault tolerance, particularly the optimization of the checkpoint and rollback recovery mechanism in DSP systems. Here the system's state is periodically saved, allowing for restarts from the latest checkpoint after a failure. Our approach, which involves optimizing the checkpoint interval to enhance system performance, is related to other methods that adjust this parameter. In high-performance computing, some methods determine the mean time to failure of cluster nodes and modify the checkpoint interval to reduce downtime from failures [8, 9, 36]. In the specific context of stream processing, [20] explore the effects of system failures and configurations on recovery, aiding in the development of more efficient checkpoint scheduling strategies. Our work differs from theirs in that while they do not seek optimal configurations for DSP jobs at runtime, Demeter actively does. In our previous work, we also investigated parameter auto-tuning of DSP jobs to improve end-to-end latencies and recovery time, yet focused on optimizing checkpoint intervals while assuming scaleouts to be static [14, 15].

5 CONCLUSION

In this paper, we presented Demeter, a method designed to enhance the resource efficiency of DSP jobs running in dynamically changing environments. By utilizing TSF to predict future workloads and MOBO to model runtime behaviors, Demeter effectively decides itself when to initiate short-lived parallel profiling runs and when to proceed with optimization. This approach guides the adjustment of multiple configuration parameters, providing near-optimal performance as workload rates vary. Our evaluation results show that Demeter not only matches the high performance of over-provisioned static configurations in terms of average end-to-end latencies and recovery times but also significantly improves resource efficiency. Specifically, Demeter showed improvements of 7% and 3% in CPU usage, and more substantially, 32% and 35% in memory usage. As the experiments show, with further execution, resource efficiency would increase, leading to additional cost savings, which is particularly important in cloud environments where

memory resources are more expensive than CPU resources. Furthermore, Demeter is designed for scenarios requiring strict data consistency and the need for exactly-once processing guarantees.

ACKNOWLEDGMENT

For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising from this submission.

REFERENCES

[1] 2017. Auto-Scaling Systems with Elastic Spark Streaming. https://www.youtube.com/watch?v=N10k7tRsTUA&ab_channel=SparkSummit. Accessed: March 2024.
[2] 2021. Apache Flink - Reactive Mode. https://flink.apache.org/2021/05/06/reactive-mode. Accessed: March 2024.
[3] Ahmed Sayed Abdelhamid, Ahmed R. Mahmood, Anas Daghistani, and Walid G. Aref. 2020. Prompt: Dynamic Data-Partitioning for Distributed Micro-batch Stream Processing Systems. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).
[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*.
[5] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2020. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems 33*.
[6] Alain Biem, Eric P. Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris N. Koutsopoulos, and Carlos Moran. 2010. IBM infosphere streams for scalable, real-time, intelligent transportation services. *SIGMOD* (2010).
[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38, 4 (2015), 28–38.
[8] John Daly. 2003. A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps. In *ICCS (LNCS, Vol. 2660)*, Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Jack J. Dongarra, Albert Y. Zomaya, and Yuri E. Gorbachev (Eds.). Springer.
[9] John T. Daly. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.* 22, 3 (2006).
[10] Lorenz Fischer, Shen Gao, and Abraham Bernstein. 2015. Machines Tuning Machines: Configuring Distributed Stream Processors with Bayesian Optimization. *2015 IEEE International Conference on Cluster Computing* (2015), 22–31.
[11] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. *VLDB* (2017).
[12] B. Gedik, S. Schneider, M. Hirzel, and K. Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE TPDS* (2014).
[13] Morgan K. Geldenhuys, Dominik Scheinert, Odej Kao, and Lauritz Thamsen. 2022. Phoebe: QoS-Aware Distributed Stream Processing through Anticipating Dynamic Workloads. *2022 IEEE International Conference on Web Services (ICWS)* (2022), 198–207.
[14] Morgan K. Geldenhuys, Lauritz Thamsen, Kain Kordian Gontarska, Felix Lorenz, and Odej Kao. 2019. Effectively Testing System Configurations of Critical IoT Analytics Pipelines. *2019 IEEE International Conference on Big Data (Big Data)* (2019), 4157–4162.
[15] Morgan K. Geldenhuys, Lauritz Thamsen, and Odej Kao. 2020. Chiron: Optimizing Fault Tolerance in QoS-aware Distributed Stream Processing Jobs. *2020 IEEE International Conference on Big Data (Big Data)* (2020), 434–440.
[16] Kordian Gontarska, Morgan K. Geldenhuys, Dominik Scheinert, Philipp Wiesner, Andreas Polze, and Lauritz Thamsen. 2021. Evaluation of Load Prediction Techniques for Distributed Stream Processing. *2021 IEEE International Conference on Cloud Engineering (IC2E)* (2021), 91–98.
[17] Zhigang Hu, Hui Kang, and Meiguang Zheng. 2019. Stream Data Load Prediction for Resource Scaling Using Online Support Vector Regression. *Algorithms* (2019).
[18] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan. 2019. A Survey of Distributed Data Stream Processing Frameworks. *IEEE Access* (2019).
[19] Pooyan Jamshidi and Giuliano Casale. 2016. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2016), 39–48.
[20] Sachini Jayasekara, Aaron Harwood, and Shanika Karunasekera. 2020. A utilization model for optimization of checkpoint intervals in distributed stream processing systems. *Future Gener. Comput. Syst.* 110 (2020), 68–79.
[21] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava C. Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *USENIX Symposium on Operating Systems Design and Implementation*.
[22] Jay Kreps. 2011. Kafka: a Distributed Messaging System for Log Processing.
[23] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthikeyan Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. *ACM SIGMOD* (2015).
[24] Jinqing Lian, Xinyi Zhang, Yingxia Shao, Zenglin Pu, Qingfeng Xiang, Yawen Li, and Bin Cui. 2023. ContTune: Continuous Tuning by Conservative Bayesian Optimization for Distributed Stream Data Processing Systems. *Proc. VLDB Endow.* 16 (2023), 4282–4295.
[25] Hamid Nasiri, Saeedeh Nasehi, and Maziar Goudarzi. 2018. A Survey of Distributed Stream Processing Systems for Smart City Data Analytics. In *SCIOT '18*.
[26] Hamid Nasiri, Saeed Nasehi, and Maziar Goudarzi. 2019. Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities. *Big Data* (2019).
[27] Max Petrov, Nikolay Butakov, Denis Nasonov, and Mikhail Melnik. 2018. Adaptive performance model for dynamic scaling Apache Spark Streaming. *YSC* (2018).
[28] Dominik Scheinert, Philipp Wiesner, Thorsten Wittkopp, Lauritz Thamsen, Jonathan Will, and Odej Kao. 2023. Karasu: A Collaborative Approach to Efficient Cluster Configuration for Big Data Analytics. In *IPCCC*. IEEE.
[29] Dominik Scheinert, Houkun Zhu, Lauritz Thamsen, Morgan K. Geldenhuys, Jonathan Will, Alexander Acker, and Odej Kao. 2021. Enel: Context-Aware Dynamic Scaling of Distributed Dataflow Jobs using Graph Propagation. In *IPCCC*. IEEE.
[30] Florian Schmidt, Florian Suri-Payer, Anton Gulenko, Marcel Wallschläger, Alexander Acker, and Odej Kao. 2018. Unsupervised Anomaly Event Detection for VNF Service Monitoring Using Multivariate Online Arima. In *CloudCom*. IEEE Computer Society.
[31] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System, Mohammed G. Khatib, Xubin He, and Michael Factor (Eds.). IEEE Computer Society.
[32] Lauritz Thamsen, Ilya Verbitskiy, Florian Schmidt, Thomas Renner, and Odej Kao. 2016. Selecting resources for distributed dataflow systems according to runtime targets. *IPCCC* (2016).
[33] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. 2014. Storm@twitter. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014).
[34] Shivaram Venkataraman, Zongheng Yang, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*.
[35] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM.
[36] J. W. Young. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM* 17 (1974), 530–531.
[37] Matei A. Zaharia, Reynold Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph E. Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark. *Commun. ACM* 59 (2016), 56 – 65.
[38] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD*. ACM.

BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era

Zebin Ren
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Krijn Doekemeijer
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Nick Tehrany*
BlueOne Business Software LLC
Beverly Hills, California, USA

Animesh Trivedi
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

ABSTRACT

Flash SSDs have become the de-facto choice to deliver high I/O performance to modern data-intensive workloads. These workloads are often deployed in the cloud, where multiple tenants share access to flash-based SSDs. Cloud providers use various techniques, including I/O schedulers available in the Linux kernel, such as *BFQ*, *Multiqueue-Deadline* (MQ-Deadline), and Kyber, to ensure certain performance qualities (i.e., service-level agreements, SLAs). Though designed for fast NVMe SSDs, there has not been a systematic study of these schedulers for modern, high-performance SSDs with their unique challenges. In this paper. we systematically characterize the performance, overheads, and scalability properties of Linux storage schedulers on NVMe SSDs with millions of I/O operations/s. We report 23 observations and 5 key findings that indicate that (i) CPU performance is the primary bottleneck with the Linux storage stack with high-performance NVMe SSDs; (ii) Linux I/O schedulers can introduce 63.4% performance overheads with NVMe SSDs; (iii) Kyber and BFQ can deliver 99.3% lower P99 latency than None or MQ-Deadline schedulers in the presence of multiple interfering workloads. We open-source the scripts and datasets of this work at: https://zenodo.org/records/10599514.

CCS CONCEPTS

• **General and reference** → *Empirical studies*; • **Software and its engineering** → **Secondary storage**.

KEYWORDS

Linux storage scheduler, Quality of service, Measurements, NVMe

ACM Reference Format:
Zebin Ren, Krijn Doekemeijer, Nick Tehrany, and Animesh Trivedi. 2024. BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645053

*Work done while the author was at Vrije Universiteit Amsterdam.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645053

1 INTRODUCTION

Modern flash-based NVMe solid-state drives (SSDs) are able to deliver millions of I/O operations per second (IOPS) and single-digit microsecond-level latency [7, 11]. These SSDs are widely used in multi-tenant cloud environments where their performance, bandwidth, and latency are shared among multiple tenants or workloads [3, 5, 13, 28, 32, 38, 43, 44]. In multi-tenant cloud environments, there is commonly a scheduler of I/O requests designed to deliver fairness with quality-of-service guarantees, also known as service level agreements (SLA), for the cloud services [38, 44].

Designing a fair, high-performance, low-overhead I/O scheduler has been a topic of extensive research over the past decade, with studies focusing on providing proportional performance sharing with request scheduling [1, 26, 40, 46, 52, 56, 57], low latency guarantees [27, 39, 42], and SSD-supported features acceleration [29, 31, 54]. Despite these studies, the emergence of high-performance NVMe SSDs has created multiple unique challenges for storage schedulers (or I/O schedulers) that have not been systematically studied or characterized. These unique challenges come from three distinct dimensions that we cover in this study: (i) performance overheads from the complex storage software stack on fast SSDs; (ii) scalability challenges in the presence of highly concurrent I/O operations on NVMe SSDs; and (iii) interference management among competing I/O requests at high CPU loads.

Firstly, *state-of-the-practice* I/O schedulers available in Linux (BFQ, MQ-Deadline, and Kyber) are not yet studied at the performance scale of millions of IOPS which modern NVMe SSDs can deliver. At this scale, small overheads from the Linux storage stack become a performance bottleneck [51, 55]. These overheads come from operational complexity (dispatching, merging, sorting, and staging I/O requests) that determine the maximum performance a scheduler can deliver. Various enterprise software recommends using no scheduler (also known in Linux as None) on high-performance SSDs to limit or eliminate these overheads [49, 50]. However, in this work, we demonstrate that this decision can sacrifice the quality of service and fairness among workloads. Secondly, modern server machines are highly parallel with multiple CPU cores and SSDs with parallel I/O queues. Here, overheads related to locking, synchronization, and queue management become the key performance bottlenecks. Thirdly, the widely-adopted flash-based SSDs have different read and write performance characteristics, and the reads and writes interfere with each other [20, 25]. This interference creates challenges for I/O schedulers to fairly schedule mixed read-write workloads. Furthermore, a shared environment

brings new challenges to I/O schedulers [30, 32, 45] since workloads have diverse I/O requirements. For example, throughput-bound workloads like batch analytics (e.g., Spark) require high IOPS, and real-time workloads like interactive queries [16] expect low, predictable latencies. I/O schedulers need to provide SLA guarantees to satisfy these requirements across all the tenants, possibly simultaneously [37, 41, 47, 48]. There have been attempts to design SSD-aware schedulers [27, 46, 52, 56] (including Linux Kyber [8]). However, the performance of these *state-of-the-practice* schedulers (available in the Linux kernel) has not been studied or quantified on high-performance NVMe SSDs with millions of IOPS. To summarize, the high performance, scale, and interference properties of modern NVMe SSDs motivate us to systematically study and characterize the performance of Linux I/O schedulers.

In this study, we aim to measure, quantify, and analyze overheads from three widely available Linux storage I/O schedulers, BFQ [1], Kyber [8], and MQ-Deadline [9] on flash-based NVMe SSDs (§2). We start our investigation by studying the performance (expressed as IOPS or latencies) using fio, a widely used microbenchmark with the high-performance `io_uring` storage engine [19] on our 8 NVMe SSDs setup with a peak performance of 5.9 million 4KiB IOPS (see Table 1). We specifically motivate and investigate the following research questions (**RQ**) around the performance and overheads of Linux I/O schedulers with the key findings (**KF**):

(RQ1) What is the CPU overhead for modern fast NVMe SSD devices? Can the Linux storage stack saturate multiple NVMe SSDs? This research question is important to establish the baseline performance of the SSDs around which the rest of the scheduler analysis is done. The key motivation for this question is to establish under which configurations the CPU cores or SSDs become a bottleneck, and how they influence the performance of I/O schedulers. **KF-1:** *for high-performance NVMe SSDs, the CPU is the key bottleneck (even with the None scheduler), thus making the scheduling "efficiency" of I/O schedulers a key factor in the performance delivered to the workloads* (§3).

(RQ2) What are the scheduler overheads, and how do these overheads scale with the I/O and device concurrency? NVMe devices have deep I/O queues with multiple parallel queues. Hence, typically it takes multiple requests to saturate a single device. Here we study the overhead of I/O scheduling and how it scales with concurrent requests when requests come from a single workload (intra-process) and multiple workloads (inter-process). **KF-2:** *once the CPU becomes the bottleneck, the Linux I/O schedulers can induce 63.4% overheads in throughout and 50× increase in the P99 latencies over the None scheduler. We also report that the presence of multiple SSDs helps to reduce overheads associated with the device-specific locking and synchronization overheads* (§4).

(RQ3) How can I/O schedulers help to control interference in the presence of competing workloads, specifically latency-vs-bandwidth and read-vs-write? These kinds of mixed workloads are quite common in the cloud/enterprise settings. Hence, it is important to study what are the scheduler overheads in this scenario. The extent of the interference typically governs what kind of SLAs (99 percentile, worst-case performance scenarios) storage service providers can offer to their tenants. **KF-3** and **KF-4:** *Kyber and BFQ can provide good bounded performance in the presence of interference when the CPU is not the bottleneck, however, BFQ suffers from performance scalability overheads. Hence, we conclude that overall Kyber is the best fit Linux I/O schedulers for SSDs* (§5).

Table 1: Details of the benchmarking environment.

Component	*Configuration*
CPU	Single socket Intel(R) Xeon(R) Silver 4210R CPU 10 cores @ 2.40GHz, Hyper-threading disabled, Turbo disabled.
Memory	256GB, DDR4.
Storage	8× Samsung 980 PRO 1TB, Average latency (r/w): ~68/15 μs, peak random read IOPS: 1M@4KiB/device at the queue depth 32.
Software	Ubuntu 20.04 with Linux kernel `v6.3.8` (released April'23), fio v3.35, SPDK 22.09.

(RQ4) How do a scheduler configuration parameters affect the schedulers' behavior on competing workloads? The Linux I/O schedulers provide tunable parameters, which affect the schedulers' behavior. Based on our empirical findings, we further present a detailed analysis of the Kyber scheduler which is specifically designed for modern SSDs. It has two unique configuration parameters: read and write target latencies. We perform a configuration space exploration for Kyber. **KF-5:** *Kyber can be configured to prioritize latency or total throughput by tuning its read and write target latencies, but not both* (§6).

Our key contributions in this work include:

- To the best of our knowledge, this is the first-of-its-kind systematic study about overhead quantification and characterization of *state-of-the-practice* I/O schedulers with modern NVMe SSDs, exploring their performance, scalability, and interference patterns, resulting in 23 observations and 5 key findings.
- We explore the configuration space of Kyber, an SSD-optimized scheduler. We report that Kyber has the least amount of CPU overhead, and it can provide bounded performance in the presence of read/write interference.
- To facilitate reproduction, we open-source the design and implementation of our code and datasets at https://github.com/stonet-research/icpe24_io_scheduler_study_artifact. Permanent link: https://zenodo.org/records/10599514.

2 BENCHMARKING ENVIRONMENT

In this section, we present details about the benchmarking environment, workloads used, and selected Linux storage schedulers.

2.1 Hardware and Software

We use fio [4] as the workload generator with the `io_uring` I/O engine [14, 19, 51]. Our setup is able to deliver a peak random read performance of 3.4 Million IOPS with the Linux storage stack under Linux v6.3.8 (5.9 Million IOPS with SPDK that by-passes the kernel), and an average read latency of 68 μs (4KiB, with a queue depth of 1, QD=1); hence, creating a unique opportunity to study I/O schedulers in this high-performance I/O setup. We use three metrics to evaluate the performance of the Linux I/O schedulers: throughput, latency, and CPU usage. We measure the

throughput as I/O operations completed per second (IOPS). Latency measurements focus on the tail latency where we report 99 percentile latencies (P99) with a complete CDF distribution. The average CPU usage is measured using fio, which reports "the CPU time used by the process/run time". CPU usage = 1 means that the process uses a whole CPU core. fio gets the resource usage with the Linux `getrusage` system call [6]. We precondition the flash SSDs according to [18], by sequentially writing the entire device, then randomly writing 4KiB blocks with a total of 2TiB data. Each device can deliver 1 MIOPS@4KiB random read according to the specification. However, we only get ~770 KIOPS with 4KiB random read workload after the preconditioning.

2.2 Workload Patterns and Methodology

We focus on two kinds of applications in this work, latency-sensitive and throughput-bound. We use L-app to represent latency-sensitive applications such as database queries and T-app to represent throughput-bound applications such as batch processing jobs like Map-Reduce. We use three kinds of synthesized workloads to simulate these two kinds of applications:

(1) **L-app** (latency-sensitive application) generates requests of 4KiB block sizes with an I/O depth of 1 (only one outstanding request at a time, we also refer to the I/O depth as queue depth or QD in the following sections).
(2) **T-4KiB-app** (small I/O, throughput-bound application with 4KiB block size) generates requests of 4KiB block sizes with an I/O depth of 128. With 4KiB block size, a single core can not saturate the evaluated Samsung SSD on our setup. We use T-4KiB-apps to show the effect of I/O schedulers on I/O performance when the CPU is the bottleneck.
(3) **T-64KiB-app** (large I/O, throughput-bound application with 64KiB block size) generates requests of 64KiB block sizes with an I/O depth of 128. With the 64KiB block size, the evaluated Samsung SSD can be saturated with a single CPU core in our setup for all I/O schedulers. We use T-64KiB-apps to show the effect of I/O schedulers when the SSD is the bottleneck.

Experiments with read-only workloads run for 150 seconds (2 minutes + 30 seconds warm-up time) since the read performance of flash-based SSDs is stable. For applications that issue writes, we run each experiment for 12 minutes (6 minutes + 6 minutes warm-up time) with 5 repetitions to get stable results, we report both the average value and standard deviation.

2.3 I/O Schedulers Under Study

The Linux kernel has four multi-queue enabled I/O schedulers: None, BFQ [1], Kyber [8], and MQ-Deadline [9]. For this study, we use the default configuration of each scheduler as they are the most likely used configurations for real workloads. For Kyber we further explore the configuration space to synthesize guidelines (§6).

None is the default I/O scheduler for NVMe devices that is recommended often to reduce the scheduling overheads with fast NVMe storage devices [49, 50]. Technically, None is not an I/O scheduler since it dispatches I/O requests to the NVMe driver immediately when it gets a new request without reordering the requests. Due to its simplicity, None has the lowest overheads among all I/O schedulers. Hence, we select it as the baseline scheduler.

MQ-Deadline is the multi-queue version of the Deadline scheduler [2]. The main goal of MQ-Deadline is to guarantee the start service time for a request. MQ-Deadline maintains two read-write queue pairs, a sorted queue pair and a FIFO queue pair. MQ-Deadline issues I/O requests in increasing sector orders (from the sorted queue) unless there is a request that violates the service deadline (from the FIFO queue). When the service deadline is violated, MQ-Deadline issues the request from the FIFO queue.

BFQ (Budget Fair Queuing) is a proportional-sharing I/O scheduler that is designed to provide fair bandwidth sharing and low latency for latency-sensitive applications. BFQ associates each process with an internal request queue and a budget according to each process' weight in the number of sectors. BFQ uses worst-case fair weighted fair queuing+ (WF2Q+) [15] to select the next queue to service and exclusive device access is given to the selected queue until its I/O budget is used up or a timeout happens. To provide low latency for real-time applications such as video players, BFQ uses heuristics to detect applications that are sensitive to latency and gives them higher priority. BFQ is the most complex Linux kernel I/O scheduler (BFQ is ~10,000 LOC, against ~1,000 LOC for Kyber and MQ-Deadline), and is believed to have the highest overhead among the Linux I/O schedulers [1, 12].

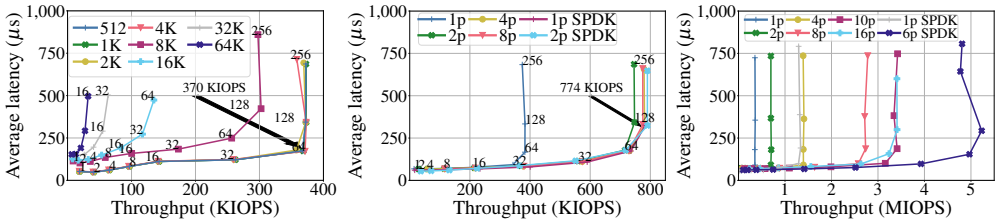
Kyber is designed for fast multi-queue devices to deliver low latency for reads. It is based on a heuristic that a process that issues a read I/O request usually waits for the request to finish and the data to be available (synchronous completion). In contrast, a process that issues a write I/O request usually can continue executing before the writes are finished (asynchronous completion). Thus, Kyber prioritizes reads over writes, but not to the extent where writes are starved. To achieve high responsiveness, Kyber prevents requests from building up on the device side with tokens. A detailed description of how Kyber works is presented in §6.

3 BOTTLENECK ANALYSIS: CPU OR NVME

We start our analysis by characterizing the performance of Samsung NVMe devices with the None scheduler to explore how CPU or NVMe devices become a bottleneck. Figure 1 shows our results. In the graphs, each line (same block size, increasing QD) has a hook shape: as QD increases, the throughput grows fast (x-axis), while the latency remains stable. At a certain turning point, the latency starts to grow fast with throughput remaining the same (because of queuing delays). This turning point is called the **saturation point** or **knee point**. At the saturation point, either the CPU or the SSD becomes the bottleneck.

3.1 What throughput and latency can a single SSD and a single CPU core deliver?

To answer this question, we configure a single SSD with a single CPU core and issue a random read workload. We measure the throughput and latency as we increase the number of outstanding requests (i.e., queue depth). Figure 1a shows the throughput as IOPS (x-axis) with the average latency (y-axis, the lower the better) for multiple queue depths (points on the lines). In this setting, we report that the performance of request sizes smaller than or equal to 4KiB are similar in nature, all saturating at a queue depth of 64 with 370 KIOPS as the peak performance. After this point,



(a) Vary request sizes, 1 SSD. **(b) Vary # processes with 1 SSD.** **(c) Vary # processes with 8 SSDs.**

Figure 1: Throughput and latency of a Samsung 980 PRO 1TB with varying queue depth and number of processes.

increasing the queue depth leads to further queuing delays, hence an increase in the latency. *At this point, the CPU is the primary bottleneck as it is 100% utilized, yet the SSD device itself is not saturated* (**Observation-1, O-1**). For request sizes greater than 4KiB, the throughput decreases proportionally, and the CPU gets saturated at shallow queue depths, hence, the latencies start to increase.

3.2 How does the performance scale as we increase the number of CPU cores or SSDs?

In the subsequent experiments, we increase the number of CPU cores with one SSD (Figure 1b) and then the number of SSDs (Figure 1c). We observe in Figure 1b that with 4 cores (denoted as 4p, or processes), a single SSD is saturated with the peak performance of 774 KIOPS. In Figure 1c, we continue to scale the number of CPU cores with all 8 SSDs (the max possible). The figure shows that the peak performance shifts from 774 KIOPS (from Figure 1b) to 3.4 MIOPS for 10 CPU cores (and 16 as well). In this configuration, all 10 CPU cores are 100% utilized and this is the peak performance the Linux kernel can deliver with the `io_uring` engine (bounded by the CPU performance). We also plot the SPDK performance, which is a high-performance, kernel-bypass storage stack [10]. With SPDK, it takes 6 CPU cores ("6p SPDK" line) to deliver 5.2 MIOPS, saturating the SSDs (within 85% of the peak possible 6.2 MIOPS (8×774 KIOPS) that our hardware should deliver). *This demonstrates that SPDK (1.3 MIOPS/core) is still the state-of-the-art storage stack, which is at least 3.6× more efficient (single core peak IOPS performance with 8 SSDs, "*`1p`*" and "*`1p SPDK`*" lines in Figure 1c) than the Linux storage stack (370 KIOPS/core)* (**O-2**). SPDK can deliver higher throughput than the Linux storage stack when the CPU is the bottleneck because SPDK is more CPU efficient than the Linux storage stack. Previous study shows that SPDK needs fewer CPU instructions to process each I/O request than the Linux storage stack [51].

3.3 Summary

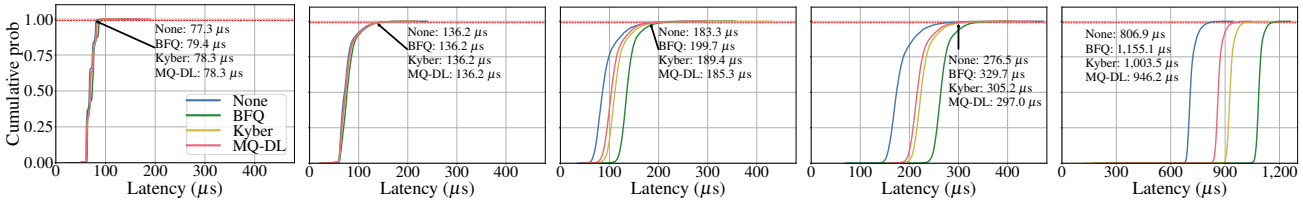
What is the key resource bottleneck for the L-app, T-4KiB-app, and T-64KiB-app? The key finding (**Key Finding, KF1**) here is that as the NVMe device speeds are improving, the CPU becomes the primary bottleneck. Modern fast NVMe storage devices like Samsung 980 PRO 1TiB, can require more than a single CPU core to saturate the performance of the SSD. On our setup, L-apps (QD=1), and T-4KiB-apps (QD=128) are bounded by the CPU performance. We report that the T-4KiB-app needs at least 4 T-4KiB-apps on 4 CPU cores to saturate a single SSD. The T-64KiB-app can saturate a single SSD with only one CPU core (not shown). *Hence, the age-old mantra of "CPU is fast, I/O devices are slow" does not hold anymore for modern fast NVMe SSDs* (**O-3**). We answer **RQ1** by identifying when the CPU or the SSD becomes a bottleneck (the inflection or saturation points). With the Linux I/O stack, the CPU can only deliver 51.6% of the peak hardware throughput (3.2 MIOPS out of a possible 6.2 MIOPS) before it becomes the bottleneck. When we introduce an I/O scheduler to deliver a quality-of-service (QoS) in this setting, it also competes for the CPU cycles. The operational complexity of the scheduler determines the raw performance loss that is traded to deliver a quality-of-service (QoS). In the following section, we study the impact of I/O schedulers (§4), and quantify the performance loss and QoS in the presence of competing tenants with interference (§5).

4 I/O SCHEDULER SCALABILITY

In the previous section, we show the baseline performance and overhead between a CPU and NVMe devices without I/O schedulers with multiple concurrent requests and processes. We report that a single process gets 370 KIOPS at the queue depth (QD) of 64, and the Linux storage stack can not fully saturate 8 SSDs even with 16 T-4KiB-apps (100% CPU utilization). In this section, we introduce I/O schedulers and answer the **RQ2** about how the performance and overheads scale with increasing amounts of concurrency with NVMe devices with I/O schedulers. We evaluate the scalability by analyzing workload throughput (IOPS) and tail latency (P99) with a varying number of hardware resources. Specifically, we measure the scalability of both latency-sensitive and throughput-bound workloads across 2 resource axes: number of SSDs and CPU cores. We look at both intra- and inter-process scalability. Intra-process scalability refers to configurations where we increase the concurrency within a single process by increasing the number of concurrently issued outstanding I/O requests. With inter-process scalability, we increase the concurrency by increasing the number of parallel processes while keeping the concurrency within each process fixed. The expectation here is that multiple processes exercise the scheduling, locking, and synchronization overheads within the scheduler.

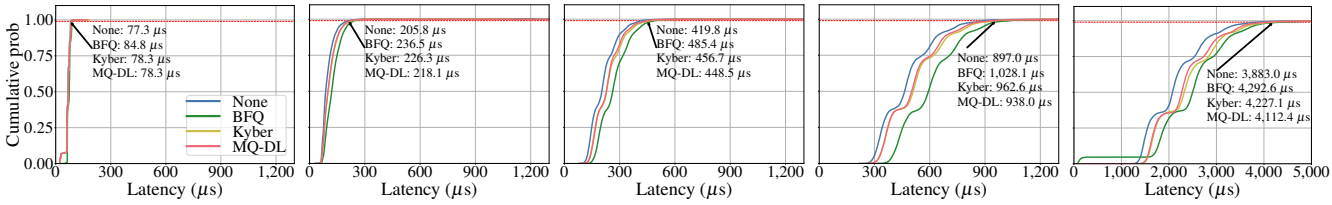
4.1 Scheduler Overheads on Latency

What are the scheduler overheads, and how do they scale with increasing I/O concurrency? In this section, we study the impact of I/O concurrency overheads from schedulers on latency-sensitive applications (L-app). For this, we study the intra- (Figure 2) and inter-process (Figure 3) concurrency overheads. For **intra-process concurrency**, we have a single process (pinned to a core) that



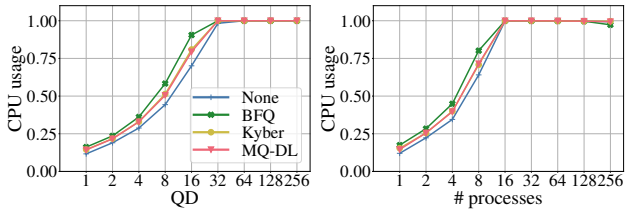
(a) QD = 1 (b) QD = 16 (c) QD = 32 (d) QD = 64 (e) QD = 256

Figure 2: Intra-process scalability latency CDFs with increasing queue depth (QD); Note the different x-axis scale for (e).



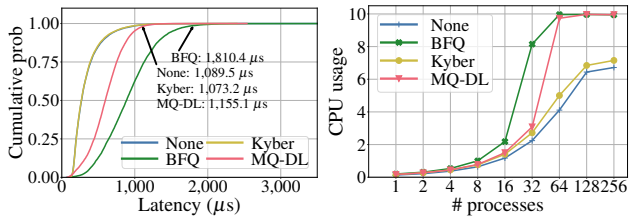
(a) 1 L-app (b) 16 L-apps (c) 32 L-apps (d) 64 L-apps (e) 256 L-apps

Figure 3: Inter-process scalability latency CDFs with increasing number of L-apps; Note the different x-axis scale for (e).



(a) Intra-process (b) Inter-process

Figure 4: CPU usage for intra/inter-process concurrency.



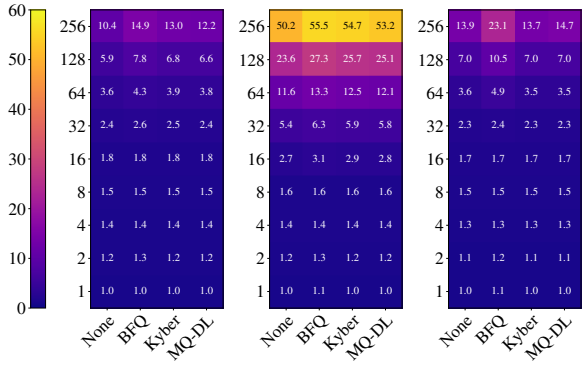
(a) Latency CDF for 256 L-apps (b) CPU usage

Figure 5: L-app inter-process scalability (10 cores, 1 SSD); Note in (b) the y-axis is the CPU usage, which is from [0–10], representing 10 CPU cores.

issues 1, 16, 32, 64, or 256 outstanding requests at a time. For this process, the latency CDF is shown in Figure 2 (annotations are for P99 latency). There are two key observations here. Firstly, for the concurrency of 1 and 16, the latency profile shape of different I/O schedulers looks quite similar. This is due to the fact that as we increase concurrency for a single process, the CPU load increases but still remains under 100%, hence having spare capacity (Figure 4a). We still report that P99 latencies increase up to 73.9% for *all* schedulers (from 78.3 to 136.2 μs). *As long as the CPU is not 100% utilized, all three schedulers have comparable performance* (**O-4**). Secondly, when the CPU load is 100% (at QD=32), the scheduler performance starts to diverge significantly (Figure 2 (c–e)). *Hence, there is a clear number of outstanding requests where BFQ introduces the maximum overheads (43.2% over the None scheduler for QD=256)* (**O-5**). The overall P99 latencies deteriorate from 77.3–79.4 μs (QD=1) to 806.9–1,155.1 μs (QD=256), due to the CPU overheads (as we show previously that a single app can not saturate a single SSD).

In the case of **inter-process concurrency** (multiple concurrent L-apps pinned on a single CPU core with each having QD=1), the latencies deteriorate faster than the intra-process configurations as shown in Figure 3. With the inter-process setup, the Linux kernel has to deal with multiple concurrent processes and associated abstractions and overheads (scheduling, context switching, virtual memory). In a single process case, many of these overheads could be amortized or eliminated. Hence, we also report that the single CPU core where all of these concurrent processes are pinned is saturated with QD=16 via 16 processes as shown in Figure 4b. In comparison, the intra-process saturation point is at QD=32. *Overall as the concurrency increases, the inter-process P99 latencies are approximately 4× higher than their intra-process latencies* (**O-6**). For example, at QD=256 concurrency, the intra-process and inter-process P99 latencies with Kyber are 1,003.5 μs and 4,227.1 μs, respectively.

We further experiment with the **inter-process setup with all 10 cores**, thus no pinning and restricting the performance to a single core. We show the results in Figure 5. *An interesting observation is that with more CPU cycles being available (single core to 10 cores), all the schedulers improve their P99 latencies and bring it closer to their single CPU core, intra-processes performances (Figure 2e), due to the reduction of process scheduling and context switches* (**O-7**). Kyber and None improve much better than BFQ and MQ-Deadline. Their performances are closer to each other with overlapping lines in Figure 5a. BFQ has 1.57× worse P99 latency than its intra-process counterpart. Both BFQ and MQ-Deadline have more slanted shapes, thus having higher P50 and P75 latencies than None and Kyber. This behavior can be explained by their CPU utilization as shown



(a) 1 core, 1 process y-axis: QD **(b) 1 core, QD 1 y-axis: # processes** **(c) 10 cores, QD 1 y-axis: # processes**

Figure 6: Normalized heatmap for P99 tail latency differences (scale from 0–60×) for various concurrency levels (y-axis, 1–256) achieved with (a) QD increase with a single process (intra-process) with 1 CPU core; (b) increasing the number of processes with 1 QD on 1 CPU core (inter-process); and (c) increasing the number of processes with 1 QD on 10 CPU cores (inter-process). The baseline is the None scheduler with 1 concurrency (bottom left box with 1.0 value).

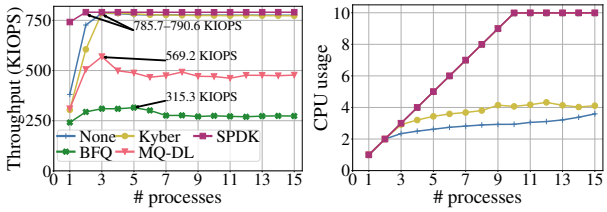
in Figure 5b where BFQ and MQ-Deadline have significantly higher CPU utilization than Kyber and None. With 64 processes, both BFQ and MQ-Deadline are CPU-bounded and thus experience quick degradation of latencies. For 256 processes (for which Figure 5a shows the latencies), None and Kyber use 7 out of 10 CPU cores whereas BFQ and MQ-Deadline use all 10. *These observations summarize that among the three schedulers under study, Kyber is the most light-weight, while BFQ is the most CPU intensive and complex* (**O-8**).

In order to visualize the difference between intra- and inter-process latencies overheads we also plot a normalized heatmap in Figure 6. Here we illustrate P99 latencies degradation (normalized to the 1 concurrency with None scheduler) as we increase the level of concurrency in the system (the y-axis, from 1–256) in three configurations: intra-process (1 CPU core), inter-process (1 CPU core), inter-process (10 CPU cores). The heatmap shows that up to a concurrency of 8, the P99 latencies increase slowly over the None scheduler, and intra-process counterparts (the baseline, showing the normalized value of 1.0). *Between 8–16 concurrency, the CPU becomes the bottleneck, and at this point, the deterioration starts very quickly reaching higher than 50× for inter-process overheads on a single CPU for all schedulers* (**O-9**).

4.2 Scheduler Overheads on Throughput

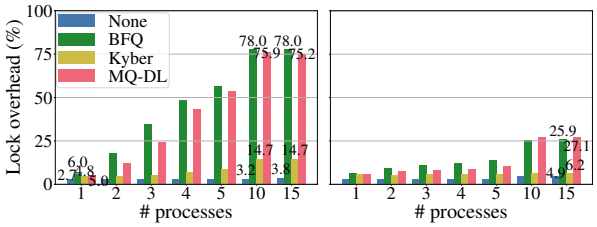
We now bring our attention to the throughput-bounded T-apps where we measure peak throughput (IOPS) and show what scalability and overheads are observed by such workloads with I/O schedulers. In this section, we study four specific questions regarding the scalability properties of the CPU cores and SSDs:

Firstly, what are the scheduler overheads for the throughput-bounded applications, and how do they scale with the number of CPU cores? We start with reporting the scheduler performance



(a) Total throughput **(b) Total CPU usage**

Figure 7: T-app inter-process scalability (10 cores, 1 SSD) with an increasing number of T-4KiB-apps.

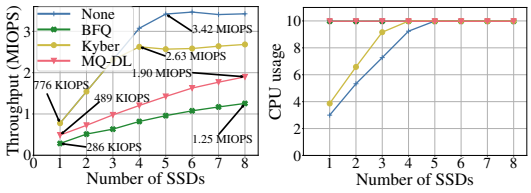


(a) 1 SSD **(b) 8 SSDs**

Figure 8: Combined lock overhead of T-4KiB-apps with an increasing number of concurrent processes.

(in IOPS) as we increase the number of T-4KiB apps. Recall from §3 that a single T-4KiB-app can not saturate a single SSD, and at least 3–4 T-4KiB-apps are needed to saturate a single SSD with the None scheduler. Figure 7 shows scalability results in throughput (IOPS, y-axis, higher is better) for multiple T-4KiB-apps (x-axis) on a single SSD. There are two key observations here. Firstly, the Kyber scheduler performs very closely to None, where both reach the peak single SSD performance of 785.7 KIOPS with 3 processes. For reference, we also have a line for SPDK that can deliver 790.6 KIOPS with a single CPU core, thus demonstrating a wide gap between the in-kernel and kernel-bypass (SPDK) storage stacks. Secondly, both BFQ and MQ-Deadline can not reach the peak device throughput with any number of concurrent T-4KiB-apps. BFQ and MQ-Deadline deliver a peak performance of 315.3 KIOPS (0.40× of the peak 785.7 KIOPS) and 569.2 KIOPS (0.72× of the peak 785.7 KIOPS), respectively. *This loss represents a significant performance degradation, and we conclude that BFQ and MQ-Deadline are unsuitable to be used with modern NVMe SSDs* (**O-10**). The reason for this performance degradation is related to each scheduler's complexity, CPU utilization, and scalability bottlenecks, specifically lock contention.

Figure 7b shows the CPU utilization where we report that despite having CPU available (not all 10 CPU cores are 100% utilized until 10 processes), both BFQ and MQ-Deadline suffer from significant lock contention, thus limiting their performance scalability. To further analyze their CPU utilization behavior, we break down the CPU utilization of these I/O schedulers. We count the total number of CPU cycles (`cpu-cycles` counter with the Linux `perf` framework) and classify lock-related CPU cycles by attributing them to specific lock-related functions: `native_queued_spin_lock_slowpath`, `_raw_spin_lock`, `_raw_spin_lock_irq`, `_raw_spin_lock_irqsave`, and `mutex_lock`. We then plot the fractions of CPU cycles spent in such lock-related functions in Figure 8.



(a) Total throughput (b) Total CPU usage

Figure 9: T-app inter-process scalability (10 cores, 10 concurrent T-4KiB-apps) with an increasing number of SSDs.

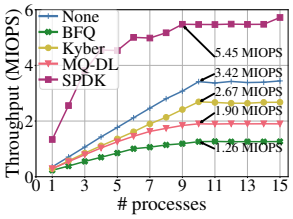


Figure 10: T-app inter-process scalability (10 cores, 8 SSDs) with an increasing number of T-4KiB-apps (x-axis).

From these graphs, we observe that even for one SSD, BFQ and MQ-Deadline spend a significant fraction of their CPU cycles on locking/synchronization functions (as high as 78.0% for 10–15 processes). Increasing the number of available SSDs from 1 to 8 (Figure 8b) reduces the overall CPU cycles spent in lock-related functions for BFQ and MQ-Deadline significantly, by up to 67.1%, suggesting device-specific nature of these locking overheads. Yet even in this case, almost a quarter of CPU cycles (up to 27.1%) are spent on the locking-related functions. In comparison, Kyber only spends 14.7% (single SSD) to 6.2% (8 SSDs) of CPU cycles on lock-related functions. *Based on this analysis, we conclude that locking is the cause of bottleneck for scalability of BFQ and MQ-Deadline and must be urgently tackled* (**O-11**). As of January 2024, Linux kernel developers have also identified this locking issue and are improving the scalability of the BFQ and MQ-Deadline schedulers [35, 36].

Secondly, how does the performance scale in the presence of lock-related overheads with the number of NVMe SSDs? Figure 9 shows our results as we study the performance scalability properties of the schedulers (as IOPS, on the y-axis, higher is better) with 10 T-4KiB-apps (fixed) when we increase the number of SSDs from 1 to 8 (x-axis). In this experiment, the relative overheads and ranking of I/O schedulers remain the same, where Kyber performs the best, followed by MQ-Deadline and lastly BFQ. The performance of BFQ and MQ-Deadline improves sub-linearly from 286 KIOPS and 489 KIOPS with a single SSD to 1.25 MIOPS and 1.90 MIOPS for 8 SSDs, respectively. An interesting observation here is that even with a single SSD, the CPU load for BFQ and MQ-Deadline is 100% for all 10 CPU cores, yet the performance improves as the number of SSDs increases. We speculate (not verified) that this is due to the presence of device-specific locking that exists for a single device, but the presence of multiple devices offers more opportunities for parallelism without being restricted by a single device lock. In the case of Kyber and None, they reach the CPU saturation points with 4 and 5 SSDs, respectively, thus showing IOPS scaling up to those points, delivering a peak performance of 2.63 MIOPS (Kyber) and 3.42 MIOPS (None).

Thirdly, what is the peak performance that various I/O schedulers can provide in Linux? We run the full hardware configuration with 10 CPU cores, 8 SSDs, and vary the number of processes from 1 to 15, and measure the peak IOPS performance of different I/O schedulers. Figure 10 shows our results. There are three specific observations from this experiment. Firstly, there is a clear order in terms of performance among the schedulers (in ascending order): BFQ (1.26 MIOPS), MQ-Deadline (1.90 MIOPS), Kyber (2.67 MIOPS) and None (3.42 MIOPS). Secondly, all of these schedulers reach their peak throughput with 10 processes where all CPU cores become 100% utilized. Lastly, for comparison, we also show the performance of SPDK that can deliver 5.45 MIOPS with 9 CPU cores, demonstrating an almost 2× performance gap to the next best I/O scheduler, Kyber. *From this experiment, we conclude that the CPU and associated heavy Linux software stack and I/O schedulers have become the performance and scalability bottleneck with modern NVMe devices* (**O-12**). Though this final takeaway message is not a surprising find, our key contribution is quantifying the gap between schedulers via controlled experiments where we characterize the performance of the I/O schedulers under various conditions (CPU or SSD becoming the bottleneck). For example, between BFQ and Kyber, there is a performance difference of 2.1×.

Lastly, How do I/O schedulers scale with throughput-bound applications on a single SSD with large request sizes (64KiB)? Unlike the prior evaluation, where the CPU becomes the bottleneck with the T-4KiB-apps, we repeat the same experiments for T-64KiB-apps and report that in this scenario all configurations are SSDs bounded. In these experiments, the performance of all I/O schedulers is almost identical (not shown in any graph). *All the I/O schedulers are able to saturate the SSD with only 2 processes for bandwidth-driven workloads* (**O-13**). The None scheduler reaches a peak throughput of 232 KIOPS with only a single T-64KiB-app process. As the SSD is the bottleneck for the T-64KiB-apps, the CPU utilization for all I/O schedulers remains low, (less than 2 CPU cores). BFQ has a slightly higher CPU utilization of 1.3 CPU cores, 8.3% higher than None (1.2 CPU cores).

Answering RQ1 and RQ2 with KF-2: Based on the analysis in this section, we summarize when the CPU is not the bottleneck, all schedulers perform similarly. As the CPU progressively becomes a bottleneck (i.e., 100% utilization), the Linux I/O schedulers can introduce up to 63.4% performance overheads for the throughput, and more than 50× degradation for P99 latencies over the None scheduler. We also report that the presence of multiple SSDs improves the performance scalability of I/O schedulers due to eschewing the device-specific locking overheads. Based on these findings we recommend using Kyber scheduler (lowest CPU overheads) for NVMe SSDs with modern multicore CPU machines.

5 I/O INTERFERENCE WITH CONCURRENT WORKLOADS

In this section we investigate RQ3: "*how can I/O schedulers help to control interference in the presence of competing workloads, specifically latency-vs-bandwidth, and read-vs-write*". Quality-of-service (QoS) is an essential component of I/O schedulers in multi-tenant

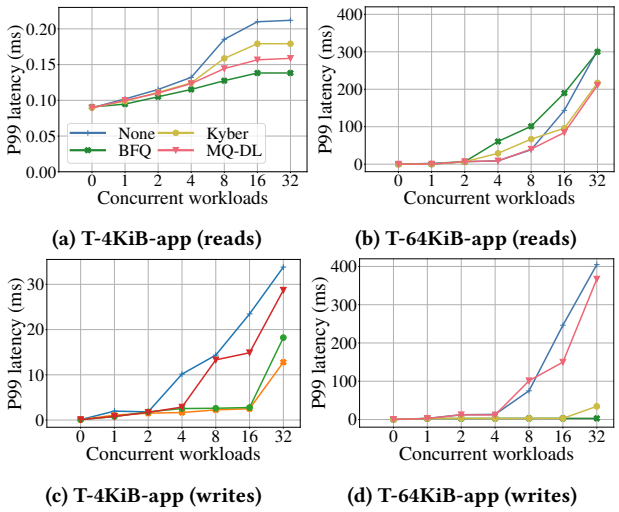


Figure 11: L-app tail latency with an increasing number of interfering background applications; *Note: scales differ on the y-axis and they are in Milliseconds!*

environments (e.g., defined as SLAs), especially for latency-critical workloads. However, there is limited literature available on the QoS of *state-of-the-practice* I/O schedulers available in Linux with modern high-speed NVMe SSDs. In this section, we evaluate the QoS of applications in multi-tenant environments. We define the QoS as the P99 tail latency for latency-sensitive applications and the IOPS for throughput-bound applications. The unique aspect of our study is that we study the mixed settings (mixing latency with throughput) as well as read-write interference, the latter of which is unique to flash-based SSDs [20, 26, 33, 40, 46, 52, 56, 57].

5.1 Latency Interference from Concurrent Read-Write T-4KiB-apps

For latency-sensitive applications (L-apps), it is essential that tail P99 latency is low and bounded even in the presence of concurrent workloads. Below, we determine if Linux I/O schedulers are able to meet such demands by controlling the interference among workloads. We devise an experiment in which one foreground L-app issues I/O requests concurrently with multiple interfering throughput-heavy T-4KiB-apps and T-64KiB-apps workloads in the background. Considering that SSD writes and reads typically have a different performance within flash that can lead to read-write interference, our background workloads issue both (random) read and write I/O requests to the SSD.

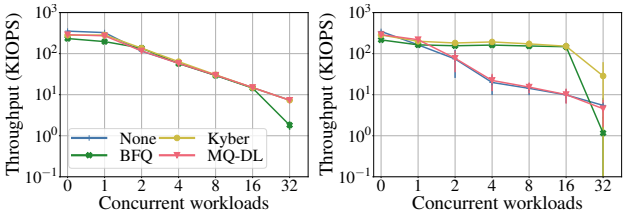
What kind of bounded P99 latencies do the I/O schedulers offer when a foreground L-app runs with background read and write T-apps? For this experiment, we measure the P99 latency of a single L-app while we increase the number of background read and write applications. Figure 11 shows our results of P99 latencies (y-axis, lower is better) with four different configurations with an increasing number of background workloads (on the x-axis): (a) multiple random read T-4KiB-apps (CPU bounded); (b) multiple random read T-64KiB-apps (SSD bounded); (c) multiple random write T-4KiB-apps (CPU bounded); and (d) multiple random write T-64KiB-apps (SSD bounded). From Figure 11 we observe that BFQ has the lowest P99 read latencies under the majority of the evaluated scenarios (except for random read T-64KiB-apps), with up to 34.8% lower latency than None (212.0 μs vs. 138.2 μs). *The lower latency for BFQ holds for small (4KiB) reads and small (4KiB) and large (64KiB) write requests* (**O-14**). We have two assumptions about why BFQ achieves low latency for the L-app: (1) the BFQ-specific option `low_latency` [1] automatically detects and provides low latency for latency-sensitive applications, which is enabled by default; and (2) BFQ gives exclusive access of the SSD to a workload (no concurrent request dispatching from multiple workloads), which reduces the interference from other workloads. From Figure 11 (c) and (d) we also report that Kyber has low latencies with concurrent small (4KiB) and large (64KiB) write requests (similar performance to BFQ for writes). However, the latencies are only slightly better than using the None scheduler for concurrent reads (Figure 11b). The reason for Kyber's low P99 latency in the presence of concurrent writes (Figure 11 (c-d)) is that Kyber prioritizes read requests over writes [8] under all circumstances. Considering that the L-app solely issues reads, this means that the foreground L-app is prioritized among the concurrently writing background workloads. *Hence the L-app with reads is prioritized among the concurrent writers, but is treated similarly and reaps no benefit with concurrent readers* (**O-15**).

We further identify that the None scheduler is not capable of bounding or guaranteeing any latency performance, and performs the worst in multiple scenarios. Specifically with read-write interferences, as shown in Figure 11 (c-d), the P99 latency deteriorates very quickly. With 1 background workload, the performance of the None scheduler is the same as other schedulers. However, as the number of background workloads increases the gap widens to as large as 139× (404.8 ms for None vs. 2.9 ms for BFQ in Figure 11d).

In Figure 11c and Figure 11d, the read latency of MQ-Deadline increases significantly (up to 32.4× with T-4KiB-apps and 4,142.2× with T-64KiB-apps) as the number of concurrent write applications increases. The performance trend is very similar to the performance trend when None scheduler is used. The reason is that MQ-Deadline does not have different priorities between reads and writes, and does not prioritize one application over the other. Therefore, the effects of read/write interference are not controlled, as requests are dispatched summarily to the SSD for both operations as they arrive in the scheduler. As the SSD becomes the bottleneck with 3-4 concurrent workloads, the I/O latencies reflect the read-write latencies of the SSD performance with interference. To summarize (**KF-3**), (i) BFQ offers better control over interference (lower P99 latencies) for latency-sensitive workloads (while trading performance, §4) than other schedulers; (ii) Kyber and BFQ excel in managing read-write interferences among concurrent workloads. However, Kyber has poor performance with read-read interference.

5.2 Throughput Interference from Read-Write Workloads

For throughput-bound applications (T-apps), throughput should be as high as possible and shared evenly among concurrent applications. Therefore, we evaluate if a foreground T-app can maintain a high throughput in the presence of concurrently running background workloads. We run a single foreground T-4KiB-app with an



(a) T-64KiB-apps (reads) (b) T-64KiB-apps (writes)

Figure 12: Read throughput (IOPS) of a T-4KiB-app workload with an increasing number of interfering background T-64KiB-app workload. Note: The y-axis is log-scale.

increasing number of concurrent T-64KiB-apps in the background; hence, introducing workload competition for throughput resources. Similar to §5.1, we evaluate the interference from both read and write background workloads. We plot the throughput of the foreground T-4KiB-app in IOPS (y-axis, higher is better) in Figure 12, with the number of concurrent background T-64KiB-apps increases.

Can I/O schedulers evenly share the IOPS performance among concurrent throughput bounded (4KiB and 64KiB) workloads? With the even sharing the expectation is that an IOPS-driven workload (T-4KiB-app) should observe linearly bandwidth deterioration as throughput-bounded workloads are run in the background. Figure 12a demonstrates that none of the schedulers are capable of offering a bounded and linear IOPS performance sharing with read-read interference, and the performance of the foreground workload drops very quickly (the y-axis is logarithmic). *An investigation of the total throughput (sum of the T-app and background operations) shows that the IOPS is divided equally among the applications* (**O-16**). However, a point to be noted here is that even though 4KiB and 64KiB applications both receive equal IOPS, the 64KiB applications get 16× more bandwidth than the 4KiB application due to their larger request size. Hence, all schedulers fail to provide equal bandwidth sharing to ensure a *proportional IOPS sharing* among the workloads.

IOPS sharing in the presence of read-write interference is managed better. *In Figure 12b we report that Kyber and BFQ offer the highest throughput for read operations for the foreground workload* (**O-17**). We have two key findings. Firstly, None and MQ-Deadline both lead to read IOPS degradation for the foreground workload in the presence of read-write interference from the background writer workloads. A reason for this is that background writes are faster than the reads (lower latencies), hence, they can occupy the device's internal bandwidth the majority of the time. None and MQ-Deadline do not have any mechanisms to throttle writes to help reads. In contrast, Kyber differentiates the two, and prioritizes the reads, hence, maintaining a flat-line performance for the foreground T-4KiB-app. The reasons that BFQ leads to a higher throughput are twofold: (1) the fair sharing where BFQ equally divides the available bandwidth between applications; (2) BFQ prioritizes interactive applications (i.e. the foreground app could have been identified as interactive). *Beyond 16 concurrent applications in Figure 12b, a bottleneck can be identified for BFQ* (**O-18**). The throughput of BFQ decreases by 99.2% between 16 and 32 concurrent applications and averages at 1.2 KIOPS, down from 147.7 KIOPS with 16 background apps. This leads to a significantly lower throughput than the throughput of the other schedulers (from 73.9% up to 95.8%). This is a consequence of its locking overheads as lock contention increases significantly after 16 processes.

Hence, our findings recommend to (**KF-4**) (i) use Kyber or BFQ to control read/write interference; (ii) be aware of concurrency limits to the schedulers as the CPU becomes 100% occupied, a configuration that can lead to significant performance losses.

6 KYBER CONFIGURATION EXPLORATION

In the previous sections, we observed that both BFQ and MQ-Deadline are unable to saturate a single fast SSD because of high CPU overhead and scalability issues, which makes them unsuitable for such SSDs. Kyber, on the other hand, is specifically designed for fast multi-queue devices and has better scalability than BFQ and MQ-Deadline. Henceforth, we focus the rest of our studies on Kyber. In particular, we look at configuring Kyber since Linux provides various configuration parameters for this scheduler. We evaluate how Kyber's parameter configuration affects its performance characteristics, and we give guidelines on configuring Kyber in practice, thus answering *how do a scheduler configuration parameters affect the schedulers' behavior on competing workloads?* (**RQ4**).

Kyber has two configurable parameters [8], `read_lat_nsec` (default = 2 ms) and `write_lat_nsec` (default = 10 ms) — we will refer to them as *R lat* and *W lat*. These parameters control the desired or target request latencies. In order to prioritize a read or a write type of I/O request, Kyber uses tokens for read and write requests. The number of tokens helps Kyber to control the maximum number of I/O requests of a particular type in flight, bounding the size of the request queue to SSDs, and as a result, the maximum latency for the I/O request. The number of tokens can not be configured with Kyber's parameters directly. Kyber controls the number of tokens by closely monitoring the current read and write completion latencies and comparing them against the target latencies, i.e., *R lat* and *W lat*. The number of tokens remains the same if both achieved read and write latencies are lower than the target latencies. If the completion latency exceeds the target latency, Kyber increases the number of tokens. Hence, setting a lower target latency for reads or writes implicitly prioritizes it (by not letting the requests build up in the queue, thus dispatching it immediately).

The number of tokens for a particular type of request (read or write) is reduced when (1) the achieved *P90 latency* for that request type is lower than the target latency (i.e., this type of request is well served); and (2) the achieved *P99 latency* for the other type is higher than the target latency (i.e., the other request type is badly served). In this case, Kyber can re-prioritize the badly served request by reducing the number of tokens for the other type. The minimum number of tokens for both read and write is 1, and the maximum number of tokens is 256 for read and 128 for write. We define the number of read tokens as *R Tokens* and write tokens as *W Tokens*. Below, we investigate how setting Kyber's target latencies affects Kyber's throughput and tail latency.

Table 2: Kyber configuration impact on the throughput of a concurrent read and write application (R and W KIOPS). Highlighted entries are discussed in §6.1.

ID	*R lat*	*W lat*	*R, in KIOPS*	*W, in KIOPS*	*R Tokens*	*W Tokens*
0	2 ms	10 ms	156.7±5.3	103.4±5.1	256±0	93±7
1	0 s	10 ms	189.4±11.5	79.4±2.2	256±0	54±15
2	2 ms	0 s	137.8±0.6	114.8±4.2	253±2	128±0
3	0 s	0 s	136.1±1.0	118.5±5.0	256±0	128±0
4	1 s	1 s	137.2±1.8	113.6±5.7	256±0	128±0
5	0 s	1 s	219.9±11.0	70.0±0.3	256±0	1±0
6	1 s	0 s	2.1±0.0	118.8±5.4	1±0	128±0

6.1 Mixing throughput-bound mixed read-write workloads

Kyber limits the maximum number of concurrent requests on the device side through the number of tokens. Limiting the number of requests decreases the throughput, which prevents throughput saturation and leads to SSD idling. Thus, the hypothesis is that changing the target latency directly affects the throughput. To test this hypothesis, we run a foreground T-4KiB-app with random reads and a background T-4KiB-app with random writes concurrently. We issue I/O requests to one SSD and pin each process to a different CPU core to avoid interference with the process scheduler. Table 2 shows the read and write throughput (and accompanying number of tokens) under various Kyber configurations (identified with unique *ID*s). The default configuration is *ID* 0 (*R lat*: 2 ms, *W lat*: 10 ms).

If we change the default *R lat* to the minimal value (0), the read throughput increases significantly (20.9%) at the cost of write throughput (a 23.2% decrease), a change that is reflected as a significant decrease in the number of write tokens (93 to 54). Similarly, when *W lat* is changed to the minimal value (0), the write throughput increases significantly (11.0%) at the cost of read throughput (a 12.1% decrease), a change that is reflected as a slight decrease in the number of read tokens (256 to 253). *In short, setting the target latency to the minimum for a particular type of request (read or write) leads to a significant increase in the throughput (up to 20.9%) at the cost of the other type's throughput degradation (up to 23.2%)* (**O-19**).

In *ID* 3–4, we set *R lat* and *W lat* both to the same unrealistic extreme values, the minimum (0 s, highest priority) and an arbitrarily high value (1 s, least priority). This leads to the maximum number of tokens for both reads and writes (256 for read and 128 for write). Lastly, we set *R lat* or *W lat* to 0 s while the other is set to 1 s (*ID* 5, 6). With this configuration, we try to get the lowest latency possible for either reads or writes. *This configuration leads to a significant throughput increase for the prioritized target request type (40.3% for reads, 14.9% for writes), but leads to a significant throughput decrease for the other type also (98.7% for reads, 32.3% for writes)* (**O-20**).

6.2 Latency-sensitive Read Workload with a Write-driven Throughput Workload

To investigate how configuring Kyber affects the latency of L-apps with background write T-4KiB-apps, we run an L-app (read) with a background T-4KiB-app (write) on a single SSD, with each application pinned to a separate CPU core. In Table 3 we show the P99 tail latency in milliseconds of L-apps with the number of *W Tokens* shown in parentheses. The chosen *W lat* is represented with the columns and the *R lat* with the rows.

Table 3: Kyber configuration impact on read P99 latency in milliseconds and W tokens (presented in "()") of an L-app running with an interfering T-4KiB-app (random write). The highlighted entries are discussed in §6.2.

		W lat				
		0 s	100 μs	10 ms	20 ms	100 ms
R lat	0 s	2.8 (128)	2.7 (119)	1.4 (10)	1.4 (1)	1.6 (1)
	50 μs	2.8 (128)	2.7 (119)	1.5 (17)	1.7 (1)	1.6 (1)
	100 μs	2.8 (128)	2.8 (116)	1.7 (39)	1.6 (1)	1.5 (1)
	500 μs	2.8 (128)	2.7 (118)	1.9 (50)	1.5 (1)	1.5 (1)
	10 ms	2.8 (128)	2.8 (128)	2.8 (128)	2.8 (128)	2.8 (128)
	100 ms	2.8 (128)	2.8 (128)	2.8 (128)	2.8 (128)	2.8 (128)

We first find a general trend between the achieved read latency and the number of W Tokens: the higher the number of W Tokens, the worse the achieved read latency (**O-21**). When the *R lat* is set to an extremely high value that is unlikely to be reached by the hardware (in our case higher than 10 ms) or the *W lat* is set to the minimum value (0), the number of *W Tokens* is set to the largest value, i.e., 128, meaning that reads are not prioritized, leading to high read latency (2.8 ms). These entries are shown with gray backgrounds. When *R lat* is set to a low value (less than or equal to 500 μs), and the gap between *R lat* and *W lat* increases, the number of *W Tokens* decreases (from 128 to 1), meaning that reads are more prioritized, leading to decreasing read latency (2.8 ms to 1.4 ms, 50% lower). These entries are shown with green backgrounds. *The larger the gap between R lat and W lat (R lat is lower than W lat), the more reads are prioritized, thus improving the achieved read latency (up to 50% lower)* (**O-22**).

6.3 Write-driven Throughput Workload with a Read Workload

To determine the effect of Kyber's parameters on throughput-bound applications in mixed workloads, we use the same experiment as used for evaluating the effect on latency-sensitive applications, but this time reporting the write latency for the T-4KiB-app. In Table 4, we show the result with throughput in KIOPS (higher is better) of the foreground T-4KiB-app writing workload. The number of write tokens is already shown in Table 3. Since reads only have QD=1 (L-app), the number of read tokens has no influence on the read performance, and we do not show the number of *R Tokens*. The configured *W lat* is represented by the columns and the configured *R lat* is depicted in the table rows. We observe that the lower the number of *W Tokens*, the lower the write throughput. *Configuring W lat to a high value (higher than 10 ms for our SSD), while setting R lat to a low value (less than 500 μs) has a negative impact on the write throughput (up to 43.5% lower write throughput)* (**O-23**). These entries are shown with gray backgrounds. If we combine the result with the previous table, we observe that if Kyber provides lower read latency, it comes at the cost of write throughput. Thus, when configuring Kyber for a workload, either read latency or write throughput can be prioritized, but not both (**KF-5**).

Table 4: Kyber configuration impact on write throughput in KIOPS of a T-4KiB-app running with an interfering L-app (random read). The highlighted entries are discussed in §6.3.

R lat \ W lat	0 s	100 μs	10 ms	20 ms	100 ms
0 s	129.4	103.4	77.3	73.6	73.1
50 μs	116.5	111.6	80.3	71.2	73.3
100 μs	128.8	128.0	82.4	73.2	73.3
500 μs	122.2	115.6	96.0	73.8	73.5
10 ms	119.6	130.9	127.5	125.4	131.8
100 ms	135.0	123.7	119.7	129.7	118.0

7 RELATED WORK

On *State-of-the-Art* I/O Schedulers. In this work, we investigate Linux's *state-of-the-practice* I/O schedulers. However, there is also a vast literature on *state-of-the-art* schedulers, which are not included with the default Linux kernel and require domain expertise. Therefore, we consider such work orthogonal. Modern SSDs use multiple channels internally to deliver high throughput and have complex internal mechanisms such as FTL, buffers, and garbage collection (GC). These idiosyncrasies require more novel designs than the *state-of-the-practice* Linux I/O schedulers. Here, we discuss *state-of-the-art* schedulers for fair-sharing, low-latency applications, and schedulers that use flash-specific functionalities.

There are many fair-sharing I/O schedulers [17, 21, 26, 46, 52, 53, 56, 57] that can be used as alternatives for the fair-sharing BFQ scheduler. Modern SSDs support multiple hardware dispatch queues and it is necessary to use multiple queues to fully saturate the SSD's performance and to provide fair sharing [56]. Additionally, NVMe features like weight round robin (WRR) allow assigning weights to applications, which can also be used to guarantee fairness across applications. This leads to designs that use multiple queues such as MQFQ [26] and multiple queues with WRR such as D2FQ [56]. We do not measure fair-sharing in this work since most Linux schedulers do not support it.

There are also a number of *state-of-the-art* schedulers optimized for latency-sensitive applications like the L-app used in this paper [27, 39, 42]. Such schedulers can be used instead of the Linux' Kyber and BFQ schedulers, which both have options to prioritize low-latency applications. This includes solutions such as blk-switch [27], which provide low latency while preserving high total throughput, and FastResponse [39] that co-designs the I/O stack and scheduler to reduce cross-layer I/O interference.

Flash-based SSDs have many idiosyncrasies that can be exploited by schedulers [22, 23, 29, 31, 52, 54]. For example, increasing SSD longevity by reducing wear-levelling [54] or reducing GC overhead [22, 23, 29]. Generally, flash-aware schedulers also treat writes and reads differently because of read/write interference [31, 46].

On Performance Characterizations. Parallel to this work, the performance of the Linux storage stack is characterized in many papers. Whitaker et al. [55] characterize the performance of Linux' I/O schedulers for ULL non-flash-based SSDs. Their findings confirm that schedulers inevitably lead to higher latency and lower throughput. Additionally, this work looks at energy efficiency, where they find that schedulers have high energy overhead, especially for BFQ. We extend this work by looking at QoS and looking at the more common flash storage (ULL SSDs are not widely deployed yet). The Linux user community has investigated I/O schedulers for SSDs as well [34], and their work showcases that using no scheduler, followed by Kyber, leads to the highest throughput and low latency for applications like MySQL and RocksDB. There are also various works that do performance characterization of emerging storage APIs such as `io_uring` [19, 24, 51]. Our work differentiates itself in terms of scale, comprehensiveness, and its sole focus on the performance characterization of Linux storage I/O schedulers.

8 CONCLUSION

In this paper, we investigate if the Linux I/O schedulers fit modern NVMe SSDs. Our results show that BFQ and MQ-Deadline have significantly high CPU overhead and scalability issues caused by locking. Thus, we suggest that BFQ and MQ-Deadline should not be used with these SSDs. Kyber has lower CPU overhead than BFQ and MQ-Deadline with near-linear scalability and thus is the best fit of these SSDs. However, the parameters of Kyber need to be tuned carefully or Kyber harms the performance. Our analysis focuses on the Linux I/O schedulers. This work can be expanded in (1) evaluating how the start-of-the-art I/O schedulers perform on the flash-based NVMe SSDs, (2) comparing how different I/O scheduling algorithms and techniques work on these SSDs and (3) optimizing the current Linux I/O schedulers to make them SSD-friendly.

Acknowledgments We thank the ICPE'24 reviewers for their invaluable and constructive feedback. This work is funded by The Dutch Research Council (NWO) grant numbers OCENW.KLEIN.561 and OCENW.KLEIN.209. The authors would like to thank Jesse Donkervliet, Sacheendra Talluri, Matthijs Jansen, and the AtLarge group at VU Amsterdam for their help with the paper. Krijn Doekemeijer is funded by the VU PhD innovation program.

REFERENCES

[1] Accessed: 2024-01-29. BFQ Budget Fair Queueing Document. https://www.kernel.org/doc/html/latest/block/bfq-iosched.html
[2] Accessed: 2024-01-29. Deadline I/O Scheduler Tunables. https://docs.kernel.org/block/deadline-iosched.html#:~:text=The%20goal%20of%20the%20deadline,value%20in%20units%20of%20milliseconds.
[3] Accessed: 2024-01-29. *Disaggregated or Hyperconverged, What Storage will Win the Enterprise?* https://www.nextplatform.com/2017/12/04/disaggregated-hyperconverged-storage-will-win-enterprise/
[4] Accessed: 2024-01-29. *fio.* https://github.com/axboe/fio
[5] Accessed: 2024-01-29. *Free Your Flash and Disaggregate.* https://www.lightbitslabs.com/blog/free-your-flash-and-disaggregate/
[6] Accessed: 2024-01-29. getrusage(2) — Linux Manual Page. https://man7.org/linux/man-pages/man2/getrusage.2.html
[7] Accessed: 2024-01-29. Intel® Optane™ SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html
[8] Accessed: 2024-01-29. Kyber Multiqueue I/O Scheduler. https://lwn.net/Articles/720071/
[9] Accessed: 2024-01-29. MQ-Deadline Implementation. https://elixir.bootlin.com/linux/latest/source/block/mq-deadline.c
[10] Accessed: 2024-01-29. *SPDK.* https://spdk.io/
[11] Accessed: 2024-01-29. *Toshiba Memory Introduces XL-FLASH Storage Class Memory Solution.* https://americas.kioxia.com/en-us/business/news/2019/memory-20190805-1.html
[12] Accessed: 2024-01-29. Two New Block I/O Schedulers for 4.12. https://lwn.net/Articles/720675/
[13] Accessed: 2024-01-29. *What is Composable Disaggregated Infrastructure.* https://blog.westerndigital.com/what-is-composable-disaggregated-infrastructure/
[14] Jens Axboe. Accessed: 2023-01-26. Efficient I/O with io_uring. https://kernel.dk/io_uring.pdf

[15] Jon C. R. Bennett and Hui Zhang. 1997. Hierarchical Packet Fair Queueing Algorithms. *IEEE/ACM Trans. Netw.* 5, 5 (1997), 675–689.

[16] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80.

[17] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols, SIGCOMM 1989*. ACM, 1–12.

[18] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proc. VLDB Endow.* 14, 3 (2020), 364–377.

[19] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of libaio, SPDK, and io_uring. In *SYSTOR '22: The 15th ACM International Systems and Storage Conference, 2022*. ACM, 120–127.

[20] Krijn Doekemeijer, Nick Tehrany, Balakrishnan Chandrasekaran, Matias Bjørling, and Animesh Trivedi. 2023. Performance Characterization of NVMe Flash Devices with Zoned Namespaces (ZNS). In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. 118–131.

[21] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. 1996. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of the ACM SIGCOMM 1996 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 1996*. ACM, 157–168.

[22] Jiayang Guo, Yimin Hu, and Bo Mao. 2015. Enhancing I/O Scheduler Performance by Exploiting Internal Parallelism of SSDs. In *Algorithms and Architectures for Parallel Processing - 15th International Conference, ICA3PP 2015. Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 9531)*. Springer, 118–130.

[23] Jiayang Guo, Yiming Hu, Bo Mao, and Suzhen Wu. 2017. Parallelism and Garbage Collection Aware I/O Scheduler with Improved SSD Performance. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017*. IEEE Computer Society, 1184–1193.

[24] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2090–2102.

[25] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017*. ACM, 127–144.

[26] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019*. USENIX Association, 301–314.

[27] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021*. USENIX Association, 113–128.

[28] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek R. Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018*. USENIX Association, 519–532.

[29] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. 2014. HIOS: A Host Interface I/O Scheduler for Solid State Disks. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014*. IEEE Computer Society, 289–300.

[30] Jungkil Kim, Sungyong Ahn, Kwanghyun La, and Wooseok Chang. 2016. Improving I/O Performance of NVMe SSD on Virtual Machines. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016*. ACM, 1852–1857.

[31] Jieun Kim, Dohyun Kim, and Youjip Won. 2022. Fair I/O Scheduler for Alleviating Read/Write Interference by Forced Unit Access in Flash Memory. In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, 2022*. ACM, 86–92.

[32] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016*. ACM, 29:1–29:15.

[33] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash = Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, 345–359.

[34] Michael Larabel. Accessed: 2023-11-16. Linux 5.6 I/O Scheduler Benchmarks: None, Kyber, BFQ, MQ-Deadline. https://www.phoronix.com/review/linux-56-nvme

[35] Michael Larabel. Accessed: 2024-02-28. BFQ I/O Scheduler For Linux Sees Big Scalability Improvement, published: 21 January 2024. https://www.phoronix.com/news/BFQ-IO-Better-Scalability

[36] Michael Larabel. Accessed: 2024-02-28. MQ-Deadline Scheduler Optimized For Much Better Scalability, published: 19 January 2024. https://www.phoronix.com/news/MQ-Deadline-Scalability

[37] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, 2020*. ACM, 591–605.

[38] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan. 2020. Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. USENIX Association, 1241–1255.

[39] Mingzhe Liu, Haikun Liu, Chencheng Ye, Xiaofei Liao, Hai Jin, Yu Zhang, Ran Zheng, and Liting Hu. 2022. Towards Low-Latency I/O Services for Mixed Workloads Using Ultra-Low Latency SSDs. In *ICS '22: 2022 International Conference on Supercomputing, 2022*. ACM, 13:1–13:12.

[40] Hui Lu, Brendan Saltaformaggio, Ramana Rao Kompella, and Dongyan Xu. 2015. vFair: Latency-Aware Fair Storage Scheduling via per-IO Cost-Based Differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015*. ACM, 125–138.

[41] Liuying Ma, Zhenqing Liu, Jin Xiong, and Dejun Jiang. 2022. QWin: Core Allocation for Enforcing Differentiated Tail Latency SLOs at Shared Storage Backend. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022*. IEEE, 1098–1109.

[42] Till Miemietz, Hannes Weisbach, Michael Roitzsch, and Hermann Härtig. 2019. K2: Work-Constraining Scheduling of NVMe-Attached Storage. In *IEEE Real-Time Systems Symposium, RTSS 2019*. IEEE, 56–68.

[43] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, 106–122.

[44] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*. USENIX Association, 361–378.

[45] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014*. ACM, 471–484.

[46] Stan Park and Kai Shen. 2012. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012*. USENIX Association, 13.

[47] Bo Peng, Cheng Guo, Jianguo Yao, and Haibing Guan. 2023. LPNS: Scalable and Latency-Predictable Local Storage Virtualization for Unpredictable NVMe SSDs in Clouds. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023*. USENIX Association, 785–800.

[48] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. 2018. MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018*. USENIX Association, 665–676.

[49] Purestorage. Accessed: 2024-01-29. *Linux Recommended Settings*. https://support.purestorage.com/Solutions/Linux/Linux_Reference/Linux_Recommended_Settings

[50] RedHat. Accessed: 2024-01-29. *Chapter 20. Setting the Disk Scheduler*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_storage_devices/setting-the-disk-scheduler_managing-storage-devices

[51] Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, CHEOPS 2023*. ACM, 35–45.

[52] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *2013 USENIX Annual Technical Conference, 2013*. USENIX Association, 67–78.

[53] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. 2007. Argon: Performance Insulation for Shared Storage Servers. In *5th USENIX Conference on File and Storage Technologies, FAST 2007*. USENIX, 61–76.

[54] Mingyang Wang and Yiming Hu. 2014. An I/O Scheduler Based on Fine-Grained Access Patterns to Improve SSD Performance and Lifespan. In *Symposium on Applied Computing, SAC 2014*. ACM, 1511–1516.

[55] Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak. 2023. Do We Still Need I/O Schedulers for Low-Latency Disks?. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. 44–50.

[56] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. 2021. D2FQ: Device-Direct Fair Queueing for NVMe SSDs. In *19th USENIX Conference on File and Storage Technologies, FAST 2021*. USENIX Association, 403–415.

[57] Minhoon Yi, Minho Lee, and Young Ik Eom. 2017. CFFQ: I/O Scheduler for Providing Fairness and High Performance in SSD Devices. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication, IMCOM 2017*. ACM, 87.

The Cost of Simplicity: Understanding Datacenter Scheduler Programming Abstractions

Aratz Manterola Lasa*
aratz@warpstreamlabs.com
WarpStream Labs
USA

Sacheendra Talluri
s.talluri@vu.nl
Vrije Universiteit Amsterdam
The Netherlands

Tiziano De Matteis
t.de.matteis@vu.nl
Vrije Universiteit Amsterdam
The Netherlands

Alexandru Iosup
a.iosup@vu.nl
Vrije Universiteit Amsterdam
The Netherlands

ABSTRACT

Schedulers are a crucial component in datacenter resource management. Each scheduler offers different capabilities, and users use them through their APIs. However, there is no clear understanding of what programming abstractions they offer, nor why they offer some and not others. Consequently, it is difficult to understand their differences and the performance costs imposed by their APIs. In this work, we study the programming abstractions offered by industrial schedulers, their shortcomings, and their related performance costs. We propose a general reference architecture for scheduler programming abstractions. Specifically, we analyze the programming abstractions of five popular industrial schedulers, understand the differences in their APIs, and identify the missing abstractions. Finally, we carry out exemplary experiments using trace-driven simulation demonstrating that an API extension, such as container migration, can improve total execution time per task by 81%, highlighting how schedulers sacrifice performance by implementing simpler programming abstractions. All the relevant software and data artifacts are publicly available at https://github.com/atlarge-research/quantifying-api-design.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Software and its engineering** → **Cloud computing**.

KEYWORDS

cloud, scheduler, API, design, performance

ACM Reference Format:
Aratz Manterola Lasa, Sacheendra Talluri, Tiziano De Matteis, and Alexandru Iosup. 2024. The Cost of Simplicity: Understanding Datacenter Scheduler Programming Abstractions. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645038

*This work was done while the author was affiliated with Vrije Universiteit Amsterdam.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645038

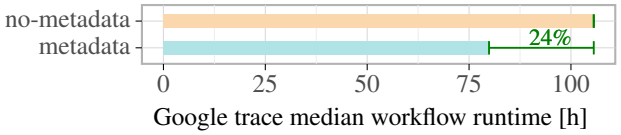


Figure 1: Performance penalty due to a missing programming abstraction: storage metadata access.

1 INTRODUCTION

Society's increasing dependence on digital technologies and infrastructure has led to the widespread use of datacenters for deploying digital services [20, 28]. Schedulers play a vital role in orchestrating datacenter resources to meet the demands of these services [24, 45]. The interfaces schedulers offer to users determine the limits of the users' ability to mold the orchestration process to support their application needs. Different schedulers offer different levels of programmability and control to users [27, 47, 51, 58]. For example, some schedulers provide restricted programming abstractions [1], minimizing user input, while others offer more flexible interfaces that empower users with greater control over resource allocation and job placement [47, 58]. This spectrum of scheduler programming abstractions raises questions about the impact of design choices on performance, simplicity, and control that users can achieve.

The first question we raise about scheduler abstraction design is: **What programming abstractions are common in current schedulers?** Knowledge of programming abstractions in existing industrial schedulers informs designers of what is currently available to the users. The programming abstractions available in academic research schedulers can also suggest to designers which abstractions are necessary to incorporate the latest resource management techniques proposed by the research community.

The second question is: **What programming abstractions are sacrificed for simplicity?** Usually, academic schedulers offer a wide set of programming abstractions, allowing the users to customize several aspects of scheduler operational behavior. On the other hand, industrial schedulers usually implement a restricted subset for increased security and robustness [46].

The third question is: **What is the performance cost of the sacrificed abstractions?** Despite their security and robustness

[1]We use programming abstraction and API interchangeably.

benefits, simpler abstractions have a performance cost. The performance cost is usually in the form of underutilized resources and slow-to-complete application jobs. To shed light on this issue, we conduct three different experiments. Figure 1 depicts an exemplary result with the median execution time of workflows in a trace from Google [55]. We consider a scheduler that implements a crucial abstraction lacking in many industrial schedulers: metadata access to the data stored on datacenters' object storage service (e.g., AWS S3). Comparing it against a scheduler lacking this abstraction, we observe a 24% reduction in median workflow runtime when using the abstraction.

To address these questions and enhance our understanding of scheduler programming abstractions, we develop a comprehensive and structured reference architecture that provides a unified view of the programming interfaces offered by schedulers. This reference architecture compliments earlier work on scheduler internals [5, 30]. It guides developers and researchers in designing and implementing scheduling APIs, capturing the essential abstractions in task scheduling and resource management within datacenter environments.

Establishing a common reference architecture brings several benefits. First, the reference architecture provides a common framework for analyzing and comparing existing industrial and academic schedulers. The comparison helps identify similarities, differences, and potential shortcomings, thus enabling the assessment of different implementations and design alternatives [5]. Second, it serves as a knowledge base for designing better schedulers that can meet the demands of modern applications by addressing shortcomings [5, 11, 22, 36]. Finally, establishing a common reference model reduces the risk of a scheduler being specialized to the current interface by providing a view of all possible programming interfaces. This helps avoid non-extensible designs that must be re-engineered at great development cost, as has been the case with Condor [51] and Borg [10] when the need for a new design arises.

To understand datacenter scheduler programming abstractions and the cost of missing ones, we make a four-fold contribution:

(1) We design a reference architecture for datacenter scheduler programming abstraction (Section 3). We propose a set of design principles and, with them, design an architecture that considers different stakeholders and the programming abstractions of existing schedulers.
(2) We analyze existing industrial and academic schedulers by mapping them to the reference architecture (Section 4). This mapping allows us to compare them using a common language. The comparison reveals abstractions proposed in literature but missing from industrial schedulers.
(3) We analyze the effect of missing abstractions on the performance of modern schedulers (Section 5). To this end, we implement three missing abstractions in an event-driven simulator and conduct simulations using real-world traces collected by major datacenter operators, e.g., Google and Microsoft.
(4) We contribute to open science and reproducibility by releasing data and software artifacts. To enable the experiments in this work, we have significantly extended OpenDC [39], a state-of-the-art simulator. We release the code enabling this work's capabilities through Github: https://github.com/atlarge-research/quantifying-api-design. The repository has been archived using Zenodo at: https://zenodo.org/doi/10.5281/zenodo.10605424.

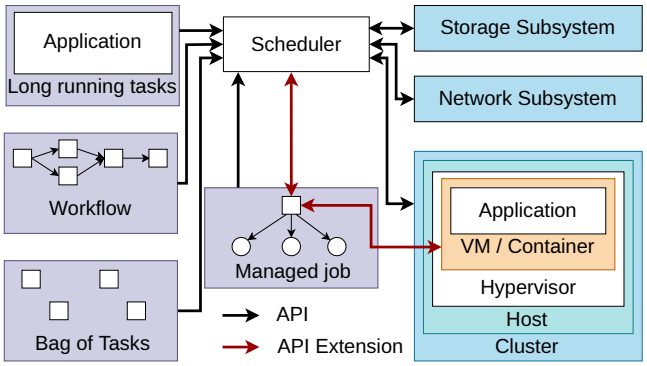


Figure 2: Datacenter scheduling system model.

2 DATACENTER SCHEDULER SYSTEM MODEL

This section contextualizes this work by describing common datacenter scheduling-related concepts depicted by Figure 2.

2.1 Workload

The workload is executed using the resources the scheduler assigns to the user. Following the taxonomy proposed by Andreadis et al. [5], we consider four types of workloads:

(1) **Batch workflows** are workloads comprising several tasks with dependencies between them.
(2) **Bag-of-tasks** are jobs formed by several tasks without any dependency between them.
(3) **Long running tasks** run for a very long time and are usually inside a host such as a VM.
(4) **Managed jobs** are workloads where a manager coordinates all the tasks, such as Spark.

The users specify the requirements to execute the workload. Usually, these comprise the amount of CPU and memory. However, in some cases, other requirements, such as the start time, the dependencies between the tasks, the scalability of the resources, etc., are also specified. To submit the workload requirements, users interact with the scheduler through its API.

2.2 Scheduling

A user submits a workload to use the resources through a central component, the scheduler [10, 41]. The scheduler takes care of several tasks: finding resources to assign to the workload based on the specified requirements, transferring the workload to the resources, starting the execution of the workload, managing the workload through its lifecycle (from placement to workload cleanup), and notifying to the user about lifecycle events.

Throughout the execution of a workload, the resource requirements of the workload and the number of resources available to the scheduler can change. Therefore, the scheduler must adapt to changing workload requirements by increasing or decreasing dynamically allocated resources. This is usually done through a specific subcomponent (e.g., the *autoscaler* in Kubernetes [2]). A

scheduler can also preempt, recover, and migrate workloads when the amount of available resources changes.

Schedulers can be monolithic [35] and run in a single process that handles all tasks. They can be distributed where tasks are split into other components, such as the autoscaler [2]. In the same way, the scheduler and its members can be replicated in several processes in parallel. Still, they must coordinate among themselves when assigning resources to the workloads. In addition, schedulers can be centralized [42], where a single entity implements the scheduler and dictates the policies and mechanisms, or it can be decentralized [51] so that several entities implement a scheduler. Each of them has different policies and mechanisms. When the scheduler is decentralized, the other instances must coordinate through a common protocol and sometimes use a central matchmaker.

2.3 Scheduler resources

The workloads are executed on top of the resources that the scheduler manages. Resources typically refer to physical machines usually located within a datacenter. These datacenters consist of multiple clusters, each housing several hosts, with each host functioning as a node within a rack. It is important to note that while our discussion primarily focuses on virtualized resources such as VMs or containers running on hosts through a hypervisor, it is also possible to manage bare metal resources. However, virtualized environments are more prevalent and present a wider range of interesting phenomena for modeling and analysis.

In this work, we model the resources of a host as the combination of CPU, memory RAM, and storage. CPUs can have different frequencies and number of cores. Memory and storage can have different sizes. We model resource consumption using a discrete model, where the workload reports how many resource it requires and for how long. The hypervisor consolidates the consumption of the different workloads through a fair-sharing policy.

2.4 Programming abstraction

Schedulers offer a set of programming abstractions for users to interact with. Programming abstractions are the API offered by schedulers and are the language by which the user submits workloads and modifies the workload's requirements during the workload's life cycle. Programming abstractions are offered through a GUI, CLI, or a protocol such as HTTP.

The API includes both the interactions of the scheduler with the applications and the resources. In this work, we investigate API extensions that allow the scheduler to interact with applications and the resources allocated after the initial resource allocation.

Resource management systems, such as autoscalers, interact with schedulers and other resource managers in a completely automated manner without any user intervention. We consider the API between these different systems a part of the scheduler programming abstraction. The API constrains the actions available to these systems. Obtaining system data and performing actions not supported by the API is difficult for the systems we analyze in this work.

3 REFERENCE ARCHITECTURE FOR SCHEDULER PROGRAMMING ABSTRACTIONS

We propose a *reference architecture* to understand and describe standard programming abstractions available in current schedulers. With systematic categorization and organization, the reference architecture will offer a framework for analyzing and comparing existing schedulers and a comprehensive view of the range of common abstractions that a scheduler can implement. This helps us answer the question *What are the programming abstractions common in current schedulers?*.

Our process for designing the reference architecture has the following steps:

(1) Stakeholder and use case identification
(2) Requirements analysis
(3) Model industrial schedulers
(4) Model emerging concepts from academia
(5) Unify industrial schedulers with emerging concepts

We describe our requirements in Section 3.1. We identify five popular schedulers in the industry, and we analyze their APIs. Consulting experts in the field, we select the following schedulers: Kubernetes [3], SLURM [35], Spark [56], Condor [51], and Airflow [1]. We further analyze these schedulers in Section 4.

For emerging concepts from academia, we conduct a systematic literature survey [33], sort the results by citations, and pick the top 15 papers with new APIs different from what we identified in industrial schedulers. We end up analyzing the following 15 academic schedulers: [9, 12, 15, 17, 18, 25, 31, 32, 38, 44, 48, 50, 53, 54, 59].

After analyzing industrial and emerging scheduler designs from academia, we extract, filter, generalize, and unify them into a reference architecture.

3.1 Requirements

We identify the requirements that must be met by the reference architecture. This has to be:

R1 Understandable. Different stakeholders should be able to easily understand the different components that make up the reference architecture, how they relate to each other, or their high-level meaning. We enable this through the principles in Section 3.2 and the description language in Section 3.3.

R2 Actionable. The design must take into account whether users can use it to take concrete actions. We use the architecture in Section 4 to identify missing abstractions in industrial schedulers. We quantify the cost of missing abstractions in Section 5.

R3 Pragmatic. The reference architecture concepts can be implemented in code and evaluate different programming abstractions comparatively. The reference architecture has been realized in the OpenDC simulator and used for experiments in Section 5.

R4 Comprehensive. Can represent *all* already known concepts used in industrial schedulers and emerging concepts from academia. We map five industrial schedulers to the reference

architecture in comparison in Section 4. The reference architecture was built by analyzing 15 research prototypes from the community.

3.2 Design principles

For the design of the scheduling programming abstractions reference architecture, we identify the following design principles.

- **P1 Separation of objects from actions.** We distinguish between the actions that can be performed and the objects, which represent the system's state, that are used as input to the actions. This separation facilitates comprehension (**R1**).
- **P2 Grouping of related actions.** There may be several actions that are related to each other. Therefore, to facilitate comprehension, related actions are grouped.
- **P3 Avoidance of concrete technologies in objects.** We keep the objects as high-level as possible to avoid strong coupling to a specific technology.

3.3 Reference Architecture Design

We analyze industrial and emerging scheduler designs from academia for scheduling abstractions. Then we extract, filter, generalize, and unify them into a reference architecture. In this process, we follow the requirements and design principles we set out in the previous subsections. The reference architecture allows us to describe the different abstractions provided by the schedulers we analyzed using a common language. This common language allows enables us to compare the schedulers' APIs to each other in Section 4.

The reference architecture is depicted in Figure 3. The high-level components of the reference architecture are *actions* and *objects* that comprise the scheduler API. Object describes the current or desired state of the system. Actions describe physical events (such as leasing a VM) that are executed when certain conditions are met. The conditions use objects in their specification. Each action must have three types of conditions: WHAT, WHEN, and WHERE, and for each condition, there can be one or more objects. This way, programming abstractions can be understood through the following syntactic structure: `<action> <object> IN <object> WHEN <object>`, where the objects and actions are filled using the reference architecture.

Listing 1: Example scheduler action.

```
Provision:Lease
UserResource<type:job, runtime:5 days>
IN SchedulerResource<type:vm,
                cpu:2.4Ghz, memory:16Gb>
WHEN Event<day:11, month:12, year:2023>
```

Consider the scheduler interaction in Listing 1; the action is "Provision:Lease," indicating the provisioning and leasing of resources. The objects involved are "UserResource" with specific characteristics such as job type and a runtime of 5 days, and "SchedulerResource" with attributes like VM type, CPU of 2.4GHz, and memory of 16GB. The condition "IN" specifies that the "UserResource" is allocated within the "SchedulerResource". Lastly, the "WHEN" condition indicates an event occurring on December 31, 2022.

Tables 1 and 2 define and describe the actions and objects within the reference architecture. These tables serve as a resource for understanding the specific elements of the reference architecture and their respective functionalities.

In addition to the visual representation of the reference architecture for scheduling programming abstractions shown in Figure 3, we have also defined a formally defined syntax which we use in Listing 1. The syntax is based on the Extended Backus–Naur Form (EBNF) and provides a structured and consistent way to express conditions using actions and objects in the programming abstractions. Due to space constraints, we do not present the formal syntax definition here but will add it as an appendix. The formally defined syntax enables precise communication using the reference architecture.

4 ANALYSIS OF INDUSTRIAL SCHEDULERS

We analyze the scheduler APIs of industrial schedulers by qualitatively mapping their features to the reference architecture: Kubernetes (v1.27) [3], SLURM (v23.02) [35], Spark (v3.4.0) [56], Condor (v10.4.3) [51], and Airflow (v3.3.0) [1]. Through the mapping, we respond to the question of *What programming abstractions are sacrificed for simplicity?* The mapping provides insights into their alignment with the idealized model. This comparison helps identify missing abstractions compared to all the ones in the reference architecture, highlighting potential shortcomings.

4.1 The mapping process

For each considered scheduler, we consult its official documentation, source code, and articles we find online. Then, using these resources, for each component of the reference architecture, we identify if there is a *complete*, *partial*, or *no match*. The meaning of the match is different for objects than for model actions. In the case of actions, a complete match is when the scheduler offers the action. A partial match is when the action is offered in a limited way; that is, the action may only be offered at a specific moment in the lifecycle, e.g., it only allows to scale when the CPU utilization is more than 80%, or when the parameters with which the action can be performed are limited, e.g., a service can only be scaled by adding VMs of the same type of resources. A no-match is when the scheduler does not offer the action. In the case of objects, a full match means that the scheduler restricts the object parameters, and the user can flexibly specify whatever parameters they need. For example, the user can add any metadata information. A partial match means the scheduler allows the user to specify only a limited set of object parameters. For example, the user can only specify CPU constraints, not any other resource type. A no-match means that the scheduler does not allow that object type.

4.2 Mapping results

Using the reference architecture, we analyze the shortcomings of the selected group of five industrial schedulers. Currently, it is not known when nor why you should use some schedulers and not others. It is also unclear if any scheduler has a clear missing gap or how to fill it. For that, it is necessary to analyze the scheduling APIs. We map their APIs into the reference architecture and aggregate the results in two tables. In Table 3, we map the actions, and in Table 4, the objects. We specify whether each action and object is a full, partial, or no match.

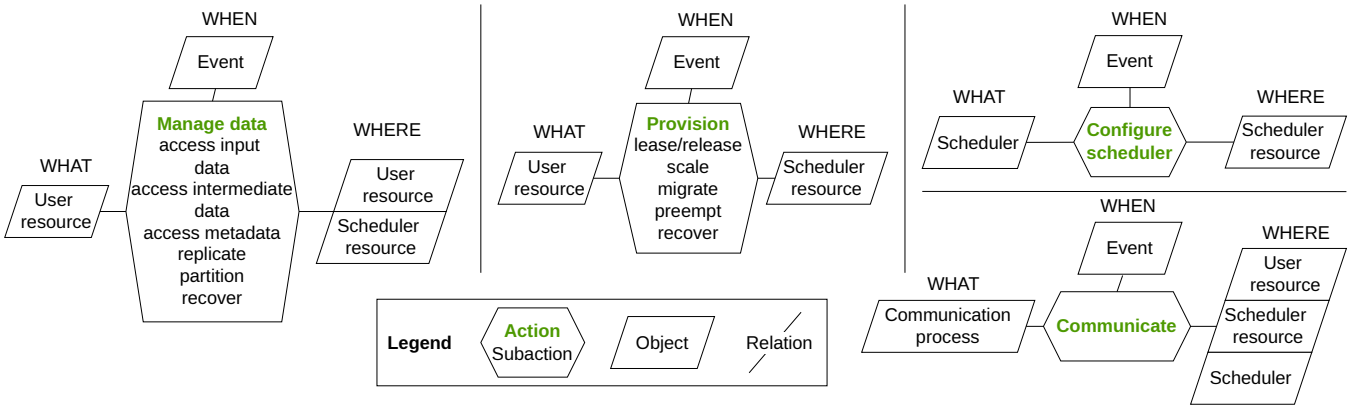


Figure 3: Reference architecture for scheduling programming abstractions.

Table 1: Description of the actions that compose the reference architecture.

Action	Sub-action	Description
Provision	Lease/release	Activation and assignment of a user resource to a scheduler resource.
	Scale	Addition or reduction of already provisioned user resources.
	Migrate	Migration of a user resource to a different scheduler resource.
	Preempt	Abortion of execution or assignment of a user resource, putting it back in the scheduler queue.
	Recover	Recover a task after failure, restart execution, or put it back into the scheduler queue.
Configure scheduler		Configuration of the behavior of the scheduler.
Replicate	Access input data	Access to data that user jobs take as input.
	Access intermed.	Access to data that user jobs generate during their runtime.
	Access metadata	Access to the information about the user data.
	Replicate	Replication of the user data.
	Partition	Partitioning of the user data so that a subset of the data is placed in different scheduler resources.
	Recover	Recovery of the user data after the failure of execution or the storage system.
Communicate		Communication with the user resources, scheduler resources, or even the scheduler, such as setting a callback for getting notified about scheduling events.

Table 2: Objects in the reference architecture.

Object	Description
Event	Representation of objects in time or instantiations of properties in objects. Such as concrete date-times (00:00 of 31st of December 2022) or an instantiation of a property like a metric reaching a numeric value (CPU utilization is greater than 80%).
User resource	Representation of any kind of input from the user. This includes execution units like a job, task, etc., but also data as a file, environment variable, etc.
Scheduler resource	Representation of resources owned and managed by the scheduler. Resources can be virtual machines, containers, storage systems, databases, etc.
Communication process	Representation of the process of communication, such as a signal, message, callback, etc.

Table 3: Full overview of programming abstraction actions of schedulers mapped to the reference architecture. Legend: ●/◗/❍ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow

Action	Sub-Action	Schedulers				
		Ku	Sl	Sp	Co	Ai
Provision	Lease / release	●	●	●	●	●
	Scale	●	❍	◗	❍	❍
	Migrate	❍	❍	❍	❍	❍
	Preempt	◗	●	❍	●	❍
	Recover	◗	◗	●	◗	◗
Configure scheduler		●	◗	●	●	●
Manage data	Access input	●	◗	●	●	●
	Access interm.	❍	❍	◗	❍	❍
	Access metadata	❍	❍	❍	❍	❍
	Replicate	❍	❍	●	❍	❍
	Partition	❍	❍	●	❍	❍
	Recover	◗	❍	●	●	❍
Communicate		◗	●	◗	◗	◗

The results indicate that industrial schedulers have several shortcomings. Several actions are under-implemented. There is a very clear pattern, where most schedulers implement three actions: `lease / release`, `configure scheduler`, `access input data`. In most cases, all others are either partially or not implemented. The biggest shortcoming is in `manage data` action and its objects, where most sub-actions and objects are not implemented. Overall, the industrial schedulers examined in our study do not provide data management abstractions to the user. This means that users have less control over the data and, consequently, less chance to optimize performance. For example, if the user has several unordered data

Table 4: Full overview of programming abstraction objects of schedulers mapped to the reference architecture. Legend: ●/◗/❍ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow.

Action	Object	Schedulers				
		Ku	Sl	Sp	Co	Ai
Provision	user resource	●	◗	◗	◗	◗
	event	◗	◗	◗	◗	◗
	sched. resource	●	●	◗	●	●
Configure scheduler	scheduler	◗	◗	●	●	●
	event	◗	◗	❍	❍	❍
	sched. resource	◗	❍	❍	❍	❍
Manage data	user resource	◗	◗	●	◗	●
	event	❍	❍	❍	❍	❍
	sched. resource	❍	◗	❍	◗	❍
Communicate	comm. process	◗	◗	◗	◗	●
	event	◗	◗	◗	❍	◗
	user resource	◗	◗	●	◗	●
	sched. resource	❍	◗	●	❍	❍
	scheduler	❍	◗	❍	❍	❍

items to process, consulting the metadata and obtaining information about the placement and requests load of the storage systems where the data is stored, could optimize how and when the data is processed.

In all other cases, the `communicate` action is partially implemented except in SLURM. Similarly, most communication objects are partial matches. This might imply a lower performance since it does not allow the user to inform the scheduler during runtime about application-level insights, nor vice versa, the scheduler to inform the user about scheduling-level insights. Moreover, partial matches imply that actions and objects are limited to a particular subset and do not allow the user to specify arbitrary inputs. For example, the Condor API only provides communication actions with user jobs, not the scheduler. Therefore, the user can dynamically inform about application-level insights to their jobs but not to the scheduler, reducing the scope of potential performance improvements.

Key Takeaway: Many actions and objects have partial or no matches, meaning their APIs are under-implemented. Consequently, they reduce users' ability and scope to optimize their applications' performance. The main shortcomings are found in `manage data` action and its objects but also in `communicate` actions and their objects to a lesser extent. Sub-actions related to provisioning other than `lease`, such as `scale`, `migrate`, and `recover`, are also not well supported by schedulers.

5 EVALUATING THE PERFORMANCE COST OF SIMPLE SCHEDULING ABSTRACTIONS

In this study, we address the limited programmability of industrial schedulers and highlight the need for greater user programmability to improve user-application performance. We identify under-implemented programming abstractions in scheduler APIs in Section 4. In this section, we design experiments to quantify the performance cost of these missing abstractions. The experiments focus on three specific use cases: *1) reservations*, *2) migration requests*, and *3) metadata access*. We analyze the shortcomings of various industrial schedulers in implementing these abstractions and propose extensions to address them. This answers the question *What is the performance cost of the sacrificed abstractions?* raised in Section 1. A comprehensive overview of these experiments can be found in Table 5, which outlines the API extensions, parameters, traces, and metrics for each use-case.

5.1 Implementation, Input Setup, and Open-Sourcing

Software: The reproducibility of the experiments is ensured through the use of the OpenDC data center discrete event simulator [39], which is deterministic. We performed multiple runs with different seeds of randomness to capture variations in the results. For each experiment run, we calculated the empirical cumulative distribution function (ECDF) to analyze the distribution of the measured metrics. This approach allowed us to assess the behavior and performance of the proposed extensions across different scenarios and obtain comprehensive insights.

Input data: Traces from private and public cloud environments, Azure [13], Google [55], and Bitbrains [49] — a Dutch ICT provider — were selected to provide realistic and diverse workload data for evaluating the proposed extensions. By leveraging real-world traces, our research captures the variability and complexity of cloud workloads, ensuring the relevance and validity of our findings. These traces are open source, and the simulator has parsers for the respective formats. The Azure and Bitbrains traces were used as they were provided, while the first 2.5 days were used from a 30-day Google trace. The characteristics of the different traces are outlined in Table 6.

Simulated environment: The number of machines in the simulated environment are different for different traces and utilization levels. The environments have 35 machines for the Google trace, 102 machines fore the Azure trace, and 1039 machines for the Bitbrains trace when simulating the workloads at 75% utilization. The machines are heterogeneous having 4 to 32 cores depending on the configuration. The precise environment specifications for each experiment are described in topology files located in the experiment's folder in the applications git repository.

5.2 Reservation

Goal: Schedulers utilize resources better if they know when tasks arrive and their resource requirements. We investigate if a scheduler with an API that accepts this additional information performs better for three different traces and by how much.

In the context of scheduling and resource allocation in datacenters, there is a specific category of jobs that are long-running and periodically submitted, which are provisioned into VMs (❶ and ❷ in Figure 4). These jobs exhibit predictable patterns, as they recur regularly and have well-defined resource requirements. Examples of such jobs include data processing pipelines, scientific simulations, and batch processing tasks.

Since their resource requirements and execution patterns are known in advance, schedulers could use this knowledge to allocate resources more efficiently and reduce waiting times. However, in

Table 5: Summary of evaluation experiments.

Name	API extension	Parameters	Fixed parameters	Traces	Metrics
Reservation Section 5.2	User provided start time and resource estimates	Reservation ratio, resource utilization	Scheduling policy (EFT)	Azure, Bitbrains, Google	Waiting time, slowdown
Migration Section 5.3	Container migration via orchestrator callbacks	Migration type, oversubscription	Resource utilization (85%), FIFO policy	Azure, Bitbrains, Google	Execution time, packing efficieny
Metadata access Section 5.4	Use storage subsystem busyness to order tasks	Metadata-aware task reorder policy	Resource utilization (80%)	Google and IBM combined	Buffer size, total time

Table 6: Characteristics of the traces used in the experiments

Workload	VMs/Tasks	Duration [days]	VM duration [days]		CPU cores		CPU capacity [GHz]		Memory [GBs]	
			Mean	σ	Mean	σ	Mean	σ	Mean	σ
Bitbrains	1250	30	28	5	3.27	4.04	2.7	0.16	11.75	32.6
Azure	1829	30	2	6	2.48	2.28	2.5	0.0	5.8	10.16
Google	1000000	2.5	0.0375	0.083	1.0	0.0	1.68	2.08	0.17	0.2

practice, existing schedulers often do not effectively utilize the predictability of these long-running and predictable jobs [54]. As a result, these jobs may be subject to sub-optimal resource allocation and longer waiting times than necessary.

We propose an extension to datacenter schedulers that enhances scheduling long-running and predictable jobs by incorporating reservation programmability. This extension enables schedulers to be aware of these jobs' recurring nature and resource requirements, allowing for more optimized resource allocation and scheduling.

To enable reservations, we extend the system by modifying the `lease` action, including two additional parameters: runtime estimates and a specified provisioning time for future reservations. When a user submits a reservation request, instead of immediately provisioning it, the scheduler adds the request to a reservation queue ❸ alongside other pending reservations. During this time, the scheduler applies algorithmic optimizations to improve future provisioning ❹. In our experiment, we employ a simple Earliest Finish Time (EFT) scheduling policy [52] to optimize the reservation queue by prioritizing tasks with earlier estimated finish times, ensuring that resources are allocated efficiently and effectively. Tasks without reservation are scheduled according to the FIFO policy. Once the specified provisioning time arrives, the scheduler provisions the reserved resources into a VM ❺, fulfilling the user's reservation request. In Listing 2, we provide an example of the extension, showcasing the syntax for reservations.

Listing 2: API for reservations using syntax from Section 3.3, with the extension highlighted in green.

```
Provision:Lease
UserResource<type:app, id:1, runtime:1h>
IN SchedulerResource<type:vm, cores:8,
    cpu-freq:2.4Ghz, memory:32Gb>
WHEN Event<day:11, month:12, year:2023>
```

We take a scheduler that does not implement reservations as our baseline and investigate the effects of incorporating reservation capabilities into this scheduler. We utilize real-world workload traces from Google, Azure, and Bitbrains to evaluate the performance. We sample a fraction (reservation ratio) of the trace to reserve and

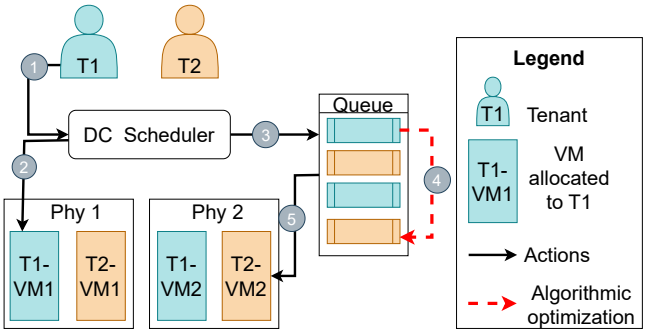


Figure 4: Reservations experiment system model.

assume we know the arrival time (from the original trace) of those tasks in advance.

The experiment configurations involve resource utilization and reservation ratio variations (the proportion of reserved resources compared to the total available resources). The resource utilization levels are set at 75%, 80%, and 85%, and the reservation ratios at 0, 0.5, and 1.0 to observe the impact of reservation programmability. These resource utilizations are common in datacenters with high resource utilization [6]. Metrics collected in the experiment include waiting time (the duration tasks spend in the queue before execution) and slowdown (the decrease in task execution speed).

Figure 5 depicts the Azure trace's waiting time and slowdown under nine different configurations. Slowdown, calculated as the ratio of execution time plus waiting time to execution time, represents the overall task performance. In the Azure trace data, we observe a clear relationship between reservation ratios, waiting times, and slowdowns. Specifically, when the system utilization reaches 85%, the system with reservations has a 43% (35-hour) shorter 50th percentile waiting time than the system without reservations (ratio=0.0 means no reservations). In the same scenario, reservations reduce slowdown by 70% (68 units) compared to not using reservations. However, at a lower utilization of 80%, there is an increase in waiting time of 2.5 hours (50th percentile) and a 12-unit (60th percentile) increase in slowdowns. In the other traces examined, there is no significant impact on the waiting times and

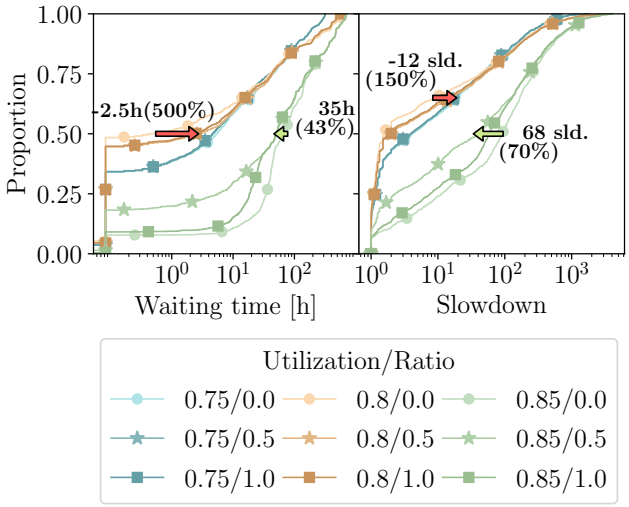


Figure 5: ECDFs of waiting time and slowdown per task of the Azure trace using the reservation extension. We evaluate the system at different utilization levels and with a different fraction of the trace being reserved in advance (ratio). Ratio 0.0 implies no reservations.

slowdowns with varying reservation ratios. This could be due to workload characteristics, resource utilization levels, or the configuration of the scheduling system. Further investigation is needed to determine the underlying reasons for the lack of impact.

The results are not as promising for the Google and Bitbrains traces. The Azure trace differs from the other traces as it has a multi-hour task duration. The Google trace has short tasks lasting seconds, and the Bitbrains trace has long jobs lasting weeks. The full analysis for the other traces is available in the technical report. **Key Takeaway:** Reservations reduce slowdown by as much as 70% for the Azure trace, but not as much for the other traces. The results are dependent on the durations of the tasks in the trace.

5.3 Migration

Goal: We investigate if offloading migration, to mitigate interference, to container orchestrators running on top of VMs leased from a datacenter scheduler is better than the datacenter scheduler itself performing VM migration. We investigate this for three traces.

Datacenter operators oversubscribe their machines as tenants often do not utilize all the allocated resources. Oversubscription means allocating more resources to tenants than there are physically available. Oversubscription leads to interference between tenants if tenants allocated to the same physical machine fully utilize their allocated resources. In such cases, the datacenter operator can migrate one or more tenants to less utilized physical machines to reduce interference.

Migration has a cost proportional to the size of the VM migrated [16, 37]. Therefore it is efficient to migrate only part of a VM if possible. Nowadays, tenants use container orchestrators (K1 in Figure 6), such as Kubernetes, making partial migration possible. The orchestrator requests resources from the datacenter

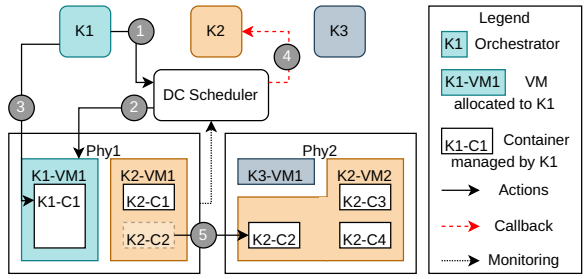


Figure 6: Migrations experiment system model.

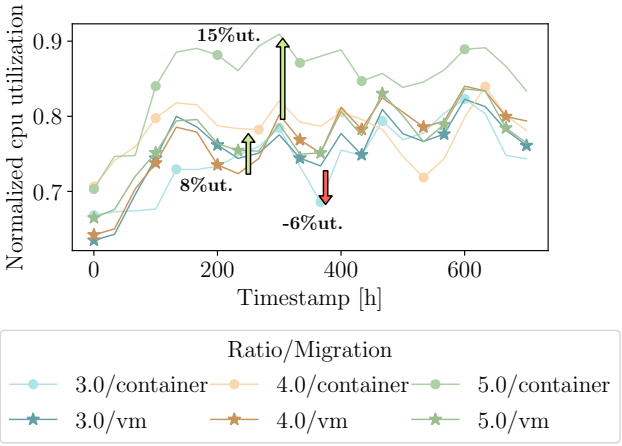


Figure 7: Task packing efficiency of the Azure trace using different migration techniques. Each line represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

scheduler ❶. The DC scheduler allocates resources in the form of VMs ❷. The orchestrator then starts application containers inside the VM ❸.

We propose an extension to datacenter schedulers that enables partial migration by making them aware of the tenants' orchestrators. The key to enabling partial migration is to enable bidirectional communication between the datacenter scheduler and the orchestrator. The orchestrator registers a remote callback with the datacenter scheduler before it requests any VM allocations. The datacenter scheduler uses this callback (❹ in Figure 6) to request the orchestrator to migrate ❺ some containers when its monitoring detects interference. In Listing 3, we provide an example of the extension, showcasing the syntax for migrations.

Listing 3: API for migration using syntax from Section 3.3, with the extension highlighted in green.

```
Communicate
CommunicationProcess<type:callback,
                     url:orchestratorhost/callback>
IN UserResource<type:app, id:1>
WHEN Event<interference:10%>
```

As a baseline, we take a scheduler implementing VM migrations and determine the impact of adding container migrations to that

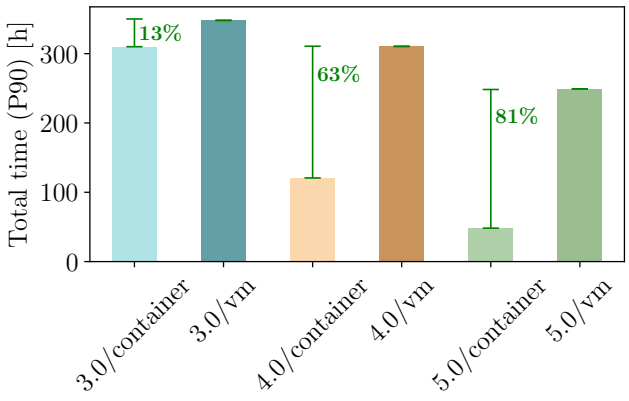


Figure 8: 90th percentile (P90) total runtime per task of the Azure trace using different migration techniques. Each bar represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

scheduler. We use three real-world workload traces from Google, Azure, and Bitbrains for our evaluation.

For each trace, we evaluate the impact of migrations at three oversubscription ratios: 3, 4, and 5. An oversubscription ratio of 3 means that each physical CPU was fully available to three tenants. Oversubscription ratios ranging from 3 to 16 are common in datacenters whose users have low utilization [34, 43]. We model the cost of migration as the time it takes to migrate the RAM used by the VM/container at a conservative rate of 512Mbps. The RAM based cost model and the migration bandwidth are supported by existing literature [37]. Our hypothesis is that migrating a container takes less time than migrating a VM running multiple containers.

We simulate 5 Kubernetes clusters simultaneously using the datacenter. We configure the datacenter topology such that the traces run at 85% average utilization. The metrics we use are total workload execution time and packing efficiency. We calculate packing efficiency by summing the CPU utilization of each virtual machine (VM) and dividing it by the total number of VMs. This metric provides insights into how effectively the resources allocated to the VMs were utilized. A higher packing efficiency indicates better utilization of resources, while a lower value suggests potential inefficiencies or underutilization. By analyzing packing efficiency, we can assess the effectiveness of the scheduling mechanisms in optimizing resource allocation and maximizing overall system performance.

Figure 7 and 8 depict the packing efficiency and the total execution time (90th percentile) of the Azure trace under six different configurations, respectively. In the Azure trace, the highest oversubscription ratio of 5.0 achieved a remarkable 15% improvement in packing compared to configurations without the API extension. Additionally, using the API led to improved performance in terms of total time per task. For example, with the highest oversubscription ratio of 5.0, the 90th percentile (P90) of total time per task in the Azure trace were reduced by 81% when container-level migrations were employed.

In the remaining Google and Bitbrains traces, using the API resulted in shorter total time per task, indicating higher performance. The 99th percentile total time per task in the Google trace showed a reduction of 73% (4.4 hours) with the highest oversubscription ratio of 5.0. However, it is important to note that not all configurations yield better performance with container-level migrations. However, in the Bitbrains trace, no significant improvement in performance is observed. The results indicate the minimal impact of container-level migrations on performance in this particular trace.

Key Takeaway: Offloading migration to container orchestrators benefited the Azure and the Google traces, not the Bitbrains trace. The Bitbrains trace differs from other traces as it has an extremely long task duration, with tasks running for weeks.

5.4 Metadata access

Goal: We investigate if providing datacenter schedulers access to additional information about task data accesses and storage subsystem busyness has a performance impact. We analyze the impact of a trace from IBM object storage [21] combined with the compute trace from Google.

Datacenters offer object storage services that enable users to store and retrieve data efficiently. Services like AWS S3 provide a scalable and reliable solution for storing large amounts of data. In the context of data analysis workloads, users often deploy applications that require accessing multiple objects from the storage (❶ and ❷ in Figure 9). These workloads (e.g.: data analytics [4], ML [19]) are often "bag of tasks" where tasks are executed independently and the objects to read are known in advance. Such workloads benefit from reordering their storage access based on the prevailing resource utilization at the time of access.

Without access to fine-grained information about object placement and load levels, users cannot optimize their data retrieval process. As a result, the workload takes longer to complete. The inefficiencies in object access lead to increased latency, reduced throughput, and decreased overall system performance [40, 57].

We propose an extension that empowers users to access object metadata to address this limitation. This extension allows users to make informed decisions regarding the order in which they retrieve data items. By introducing the `accessMetadata` action in the scheduler's programming model, users can query the metadata for specific object IDs and obtain estimates of retrieval times. The scheduler retrieves this information by monitoring the storage servers (❸). This capability enables users to strategically postpone the retrieval of objects from congested storage servers, allowing them to process those objects later when congestion levels have subsided. In Listing 4, we provide an example of the extension, showcasing the syntax for metadata access.

Listing 4: API for metadata access using syntax from Section 3.3, with the extension highlighted in green.

```
ManageData : AccessMetadata
UserResource<type:object, id:2>
IN SchedulerResource<type: object-storage>
WHEN Event<datetime: now>
```

We aim to determine how much performance existing schedulers are losing out on by not implementing metadata access. As a baseline, we take a scheduler providing an object storage service and

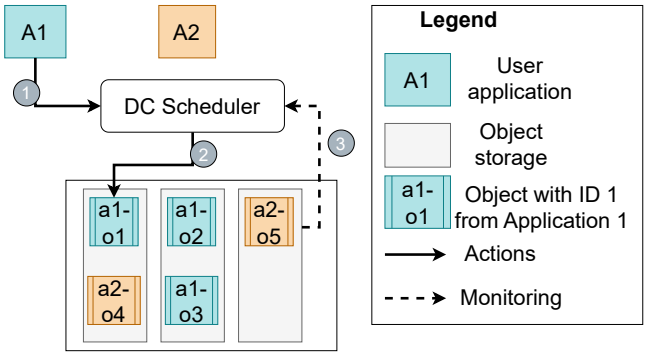


Figure 9: Metadata access experiment system model.

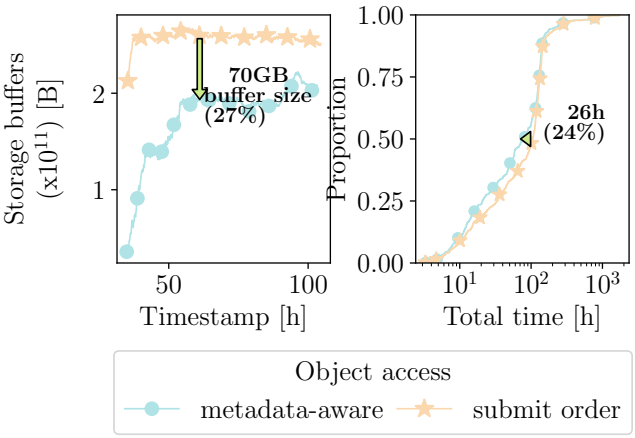


Figure 10: Comparison of buffer sizes in the object storage service (left) and ECDF analysis of total execution times per workflow (right) between the configuration with and without the metadata access API. This uses the Google Compute trace combined with the IBM object storage trace.

determine the impact of adding metadata access to that scheduler. Our evaluation is based on a combination of real-world workload traces, specifically a trace from Google and an IBM object storage trace [21]. We chose to focus on Google trace for this experiment due to its availability of detailed information about workflows.

We use the interarrival time, duration, and resource usage of tasks from the Google trace. For each task, we associate an object identifier from the IBM trace. We read identifiers from the IBM trace sequentially. This maintains the popularity distribution of object identifiers and their temporal locality. We assume each task reads from distributed storage at 1Gbps [8]. We simulate a 10 node distributed object storage system, with objects accessed by their identifiers.

We analyze the impact by activating and deactivating metadata access while maintaining a fixed workload trace and storage service utilization. The workload trace utilization is set at 80%. We capture two key metrics to evaluate the system's performance: buffer sizes of the object storage service and total workflow times. The buffer sizes provide insights into the waiting line and load balancing across servers. Smaller buffer sizes indicate lower system load and more efficient workload distribution across servers. Additionally, we measure the total time for each workflow, which encompasses both the waiting time and the execution time.

Figure 10 displays the normalized buffer sizes and total execution times of the trace. The results demonstrate that activating the metadata access API leads to substantially reduced buffer sizes, approximately 27% (70 GB), within the object storage service, resulting in improved performance. Furthermore, metadata-aware workflow execution substantially reduces total time per workflow, with a notable 24% (26-hour) decrease in the median value. These findings emphasize the critical role of metadata access in optimizing object retrievals and enhancing overall performance.

Key Takeaway: The significant performance improvements observed in reduced buffer sizes and shorter execution times highlight the value of exposing storage metadata using an API.

6 THREATS TO VALIDITY

The reference architecture we proposed has two main limitations.

First, the reference architecture design is limited to the objects we define. In our reference architecture, we identify only five distinct objects and do not specify sub-objects for each. For example, our `Scheduler Resource` object does not differentiate between an API that offers VMs or Edge mobile devices. While this is a limitation, we have deliberately chosen to keep our objects at a high level of abstraction to future-proof our architecture. As the types of resources available for scheduling are constantly changing, we believe it is more important to differentiate objects by what they represent in the highest level of abstraction than by their specific content.

However, to fully leverage the power of our reference architecture, it will be necessary to build more specific models that differentiate between schedulers with different requirements. For example, Spark-like schedulers have different scheduling requirements than Kubernetes-like schedulers. These models must differentiate between objects based on their specific content rather than just their highest level of abstraction.

Second, the simulation scenarios we use and the simulator itself are not a replacement for real-world systems. However, the simulator we use, OpenDC, has been validated for VM and container scheduling for the Bitbrains and Azure traces [39]. The storage part of the simulator and the Google trace have not yet been validated. But we do use realistic models for migration [37] and storage accesses [8]. These models based on measurements from real systems ensure that our results are indicative of real-world performance.

7 RELATED WORK

Schopf's multi-stage model of the grid scheduling process [30], the Global Grid Forum [26], and the datacenter scheduler reference architecture [5] offer conceptual models of the internal workings of schedulers. Our work complements these models by specifically addressing the external-facing aspects of scheduling, the programming interface.

Conceptual models of APIs have been proposed for specific computing environments, such as grid computing and cloud computing. Foster et al. presented a reference architecture for grid computing [22], and the National Institute of Standards and Technology

(NIST) introduced models for cloud computing [36]. While these models provide valuable guidance for designing APIs in their respective domains, they do not deal with the concrete API needs of schedulers like Spark and Kubernetes, which have unique characteristics and requirements.

Efforts have been made to develop schedulers that combine multiple scheduling abstractions into a single system, such as Ghost [29] and ESCHER [7]. Ghost delegates OS kernel scheduling decisions to users, granting them greater control over the scheduling process. ESCHER allows users to express arbitrary scheduling constraints as resource requirements, enabling fine-grained control over the scheduling process. Apache Beam [23] and CWL [14] allow users to specify a workflow and run it on multiple resource managers. But they do not allow control over the scheduling mechanism apart from simple labels.

8 CONCLUSION

In this work, we designed a reference architecture for datacenter scheduler APIs (Section 3). Our reference architecture covers APIs implemented in 5 industrial schedulers (Kubernetes, SLURM, Spark, Condor, Airflow) and 15 academic schedulers. We use the reference architecture to identify abstraction not implemented or under-implemented in the five industrial schedulers (Section 4). We find that the industrial schedulers do not implement abstractions for data management, task migration, and autoscaling.

We evaluate the performance impact of missing abstractions related to resource reservation, container migration, and storage metadata access in Section 5. We find a 27% improvement in resource usage and a 24% reduction in median workflow runtime when implementing metadata access, a 15% increase in utilization and an 81% improvement in total execution time per task (90th percentile) for container migrations, and a 43% reduction in waiting times (50th percentile) for reservations.

For future work, we intend to provide a toolkit for users to experiment with different designs using the OpenDC simulator. We also plan to validate our simulations beyond the basic validation with VMs, including validation with containers and storage services.

All our data and software artifacts are publicly available at https://github.com/atlarge-research/quantifying-api-design. The repository has been archived using Zenodo at: https://zenodo.org/doi/10.5281/zenodo.10605424

ACKNOWLEDGEMENTS

This work is supported by EU Horizon Graph Massivizer and EU MSCA CloudStars projects. This research is partly supported by a National Growth Fund through the Dutch 6G flagship project"Future Network Services".

REFERENCES

[1] 2023. Apache Airflow. https://github.com/apache/airflow.
[2] 2023. Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[3] 2023. Kubernetes. https://github.com/kubernetes/kubernetes.
[4] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 267–280. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/ananthanarayanan
[5] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. 2018. A reference architecture for datacenter scheduling: design, validation, and experiments. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 478–492.
[6] Noman Bashir, Nan Deng, Krzysztof Rzadca, David E. Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 556–573. https://doi.org/10.1145/3447786.3456259
[7] Romil Bhardwaj, Alexey Tumanov, Stephanie Wang, Richard Liaw, Philipp Moritz, Robert Nishihara, and Ion Stoica. 2022. ESCHER: expressive scheduling with ephemeral resources. In *Proceedings of the 13th Symposium on Cloud Computing*. 47–62.
[8] Haoqiong Bian and Anastasia Ailamaki. 2022. Pixels: An Efficient Column Store for Cloud Data Lakes. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 3078–3090. https://doi.org/10.1109/ICDE53745.2022.00276
[9] Luiz F Bittencourt, Javier Diaz-Montes, Rajkumar Buyya, Omer F Rana, and Manish Parashar. 2017. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing* 4, 2 (2017), 26–35.
[10] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, omega, and kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.
[11] Wo L Chang, David Boyd, Orit Levin, et al. 2019. NIST Big Data Interoperability Framework: Volume 6, Reference Architecture. (2019).
[12] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. 2000. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications* 23, 3 (2000), 187–200.
[13] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 153–167. https://doi.org/10.1145/3132747.3132772
[14] Michael R Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojša Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilović, Carole Goble, et al. 2022. Methods included: standardizing computational reuse and portability with the common workflow language. *Commun. ACM* 65, 6 (2022), 54–63.
[15] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14.
[16] Waltenegus Dargie. 2014. Estimation of the cost of VM migration. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–8.
[17] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.
[18] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.
[19] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant prefetching for distributed machine learning I/O. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 92. https://doi.org/10.1145/3458817.3476181
[20] Tomi Dufva and Mikko Dufva. 2019. Grasping the future of the digital society. *Futures* 107 (2019), 17–28.
[21] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen I. Kat. 2020. It's Time to Revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*, Anirudh Badam and Vijay Chidambaram (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/eytan
[22] Ian Foster, Carl Kesselman, and Steven Tuecke. 2001. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications* 15, 3 (2001), 200–222.
[23] Apache Foundation. [n. d.]. Apache Beam. https://beam.apache.org/
[24] F Gens. 2014. Worldwide and Regional Public IT Cloud Services.
[25] Robert Grandl, Arjun Singhvi, Raajay Viswanathan, and Aditya Akella. 2021. Whiz:{Data-Driven} Analytics Execution. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 407–423.
[26] Christian Grimme, Joachim Lepping, Alexander Papaspyrou, Philipp Wieder, Ramin Yahyapour, Ariel Oleksiak, Oliver Wäldrich, and Wolfgang Ziegler. 2008. Towards a standards-based grid scheduling architecture. In *Grid Computing*. Springer, 147–158.
[27] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX*

Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020. USENIX Association, 845–861. https://www.usenix.org/conference/osdi20/presentation/hadary

[28] Michael Haenlein and Andreas Kaplan. 2019. A brief history of artificial intelligence: On the past, present, and future of artificial intelligence. *California management review* 61, 4 (2019), 5–14.

[29] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 588–604. https://doi.org/10.1145/3477132.3483542

[30] M Schopf Jennifer. 2004. Ten Actions When Grid Scheduling: The User as a Grid Scheduler. *Grid Resource Management: State of the Art and Future Trends, Norwell, MA, USA, Kluwer Academic Publishers* (2004), 15–24.

[31] Fredy Juarez, Jorge Ejarque, and Rosa M Badia. 2018. Dynamic energy-aware scheduling for parallel task-based application in cloud computing. *Future Generation Computer Systems* 78 (2018), 257–271.

[32] Nakku Kim, Jungwook Cho, and Euiseong Seo. 2014. Energy-credit scheduler: an energy-aware virtual machine scheduler for cloud systems. *Future Generation Computer Systems* 32 (2014), 128–137.

[33] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology* 51, 1 (2009), 7–15.

[34] Ken Leoni. [n. d.]. Heroix: Maximizing VMware Performance and CPU Utilization. https://www.heroix.com/blog/vmware-vcpu-over-allocation/

[35] Don Lipari. 2012. The slurm scheduler design. In *SLURM User Group Meeting, Oct*, Vol. 9. 52.

[36] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, Dawn Leaf, et al. 2011. NIST cloud computing reference architecture. *NIST special publication* 500, 2011 (2011), 1–28.

[37] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. 2011. Performance and energy modeling for live migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*. 171–182.

[38] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. 2015. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. *Future Generation Computer Systems* 48 (2015), 1–18.

[39] Fabian Mastenbroek, Georgios Andreadis, Soufiane Jounaid, Wenchen Lai, Jacob Burley, Jaro Bosch, Erwin Van Eyk, Laurens Versluis, Vincent Van Beek, and Alexandru Iosup. 2021. OpenDC 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 455–464.

[40] Somnath Mazumdar, Daniel Seybold, Kyriakos Kritikos, and Yiannis Verginadis. 2019. A survey on data storage and placement methodologies for cloud-big data ecosystem. *Journal of Big Data* 6, 1 (2019), 1–37.

[41] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 69–84.

[42] Paul Owen. [n. d.]. Slurm Used on the Fastest of the TOP500 Supercomputers. https://www.prweb.com/releases/2012/11/prweb10149109.htm. Accessed: 2023-03-18.

[43] Platform9. [n. d.]. Resource Overcommitment. https://platform9.com/docs/openstack/infrastructure-resource-overcommitment

[44] Kavitha Ranganathan and Ian Foster. 2002. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 352–358.

[45] Tech. Rep. 2022. *2022 Leadership Vision for Infrastructure and Operations*. Gartner.

[46] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. 199–212.

[47] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek (Eds.). ACM, 351–364. https://doi.org/10.1145/2465351.2465386

[48] Siqi Shen, Alexandru Iosup, Assaf Israel, Walfredo Cirne, Danny Raz, and Dick Epema. 2015. An availability-on-demand mechanism for datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 495–504.

[49] Siqi Shen, Vincent van Beek, and Alexandru Iosup. 2015. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. IEEE Computer Society, 465–474. https://doi.org/10.1109/CCGrid.2015.60

[50] Karnam Sreenu and M Sreelatha. 2019. W-Scheduler: whale optimization for task scheduling in cloud computing. *Cluster Computing* 22, 1 (2019), 1087–1098.

[51] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurr. Pract. Exp.* 17, 2-4 (2005), 323–356. https://doi.org/10.1002/cpe.938

[52] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.

[53] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. 2012. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the third ACM Symposium on Cloud Computing*. 1–7.

[54] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[55] John Wilkes. 2011. More Google cluster data. Google research blog. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html..

[56] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

[57] Yanlong Zhai, Jude Tchaye-Kondi, Kwei-Jay Lin, Liehuang Zhu, Wenjun Tao, Xiaojiang Du, and Mohsen Guizani. 2021. Hadoop perfect file: A fast and memory-efficient metadata access archive file to face small files problem in hdfs. *J. Parallel and Distrib. Comput.* 156 (2021), 119–130.

[58] Chao Zheng, Ben Tovar, and Douglas Thain. 2017. Deploying High Throughput Scientific Workflows on Container Schedulers with Makeflow and Mesos. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. IEEE Computer Society / ACM, 130–139. https://doi.org/10.1109/CCGRID.2017.9

[59] Qiang Zheng, Kan Zheng, Haijun Zhang, and Victor CM Leung. 2016. Delay-optimal virtualized radio resource scheduling in software-defined vehicular networks via stochastic learning. *IEEE Transactions on Vehicular Technology* 65, 10 (2016), 7857–7867.

Accelerating ML Workloads using GPU Tensor Cores: The Good, the Bad, and the Ugly

Bagus Hanindhito
hanindhito@bagus.my.id
The University of Texas at Austin
Austin, Texas, USA

Lizy K. John
ljohn@ece.utexas.edu
The University of Texas at Austin
Austin, Texas, USA

ABSTRACT

Machine Learning (ML) workloads generally contain a significant amount of matrix computations; hence, hardware accelerators for ML have been incorporating support for matrix accelerators. With the popularity of GPUs as hardware accelerators for ML, specialized matrix accelerators are embedded into GPUs (e.g., Tensor Cores on NVIDIA GPUs) to significantly improve the performance and energy efficiency of ML workloads. NVIDIA Tensor Cores and other matrix accelerators have been designed to support General Matrix-Matrix Multiplication (GEMM) for many data types. While previous research has demonstrated impressive performance gains with Tensor Cores, they primarily focused on Convolutional Neural Networks (CNNs).

This paper explores Tensor Cores' performance on various workloads, including Graph Convolutional Networks (GCNs), on NVIDIA H100 and A100 GPUs. In our experiments with NVIDIA GPUs, CNNs can achieve 1.91× (TF32) and 2.42× (FP16) end-to-end performance improvements with the use of Tensor Cores, whereas GCNs struggle to surpass a 1.03× (FP16) boost. Some implementations even experience slowdowns despite software transformation. Additionally, we explore the potential of Tensor Cores in non-GEMM-like kernels, providing insights into how software techniques can map diverse computation patterns onto Tensor Cores. Our investigation encompasses several kernels and end-to-end applications, aiming to comprehend the nuanced performance impact of Tensor Cores. Furthermore, we are among the first to present third-party evaluations of H100 GPU performance over the prior A100 GPU.

CCS CONCEPTS

• **General and reference** → **Performance**; **Measurement**; **Evaluation**; **Experimentation**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Machine Learning; Matrix Accelerators; Performance Evaluation; Workload Characterization; Measurement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3653835

ACM Reference Format:
Bagus Hanindhito and Lizy K. John. 2024. Accelerating ML Workloads using GPU Tensor Cores: The Good, the Bad, and the Ugly. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3653835

1 INTRODUCTION

At the heart of Artificial Intelligence (AI) and Machine Learning (ML), General Matrix-Matrix Multiplications (GEMMs) are the most important building blocks for many applications [4, 26, 81]. In 2017, with the launch of Volta architecture [45], NVIDIA introduced Tensor Cores in their GPUs to accelerate GEMM. Tensor Core provides significant performance boost and energy efficiency when performing GEMM operations, and is accessible either through low-level assembly or various CUDA libraries [38]. Other manufacturers followed by integrating matrix accelerators into their GPUs years later [1, 24]. Recently developed hardware that targets AI and ML, including FPGA and ASIC, also has matrix accelerators, such as in Xilinx Versal FPGA [16] and Google TPU ASIC [25].

In this paper, the performance benefits of Tensor Cores are investigated across multiple workloads. Prior works on Tensor Cores evaluate Convolutional Neural Networks (CNN) [57, 76] and GEMM [14, 17]. However, the benefits of Tensor Cores in Graph Convolutional Networks (GCN) [29], which is an important emerging ML workload, have not been explored. We analyze the performance of four configurations of the GCN model and several kernels including element-wise operations. Another contribution of this paper is the measurement-based evaluation of ML acceleration using the NVIDIA H100 GPU. Apart from NVIDIA publications, there have been very few third-party works evaluating H100 GPUs. This is also one of the earliest third-party papers to measure and analyze the performance of H100 compared to its predecessor, A100. While performance evaluation of H100 appears in prior work [7], they do not present Tensor Core performance.

The objectives of this study are the following:

- Investigate the performance of the CNN and GCN, both with and without Tensor Cores, across two generations of NVIDIA GPUs, A100 [48] and H100 [49], based on hardware measurement.
- Provide third-party performance evaluation of NVIDIA H100 GPU compared to the previous generation GPU, NVIDIA A100.
- Conduct roofline analysis of the workloads to understand their characteristics and correlation with Tensor Cores performance.
- Develop GEMM-like and non-GEMM-like microbenchmark kernels to understand the performance patterns of Tensor Cores.
- Analyze the floating-point instruction mix of workloads and shed light on the types of lower precision instructions utilized, the functional units where they are being executed (e.g., CUDA

Cores, Tensor Cores), etc. across different networks and training configurations (e.g., full-precision, mixed-precision).
- Investigate the impact of new data types, such as TF32 [8].
- Investigate whether code optimizations like reshaping and padding can make non-GEMM kernels utilize Tensor Cores (eg: Implicit GEMV vs. Reshaped GEMV for FIR)

The major insights from this study are the following:
- Tensor Cores provides 1.3× to 2.9× improvements in CNN whereas only 1.03× in GCN. Among kernels, GEMM, GEMV, and Conv2D get the benefits while Element-wise and FIR fail to get any improvements in spite of transformations.
- Four different CNNs yield an average of 1.93× improvement on H100 versus the previous A100 GPU. Among the four GCN configurations experimented, two yield an impressive 8× improvement on H100 compared to A100, whereas two of the GCN configurations provide nearly no improvements.
- GCNs have 10× lower arithmetic intensity compared to CNNs, and benefits from Tensor Cores are difficult to obtain.
- There are performance anomalies while using different CUDA versions. For instance, the newest CUDA libraries gave improved performance for many workloads, however, for some of the GCNs, they yielded poorer performance than the older CUDA version.
- Non-GEMM-like kernels struggle to get any performance improvements from Tensor Cores, even with data transformations. Reshaped FIR can use batching in order to reduce performance overheads, whereas naive FIR is not even supported and cannot run on Tensor Cores.

2 BACKGROUND AND PRIOR WORK

2.1 Tensor Cores

Starting from Volta architecture (2017), NVIDIA GPUs contain CUDA Cores and Tensor Cores as illustrated in Figure 1. CUDA Cores are the default (traditional) compute units in GPUs, while Tensor Cores were later added specifically for accelerating matrix multiplications, which are abundant in many machine learning (ML) workloads [4, 26, 81]. With libraries provided by NVIDIA, Tensor Cores quickly became the workhorse for accelerating ML workloads as popular machine learning frameworks, such as PyTorch and TensorFlow, support Tensor Cores.

2.1.1 Architectural Overview. Figure 2 gives a high-level illustration of the **matrix-multiply-accumulate (MMA)** operations performed by Tensor Cores on two 4×4 matrices to produce a 4×4 matrix. Essentially, Tensor Cores perform the arithmetic expression $D = A \times B + C$ where A, B, C, D are matrices. Larger dimension matrices are possible using larger Tensor Cores instruction size and hierarchical matrix multiplication [28].

The NVIDIA Tesla V100 with Volta architecture [45] contains 640 first-generation Tensor Cores across 80 SMs[1]. The Tensor Cores in each SM can deliver 1024 FLOPs per cycle, resulting in up to 120 TFLOPs/s FP16 performance [9]. Only half-precision matrix multiplication is supported in this generation. Thus, the A and B matrices in Figure 2 are in FP16, while the resulting product matrix

[1]SM stands for Streaming Multiprocessor, which contains a collection of SIMD Units referred to as CUDA Cores (e.g., FP64, FP32, INT32), instruction schedulers, registers, shared memory, L1 cache, and texture cache (Figure 1). GPUs usually have multiple numbers of SM to achieve even higher parallelism.

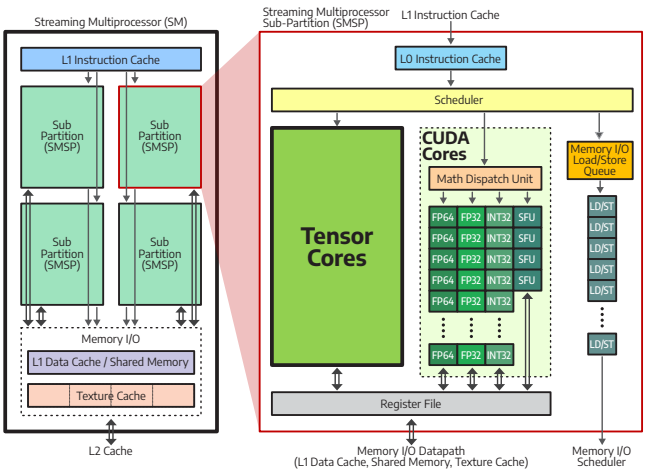


Figure 1: CUDA Cores are the default compute units while Tensor Cores are additions to accelerate matrix multiplications in GPUs

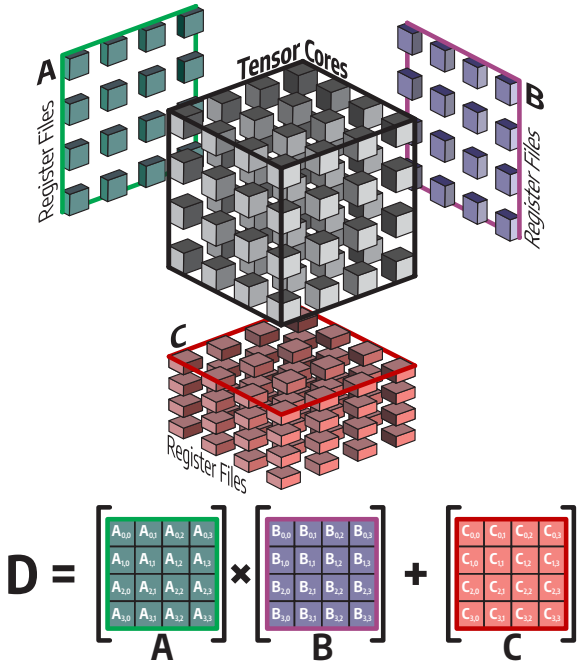


Figure 2: Tensor Cores compute $D = A \times B + C$

Table 1: Tensor Cores Evolution and Supported Precision

Generation	Architecture	Product Name	Specification			Precision Support											
						Tensor Cores							CUDA Cores				
			#SM	#CC (FP32)	#TC	FP64	TF32	FP16	BF16	FP8	INT8	INT4	INT1	FP64	FP32	FP16	BF16	INT32	INT8
1	Volta [45]	V100S	80	5120	640	-	-	✔	-	-	-	-	-	✔	✔	✔	-	✔	✔
2	Turing [46]	RTX 6000	72	4608	576	-	-	✔	-	-	✔	✔	✔	●	✔	✔	-	✔	✔
3	Ampere [48]	A100	108	6912	432	✔	✔	✔	✔	-	✔	✔	✔	✔	✔	✔	✔	✔	✔
4	Hopper [49]	H100	132	16896	528	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
4	Ada [54]	L40S	142	18176	568	-	✔	✔	✔	✔	✔	✔	✔	●	✔	✔	✔	✔	✔

✔: full-support; ●: support with reduced performance; -: not supported.

can be in either FP16 or FP32. The subsequent version of Tensor Cores supports more data types as given in Table 1.

The second generation of Tensor Cores was introduced in 2019 with Turing architecture [46] focusing on accelerating the quantized ML inference workload. It supports new data types INT8,

INT4, and INT1, which are specifically useful for ML inference workloads that can tolerate lower precision with minimum impact on model accuracy [34] as well as binary neural networks [32]. Third-generation Tensor Cores, launched with Ampere architecture [48], support new data types such as BFloat16 (BF16) [74] and TensorFloat32 (TF32) [8] with additional support for accelerating sparse matrix operations [68]. Moreover, new FP64 support opens new possibilities for Tensor Cores to be used in HPC and scientific applications [17]. The fourth-generation Tensor Cores, introduced in the Hopper architecture [49] in 2022, double the throughput per SM per cycle compared to its predecessor for all data formats [7]. A new quarter precision data type (FP8), which supports both `e4m3` (4 exponent bits, 3 mantissa bits) for more accuracy and `e5m2` (5 exponent bits, 2 mantissa bits) for more dynamic range [42], is useful for large language models. More FP64 shapes and new warp-group level Tensor Cores instructions are introduced, supporting larger instruction sizes. Fifth-generation Tensor Cores, introduced in the Blackwell architecture in 2024, support FP6 and FP4 data types.

2.2 Mixed Precision Training

Mixed precision training [41] can help reduce the amount of memory required to train the model, ease the bandwidth requirement (e.g., off-chip memory and inter-node network bandwidth), and lower the computational power needed. It uses multiple precision formats; lower precision (e.g., FP16) is used in most of the network during the training while single precision (e.g., FP32) is used in the critical parts of the network (e.g., accumulation of gradients after each optimizer step) to maintain numerical stability and accuracy. Some of the hardware has FP32 units that can execute FP16 twice the rate of FP32, such as NVIDIA Pascal architecture [44], which improves training performance. With many advantages offered by mixed precision training, vendors try to find even more efficient data formats to train AI and ML models without sacrificing the performance of the models. Google introduced BF16, which retains the dynamic range of FP32 in 16-bit format [74], while NVIDIA introduced TF32, which retains the dynamic range of FP32 while keeping the accuracy of FP16 in 19-bit format [8].

2.3 Prior Evaluation of Tensor Cores

Since its introduction in 2017, Tensor Cores have been investigated in academic and industry research. Tensor Cores improve the performance of ML workloads by using mixed precision while maintaining model accuracy [39]. Memory-bound operations often see around two times speed-up thanks to the reduced data size (e.g., FP16). In contrast, compute-bound operations benefit from Tensor Cores depending on their arithmetic intensity [40]. Prior studies show the use of Tensor Cores on NVIDIA V100 GPU gives more than 2× speed-up in training for ResNet50 [57], GNMT [51], Inception v3, and Vgg16 models [76]. In addition, quantized inference gets up to 5× higher throughput and lower latency by using Tensor Cores inside NVIDIA Turing GPU [46] across many models, including ResNet50 v2, MobileNet v2, and SSD MobileNet v2 [18, 73]. Other models, including UNet Industrial Defect Segmentation, show a slight performance drop [58]. Prior works also include arithmetic accuracy studies for GEMM [2, 34, 59], scientific computation using double precision on third and fourth-generation Tensor Cores [14, 17], and mapping GEMM-like application into Tensor Cores [10], which include Fast Fourier Transform [13], reduction [43], scientific simulations [11], and linear system solver [19]. However, prior works mostly focused on convolution and GEMM-like workloads. Workloads such as GCNs and non-GEMM-like applications have not been studied. Finally, a study is done to characterize Tensor Cores latency, throughput, and numerical behavior to get the low-level detail of Tensor Cores [67]. However, it does not show how applications behave in different generations of Tensor Cores.

2.4 Programming Tensor Cores

With CUDA, programmers can develop applications that target NVIDIA GPUs using high-level languages, such as C, C++, Fortran, and Python. The high-level code is then compiled by a compiler (e.g., `nvcc`) to an intermediate assembly language called PTX (Parallel Thread eXecution) [27], whose ISA is openly documented [56]. The PTX instructions are then compiled to device-specific Streaming Assembly (SASS) either through ahead-of-time compilation using PTX assembler (e.g., `ptxas`) or just-in-time compilation by the display driver [66].

While developing applications that only utilize CUDA Cores can be done more easily using the high-level language of choice, developing applications that specifically target Tensor Cores to achieve higher performance is a different story. The instructions that run on Tensor Cores perform **matrix multiplication and accumulation (MMA)** [38] on a predefined dimension called instruction size. The programming model of Tensor Cores constructs this operation at the warp[2] level, which is different than the regular CUDA model which constructs the operation at the thread level [39]. Multiple sizes and operands are supported using `MMA`, including half-precision (`hmma`), integer (`imma`), binary (`bmma`), and double-precision (`dmma`). These instructions is accessed via PTX through inline assembly.

Since there is a limited number of instruction sizes for Tensor Cores, tiling must be done for operations that involve arbitrary dimensions of matrices. This consists of dividing the large matrices hierarchically at the grid[3] level into multiple thread block tiles, and further decomposing them

into warp tiles with multiple thread tiles utilizing all memory types in the hierarchy (e.g., global memory, shared memory, register files) [28]. Moreover, fulfilling the data layout and memory alignment requirements of Tensor Cores may not always be straightforward, especially for applications that have irregular data structures and computation patterns [15, 61, 75]. It is also challenging to handle sub-byte operations, such as INT4 and INT1 [6]. Therefore, programming Tensor Cores is an uphill task.

To overcome this issue, NVIDIA provides libraries that implement various functions that target Tensor Cores [3], and thus instead of having to write in-line assembly for PTX, developers can call the provided functions from their applications. Among the libraries include cuBLAS, cuSPARSE, cuTENSOR, cuDNN, and CUTLASS. CUTLASS is the only open-source library from the previous list that provides C++ template for developing high-performance

[2]Warp is a group of 32 threads concurrently executing the same instructions in a lock-step fashion. A collection of warp constitutes a thread block, which runs on an SM. The scheduler inside the SM will choose which warp runs based on the readiness of operands and perform context-switching across warps to hide memory access latency.
[3]Grid is a collection of thread blocks executing a GPU kernel.

Table 2: Hardware Configuration

Platform	DGX-A100	XE9680
GPU		
Model	NVIDIA A100	NVIDIA H100
Form Factor	SXM4	SXM5
Memory Size & Type	40 GB HBM2	80 GB HBM3
Memory Bandwidth	1,555 GBps	3,350 GBps
# CUDA/Tensor Cores	6912/432	16896/528
FP32 on CUDA Cores	19.5 TFLOP/s	67 TFLOP/s
FP16 on CUDA Cores	39 TFLOP/s	133.8 TFLOP/s
TF32 on Tensor Cores[1]	156/312 TFLOP/s	494.7/989.4 TFLOP/s
FP16 on Tensor Cores[1]	312/624 TFLOP/s	989.4/1978.9 TFLOP/s
CPU		
Model (# Sockets)	EPYC 7742 (2)	Xeon 8470 (2)
Base/Turbo Clock	2.25/3.4 GHz	2.00/ 3.80 GHz
Total Cores / Threads	128 / 256	104 / 208

[1] Dense/Sparse GEMM performance

matrix multiplications with support for Tensor Cores [70]. Other libraries such as cuBLAS and cuDNN are closed-source and contain multiple algorithms and implementations, including kernels from CUTLASS, to perform linear algebra and neural network operations, respectively. They use heuristics to choose the most optimized algorithms for a given problem and target devices [33, 72]. Even though libraries make developing applications that target Tensor Cores easier, the developer must still take care of data layout and memory alignment in order to correctly use Tensor Cores.

3 EXPERIMENTAL METHODOLOGY

3.1 Hardware and Software Setup

The experiments in this paper are conducted on two different platforms, each with different generations of NVIDIA GPU, as shown in Table 2. The NVIDIA A100 (Ampere) GPU is housed in the NVIDIA DGX-A100 chassis and features third-generation Tensor Cores while the NVIDIA H100 (Hopper) GPU is housed in Dell PowerEdge XE9680 chassis and features fourth-generation Tensor Cores with double the throughput of its predecessor. Both GPUs have sparsity support in their Tensor Cores which is expected to be useful for GCN that has some sparse matrix multiplications (spMM) [23, 85]. For simplification, NVIDIA A100 and NVIDIA H100 GPUs will be referred to as A100 and H100, respectively.

On the software side, the DGX-A100 is equipped with CUDA Toolkit 11.8, along with Python 3.11.4 and PyTorch 2.0.1. Meanwhile, the Dell PowerEdge XE9680 uses CUDA Toolkit 12.0, along with Python 3.11.4 and PyTorch 2.0.1 built from the source.

3.2 Performance Measurement

The **Nsight Compute** (`ncu`) is used to characterize kernels of each workload to gain access to their low-level detail. Kernel runtime is measured by collecting `gpu__time_duration` metric with cache and clock control disabled. For measuring kernel runtime in microbenchmark, the kernel is run 100 times and the average is taken. In addition, instruction count and DRAM transactions are collected. The FLOPs number is derived from the instruction count after multiplying with the weight (e.g., `fma`, `fadd`, and `fmul` have weight of 2, 1, and 1, respectively). The weight of Tensor Cores instruction is obtained based on instruction size.

The training performance for ML workloads is measured using a wall clock. For CNN, the model is trained using their respective dataset in 10 epochs with a default batch size of 128 for Image Classification and 4 for Object Detection. On the other hand, the GCN is trained in 1000 epochs because the model and dataset are small. Wall clock time measurements for determining speed-up use a large number of epochs while `ncu` profiling for roofline and instruction-mix analysis use 2 and 5 epochs for CNNs and GCN, respectively, to ensure acceptable running times with the profiler.

3.3 Profiling Tensor Cores

The legacy NVIDIA profiling tool `nvprof` [47] only provides single metric that indicates whether tensor cores are being used by a particular GPU kernel, which is accessible through `tensor_precision_fu_utilization` metric. This legacy profiling tool is no longer supported since Ampere. Meanwhile, its successor, the NVIDIA NSight Compute (`ncu`) [53] provides access to more valuable metrics on Tensor Cores with support starting from Volta.

Prior to CUDA 12.2, `ncu` provides access to `sm__inst_executed_pipe_tensor_op_xmma` to count the number of instructions being executed by Tensor Cores. It also provides access to measure Tensor Cores utilization through `sm__pipe_tensor_cycles_active` and `sm__pipe_tensor_op_xmma_cycles_active`. Note that `xmma` can be `dmma`, `hmma`, and `imma`. While the metrics are useful to indicate the interaction of the applications with the Tensor Cores, more efforts are needed to obtain more characterization metrics, such as the total number of FLOPS being executed, which is important for roofline analysis [78]. The instructions being executed on the Tensor Cores may have different shapes, which contain a different number of FLOPs per instruction. Sometimes, the kernel name suggests the instruction size being used [64] (e.g., `ampere_h16816gemm_...` means it uses `hmma.16x8x16`, which contains 4096 FLOPs per instruction), but it is difficult and is not a universal solution. Some instructions are difficult to infer the number of FLOPs without documentation, such as `hfma2.mma` that contain 4 FLOPs [31].

Finally, `ncu` shipped with CUDA 12.2 in June 2023 provides more detailed information on how many FLOPs (or IOPs) are executed on Tensor Cores. It provides access to the new metric `sm__ops_path_tensor_src_(in)_dst_(out)` where `(in)` and `(out)` are input data type and output data type. respectively. It can also be used to identify sparse FLOPs and dense FLOPs. These metrics make profiling applications that target Tensor Cores easier, especially those that use `wgmma.mma_async` in Hopper.

3.4 Workload Configuration

To evaluate Tensor Cores' performance, two groups of workloads are prepared, consisting of end-to-end ML training and microbenchmark as shown in Table 3 and 4, respectively.

3.4.1 Machine Learning Workloads. The CNN workloads consist of four models with two different tasks, as shown in Table 3. The ResNet50 [21] and EfficientNet [69] are CNN models for image classification, which are trained using the ImageNet dataset [12]. In addition, Faster-RCNN [62] and RetinaNet [35] are CNN models for Object Detection, which are trained using COCO dataset [36].

In addition to CNNs, which have been widely evaluated on GPUs, we also use GCNs as an emerging ML workload for this study. The GCN consists of only one model [29] with two different tasks: semi-supervised node classification tasks, either transductive or inductive [77]. For the transductive approach, PubMed [65] and Chameleon [63] datasets are used, while for the inductive approach, Yelp [82] and Reddit [20] datasets are used. Interested readers can obtain

Table 3: Machine Learning Workload Configuration

Type	Model	Task	Dataset
CNN	ResNet50	Image Classification	ImageNet
CNN	EfficientNet		
CNN	FasterRCNN	Object Detection	COCO
CNN	RetinaNet		
GNN	GCN	Node Classification Transductive	PubMed
GNN	GCN		Chameleon
GNN	GCN	Node Classification Inductive	Reddit
GNN	GCN		Yelp

Table 4: Microbenchmark Kernels Configuration

Type	ID	Dimension	Config
GEMM {m,n,k}	512	{512,512,512}	fp32 fp16.1688 fp16.16816
	2K	{2048,2048,2048}	
	8K	{8192,8192,8192}	
	32K	{32768,32768,32768}	
GEMV {m,1,k}	512	{512,1,512}	fp32 fp16.1688 fp16.16816
	2K	{2048,1,2048}	
	8K	{8192,1,8192}	
	32K	{32768,1,32768}	
Conv2D {N,H,W,C}; {K,R,S}; {U,V}	A	{64,1024,1024,32}; {16,32,32}; {1,1}	fp32 fp16
	B	{64,1024,1024,32}; {16,32,32}; {1,1}	
	C	{256,512,512,32}; {16,32,32}; {1,1}	
	D	{64,1024,1024,32}; {16,4,4}; {8,8}	
	E	{32,512,512,256}; {16,32,32}; {1,1}	
	F	{32,512,512,32}; {256,32,32}; {1,1}	
FIR {s,f}	8M4	{8388608, 4}	fp32.af fp16.ig fp16.rg
	8M8	{8388608, 8}	
	32M8	{33554432, 8}	
	32M16	{33554432, 16}	
	128M16	{134217728, 16}	
	128M32	{134217728, 32}	
ElWiseAdd {v}	256K	{262144}	fp32 fp16
	4M	{4194304}	
	64M	{67108864}	
	1B	{1073741824}	

more details from the original paper [29], a review by Zhang et al. [83, 84], and a summary by Heidar et al. [22],

PyTorch [60] is used as the framework to perform ML training in this experiment. All of the CNN models are taken from TorchVision [37] while the GCN model is taken from CogDL [5], a research toolkit for deep learning graphs. This toolkit integrates the original code from Kipf et al. [29] with built-in methods to load various datasets, making it easier to do experiments. For the FP32 (full-precision) training flow, PyTorch Automatic Mixed Precision (AMP) is disabled to avoid Tensor Cores usage, while for the FP16 (mixed-precision), AMP is enabled, allowing Tensor Cores usage.

3.4.2 Microbenchmark Kernels. The microbenchmark consists of five kernels with configurations given in Table 4. The kernels are developed using C++ and CUDA which target CUDA Cores or Tensor Cores. The kernels have customizable precision, input dimensions, target execution units, and libraries. Except otherwise noted, CUTLASS [70] is the library used for two reasons: 1) CUTLASS is open-source, which allows modification of template header or low-level assembly; and 2) CUTLASS is deterministic in terms of overall execution, which allows using application replay in `ncu` for profiling while cuBLAS use heuristics to choose the best kernel depending on problem size and device. The FP32 and F16 implementations target CUDA Cores and Tensor Cores, respectively.

- **GEMM**: General matrix-matrix multiply with dimensions {m,n,k} denoting $A_{m\times n} \times B_{n\times k} = C_{m\times k}$ where A, B, C are matrices. GEMM is well-supported by CUTLASS, which has one of the most efficient hierarchical GEMMs supporting CUDA Cores or Tensor Cores. However, data layout in memory must be taken care of carefully [28]. FP16 implementation uses two Tensor Cores instructions: `hmma.16816` (fp16.16816) and `hmma.1688` (fp16.1688).
- **GEMV**: General matrix-vector multiply with dimensions {m,1,k} can be viewed as a special case of GEMM. It follows the same implementation as GEMM.
- **Conv2D**: Two-dimension convolution is decomposed into implicit GEMM [86] by CUTLASS on CUDA Cores or Tensor Cores. The Conv2D kernel has multiple configurations with {N,H,W,C} denotes batch size, height, width, and number of input channels, respectively. In contrast, {K,R,S} denotes the number of channels, height, and width of the filter, respectively. The {U,V} are horizontal and vertical stride, respectively.
- **FIR**: 1D Finite Impulse Response filtering which operates on 1D signal s and 1D filter f. The FP32 uses the ArrayFire library (fp32.af) [79] while the FP16 implementation is not supported by Tensor Cores by default. Although earlier studies have tried to map FIR into Tensor Cores, they use 2D signals and filters [30]. Therefore, for the purpose of this experiment, two approaches to map 1D FIR into Tensor Cores are proposed as follows:
  - *Implicit GEMV (fp16.ig)*: This approach is done by modifying CUTLASS implicit GEMM into implicit GEMV. Due to memory alignment requirements, many zero-padding needs to be added, resulting in 64× more operations than is necessary.
  - *Reshaped GEMV (fp16.rg)*: Another approach is to construct a matrix from a 1D signal, which will be multiplied by the vector containing the filter. Suppose a filter $f = \{f_0\ f_1\ f_2\}$ is applied into input signal $s = \{s_0\ s_1\ s_2\ s_3\}$. Then, a 4×3 matrix is constructed with first row $\{0\ 0\ s_0\}$, second row $\{0\ s_0\ s_1\}$, third row $\{s_0\ s_1\ s_2\}$, and fourth row $\{s_1\ s_2\ s_3\}$. Then, a GEMV can be performed between the signal matrix and the filter vector. This approach has one drawback regarding data reuse and memory usage where the same data appears multiple times (e.g., s_0 in the first row is the same data as s_0 in the second row but stored twice in the memory).
- **ElWiseAdd**: Element-wise vector addition operates on two vectors of the same configurable lengths $\{v\}$. The FP32 implementation (fp32) uses only C++, while the FP16 implementation (fp16) is not supported in Tensor Cores. While cuBLAS supports vector addition operation, which can be represented by $ax + y$ with scaling factor $a = 1$, at the time of writing, cuBLAS only supports this operation in CUDA Cores for single precision and double precision with no Tensor Cores support [50]. Therefore, to be able to run vector addition in Tensor Cores, both vectors must be transformed into matrices to follow the Tensor Cores operation shown in Figure 2 with matrix B being an identity matrix and matrix A and C are the two input vectors. The multiplication cannot be skipped as it is the basic operation of Tensor Cores (i.e., `mma`), resulting in expensive computation and memory access.

4 EVALUATION & DISCUSSION

4.1 What do Tensor Cores bring to the table over CUDA Cores?

Tensor Cores provide a significant jump in compute throughput for GEMM and GEMM-like kernels if specific precisions are used. Figure 3 presents the speed-up achieved for CNN and GCN workloads (Table 3), and microbenchmark kernels (Table 4) on H100.

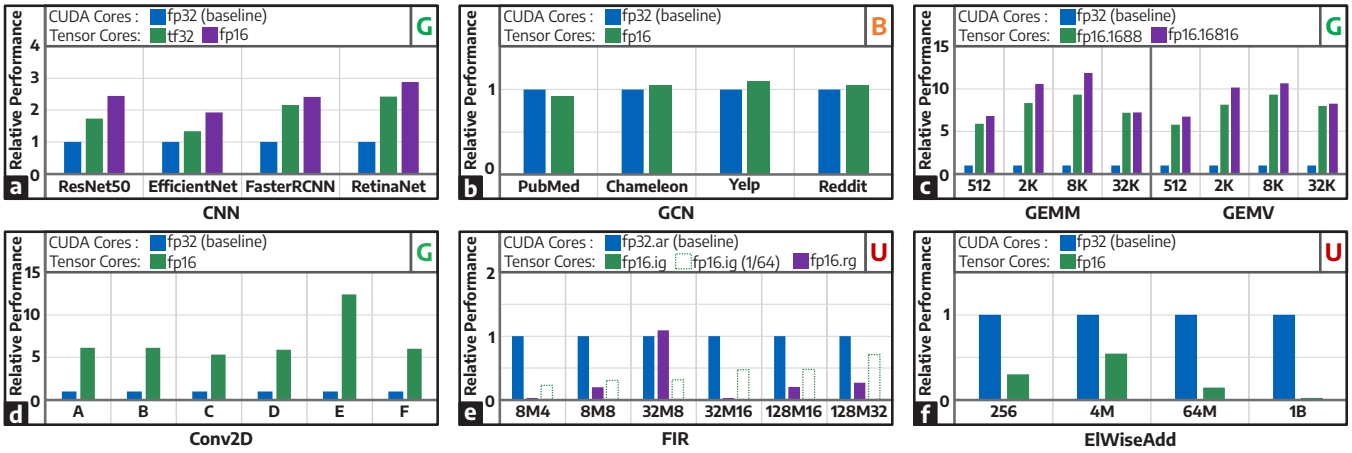


Figure 3: The speed-up obtained by using Tensor Cores over CUDA Cores on H100 across CNN/GCN workloads, and microbenchmark kernels. CNNs, Conv2D, and GEMM/GEMV are high performers with G, B, and U indicate Good, Bad, and Ugly, respectively.

4.1.1 CNN Workloads. Figure 3-a illustrates that going from FP32 on CUDA Cores to TF32 on Tensor Cores gives an average 1.91× speed-up while going from FP32 on CUDA Cores to FP16 on Tensor Cores gives an average 2.42× speed-up. To run FP32 full precision training on CUDA Cores as the baseline, PyTorch Automatic Mixed Precision (AMP) is explicitly disabled. However, the underlying CUDA libraries (e.g., cuBLAS, cuDNN) automatically demote FP32 to TF32 [8] to take advantage of Tensor Cores. Hence, an environment variable `NVIDIA_TF32_OVERRIDE=0` is set to tell CUDA libraries not to use TF32 explicitly. As a result, there are three configurations shown in Figure 3-a: full precision FP32 (blue), full precision TF32 (green), and mixed precision FP16 (purple).

4.1.2 GCN Workloads. Unlike CNN, GCN uses FP32 by default for full precision training, most likely due to the CogDL [5] that does not take advantage of TF32 on the underlying CUDA libraries. Furthermore, as shown in Figure 3-b, it only sees an average speed-up of 1.03× when going from FP32 on CUDA Cores to FP16 on Tensor Cores. Further explanation using rooflines and matrix instruction usage is provided in Section 4.2.4 and Section 4.3.2.

4.1.3 Microbenchmark Kernels. In summary, GEMM, GEMV, and Conv2D kernels get the performance benefit while FIR and ElWiseAdd experience performance degradation, as discussed below.

- **GEMM:** GEMM gets an average speed-up of 7.69× and 9.14× for fp16.1688 and fp16.16816, respectively, as shown in Figure 3-c. The highest speed-up is observed with GEMM_8K at 9.32× for fp16.1688 and 11.89× for fp16.16816, before dropping to 7.25× and 7.20× , respectively, for GEMM_32K. The GEMM_32K has vastly more elements (200M for GEMM_8K vs. 3.2B for GEMM_32K) and more intermediate results, exacerbating the data movement between on-chip and off-chip memory, which will become clear when we perform roofline analysis in Section 4.2.5.
- **GEMV:** GEMV gets performance benefits from Tensor Cores, although its average speed-up is lower than GEMM due to its lower arithmetic intensity. The achieved average speed-up is 7.82× for fp16.1688 and 8.96× for fp16.16816 (Figure 3-c).
- **Conv2D:** Since Conv2D is decomposed into implicit GEMM, it can take advantage of Tensor Cores; it achieves an average speed-up of 6.99× (Figure 3-d). The highest speed-up of 12.42× comes from Conv2D_E, whose reason will become clear in Section 4.2.5.
- **FIR:** Both FP16 implementations that target Tensor Cores show significant performance degradation as shown in Figure 3-e; the fp16.rg and fp16.ig only achieve an average of 0.30× and 0.01× performance achieved by fp32.ar that runs on CUDA Cores, respectively. The fp16.rg has redundant operations (Section 3.4.2), causing the performance drop for larger signal and filter dimensions. The fp16.ig is even more slower than the fp16.rg because of the 64 times more operations it needs to perform due to the zero-padding (Section 3.4.2). Even if there is a way to make these additional operations useful (e.g., having batched inputs with the same FIR filter or multiple independent FIR filters), it still cannot compete with the fp32.ar for large signal size (dashed green bars).
- **ElWiseAdd:** Like the FIR, ElWiseAdd also sees performance degradation, especially for larger dimensions, where it achieves an average of 0.25× performance offered by CUDA Cores as shown in Figure 3-f. While the matrix addition is fast, the multiplication with the identity matrix that cannot be skipped is expensive, especially in larger dimensions (Section 3.4.2).

4.2 Is Compute the Bottleneck or Memory?

4.2.1 Overview of Roofline Model. We use roofline charts [78] to visualize the achieved performance of applications or kernels compared to the hardware's compute capabilities and draw insights on the arithmetic intensity of applications. Both axes of the model are plotted in logarithmic scale: the y-axis represents the compute throughput (e.g., floating-point operations per second) while the x-axis represents the arithmetic intensity, which is the amount of computing that can be done per byte of data (e.g., floating-point operations per byte). The hardware roofline model, which can be obtained theoretically (e.g., from manufacturer datasheet, such as the data provided in Table 2) or empirically (e.g., using Empirical Roofline Toolkit [80]), consists of peak compute throughput, drawn as the roof, and the peak memory bandwidth (e.g., off-chip memory, cache bandwidth), drawn as the slope. Using data obtained from profiling tools (e.g., execution duration, the number of operations, and the number of memory read and write), the position of each application or kernel in the roofline chart can be determined, which gives insight whether the application or kernel is compute- (i.e., closer to the roof) or memory-bound (i.e., closer to the slope) and what optimization techniques should be performed.

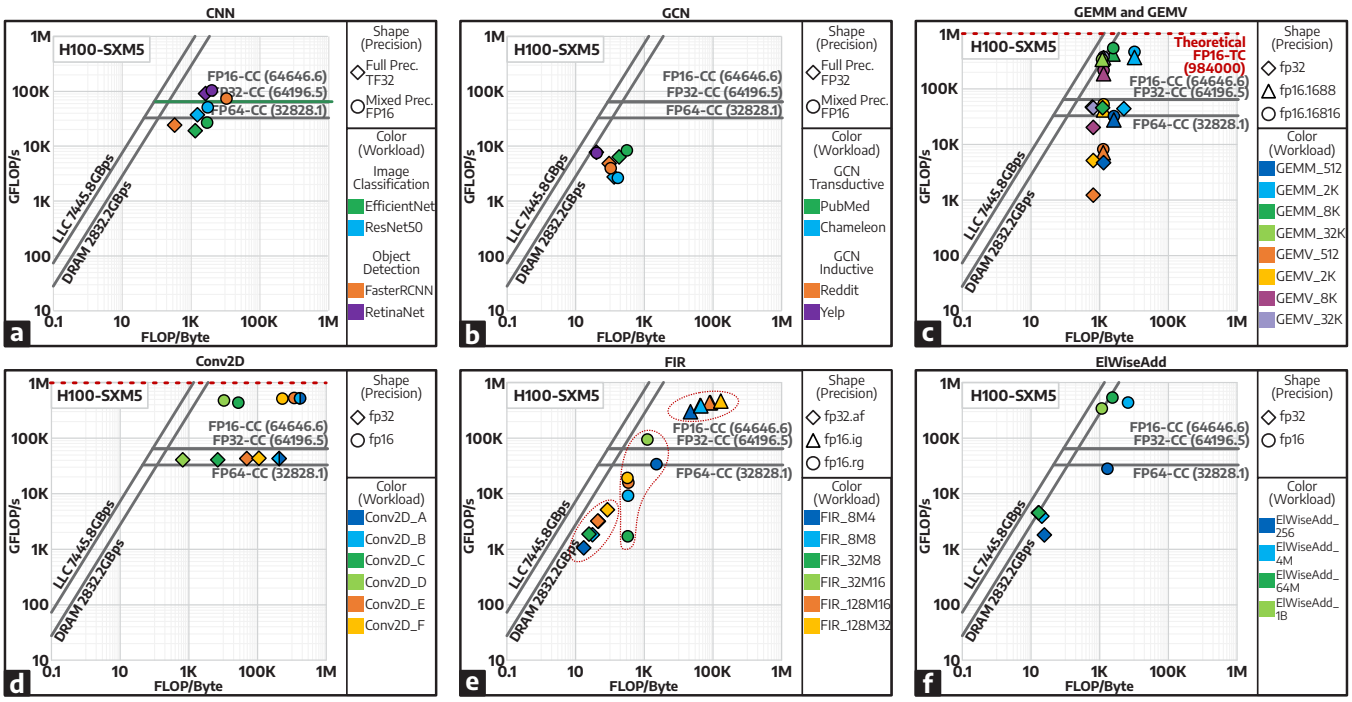


Figure 4: The roofline model for H100 (obtained using ERT), and the characterization of CNN/GCN workloads, and microbenchmark kernels. GCN has less than 1K flops/byte while Conv2D goes above 100K flops/byte. Diamond shape indicates baseline without Tensor Cores and triangles/circles indicate Tensor Cores versions.

4.2.2 H100 Roofline Model. Figure 4 shows the hardware roofline model for H100, obtained using ERT [80]. There are two points to highlight: 1) The roofs represent the peak compute throughput of the CUDA Cores: 64.64 TFLOP/s for FP16 (FP16-CC), 64.12 TFLOP/s for FP32 (FP32-CC), and 32.82 TFLOP/s for FP64 (FP64-CC) since, at the time of writing, ERT does not support `hmma` nor `hgmma` to measure the peak compute throughput of Tensor Cores (FP16-TC); and 2) the ERT is only able to achieve 50% of the theoretical compute throughput of FP16 on CUDA Cores (Table 2). The latter may be caused by two reasons: 1) ERT may need to be updated to account for new architecture, or 2) The CUDA Cores of Hopper may have the same FP16 compute throughput as the FP32. This happens with Ada Lovelace [54] (e.g., NVIDIA L40S [55]), which shares some of the architecture with Hopper, although Hopper datasheet mentions FP16 to be twice the rate of FP32 on CUDA Cores [49]. For the bandwidth, ERT is able to achieve 2,832 GBps on the HBM3 DRAM (84.5% of 3,350 GBps theoretical bandwidth for H100).

4.2.3 CNN Workloads. The use of TF32 during full precision training (Section 4.1.1) allows all models to achieve significantly higher GFLOP/s, with some exceeding the FP32-CC roof, by leveraging Tensor Cores (Figure 4-a). The performance improvements in using TF32 compared to FP32 for full-precision training are two folds: 1) Convolution operations, which are abundant in CNN, can be done on Tensor Cores, which have significantly higher compute throughput than CUDA Cores; and 2) TF32 has lower 19-bit data size compared to FP32 32-bit data size, which reduces the pressure on the memory bandwidth. Furthermore, the use of FP16 on mixed precision training by enabling PyTorch Automatic Mixed Precision improves performance even further, which comes from the ability of Tensor Cores to compute FP16 at twice the rate of TF32 and slightly lower data size (16-bit FP16 vs. 19-bit TF32). Special mention goes to FasterRCNN, shown in orange color, which gets the most benefits (i.e., biggest change in FLOPs/byte) from reduced memory bandwidth by switching from TF32 to FP16.

4.2.4 GCN Workloads. In general, all of the GCN workloads are memory-bound, even after switching from full-precision training (FP32) to mixed-precision training (FP16) as shown in Figure 4-b. The use of Tensor Cores for mixed-precision training has very few improvements in performance as discussed in Section 4.1.2; only PubMed, shown in green, enjoys some improvements compared to other GCN configurations in terms of arithmetic intensity and achieved compute throughput. However, it does not translate to positive speed-up (Figure 3) due to extra operations needed when using Tensor Cores (e.g., COO to CSR sparse matrix format conversion).

4.2.5 Microbenchmark Kernels. The roofline analysis for each kernel of the microbenchmark is given as follows.

- **GEMM:** The GEMM kernels are shown in dark blue, light blue, dark green, and light green colors in Figure 4-c. The dimension of GEMM_512 (dark blue) is too small to take advantage of the compute throughput offered by either CUDA Cores (diamond) or Tensor Cores (triangle and circle). Meanwhile, the other GEMM configurations (GEMM_2K, GEMM_8K, GEMM_32K) in FP32 (diamond) can almost saturate the FP32 compute throughput offered by CUDA Cores (i.e., almost hitting the roof of FP32-CC). The FP16 version of GEMM_8K and GEMM_32K (dark green triangle, green triangle, dark green circle, and green circle) can push through the roof of FP16-CC thanks to the use of Tensor Cores until the memory bandwidth of HBM3 DRAM becomes their limit. The theoretical FP16 performance of the Tensor Cores in H100 is 989 TFLOP/s (Table 2), which most likely won't be achieved by

GEMM due to memory bandwidth limitation. The use of larger `hmma.16816` (fp16.16816), denoted by circle, gives higher compute throughput compared to the `hmma.1688` (fp16.1688), denoted by triangle, while giving the same arithmetic intensity. Finally, it is worth mentioning that the GEMM_2K has a significantly higher FLOP/byte compared to other configurations. The dimension of the matrices is small enough to fit into on-chip memory. The 12 Million (2048×2048×3) FP16 elements have a total size of around 24 MB while H100 has 33 MB of registers, 33 MB of combined L1 Cache and Shared Memory, and 50 MB of L2 cache.

- **GEMV:** The GEMV kernels are shown in orange, yellow, purple, and violet colors in Figure 4-c. GEMV has lower data reuse compared to GEMM, and hence lower arithmetic intensity (i.e., located to the left of GEMM counterparts) and lower number of operations, especially for lower dimensions GEMV_512 and GEMV_2K (orange and yellow) whose FP32 versions (diamond) cannot fully utilize the available CUDA Cores on H100. On the other hand, the largest dimension (GEMV_32K) can almost hit the FP32-CC roof. Moving to FP16 versions (triangle and circle), only GEMV_8K and GEMV_32K can push through the roof of FP16-CC until they hit the memory bandwidth slope. Like the GEMM, the use of `hmma.16816` (fp16.16816) gives higher compute throughput compared to the `hmma.1688` (fp16.16816) on GEMV.
- **Conv2D:** As mentioned earlier in Section 3.4.2, the 2D Convolution is decomposed into implicit GEMM. The 2D convolution has more data reuse compared to GEMM, where the data reuse mostly comes from the use of 2D filters, which are applied to many 2D input signals. As shown in Figure 4-d, in general, both FP32 (diamond) and FP16 (circle) of Conv2D almost reach the roof of FP32-CC and the theoretical roof of FP16-TC (drawn as a dashed red line), respectively. The Conv2D_A (dark blue) and Conv2D_B (light blue), which have sixteen 32×32 filters (Table 4), have the most data reuse, leading to the highest arithmetic intensity (i.e., located to the right side of the roofline chart). The amount of memory needed to store all of these filters in both FP32 and FP16 are 64 KB and 32 KB, respectively, which can be stored sufficiently inside the shared memory of H100 (256 KB of combined L1+Shared memory per SM). On the other hand, Conv2D_C (dark green) and Conv2D_D (light green) have the least data reuse due to the smaller size of filters being used (Conv2D_C) and the larger convolution stride (Conv2D_D). Moving to FP16 with Tensor Cores (circle), all Conv2D configurations push through the FP16-CC roof. Special mention goes to Conv2D_E (orange) with its 256 input channels and smaller 512×512 input signals that allow for more data reuse. It almost achieves the theoretical FP16 peak performance of Tensor Cores, followed by Conv2D_F (yellow).
- **FIR:** Figure 4-e shows three clusters of workloads, which correspond to three implementations of FIR as discussed in (Section 3.4.2): fp32.ar (diamond), fp16.rg (circle), and fp16.ig (triangle). The FP32 version (fp32.ar) is already bandwidth-limited, with all of them positioned near each other at the slope of HBM3 DRAM. This also indicates that Tensor Cores cannot accelerate FIR as it is already bandwidth limited, unlike GEMM, GEMV, and Conv2D. The fp16.rg implementation has higher arithmetic intensity due to the redundant operations as a result of how the signal's data is laid out to form a matrix as discussed in Section 3.4.2. On the other hand, the fp16.ig tries to mimic the implicit GEMM that Conv2D has, except it uses implicit GEMV. Nevertheless, both approaches to map FIR to Tensor Cores (fp16.ig and fp16.rg) show unfavorable results compared to the fp32.ar on CUDA Cores.
- **ElWiseAdd:** Figure 4-f shows the FP32 version (diamond) of element-wise addition is already memory-bound with very low arithmetic intensity, hitting the slope of HBM3 DRAM bandwidth. On the other hand, the FP16 version (circle) has higher compute throughput and arithmetic intensity, which solely comes from the fact that the element-wise addition must be transformed to matrix-multiply-accumulate operations to be able to use Tensor Cores. Sadly, this does not improve performance since the multiplication is expensive, especially for large matrix sizes.

4.3 What Percentage of Floating-Point Instructions Offloaded to Tensor Cores?

Figure 5 shows the floating-point instruction/operation mix for CNN, GCN, and microbenchmark kernels. Since Tensor Cores instruction performs multiple floating-point operations, the weighted numbers are used (Section 3.2). The instruction/operation mix gives insight into what instructions could be offloaded to Tensor Cores.

4.3.1 CNN Workloads. As previously discussed in Section 4.2.3, the underlying CUDA libraries demote the FP32 to TF32 for full precision training in order to use Tensor Cores. This is further confirmed by the instruction mix shown in Figure 5 (top four sets) where most floating-point instructions are TF32 running on Tensor Cores with `hmma.1688` instructions shown as yellow bar (e.g., GEMM kernel `sm80_xmma..._tf32f32...`) and newer `hgmma` shown as olive-green bar (e.g., GEMM kernel `sm90_xmma..._tf32f32...`). Small percentage of operations are still executed by CUDA Cores as shown by the green (FP16) and blue (FP32) bar, which come from kernels that cannot be mapped into Tensor Cores (e.g., element-wise). Moving to mixed precision training with FP16, the composition is largely the same with FP16 running on Tensor Cores with `hmma.1688` (light orange bar), `hmma.16816` (dark orange bar), and newer `hgmma` (dark brown bar) instructions. It is worth mentioning that `ncu` shipped with CUDA 12.2 is used to calculate the number of floating-point operations that `hgmma` instructions do as it is difficult to infer this information from kernel name alone (Section 3.3).

4.3.2 GCN Workloads. Unlike CNN, the full precision training on GCN uses FP32 on CUDA Cores as shown in Figure 5 (middle four sets of bars) where majority of the instructions are `ffma`. Moving to mixed precision training with FP16, none of them use the newer `hgmma` instructions on Tensor Cores; the majority use `hmma.1688` and `hmma.16816` with Chameleon is observed to use older `wmma.161616` instructions. In addition, a small number of FP32 and FP16 instructions are executed on CUDA Cores, particularly for element-wise kernels, which are many in GCN workloads, outweighing the speed-up provided by Tensor Cores.

4.3.3 Microbenchmark Kernels. Unlike CNN and GCN workloads, the data type and instruction size used in the microbenchmark kernels can be specified explicitly. The lowest four sets of bars illustrate the microbenchmarks in Figure 5.

- **GEMM, GEMV, and Conv2D:** Both GEMM and GEMV have the instruction mix corresponding to the data type and the instruction size used: FP32 mostly uses `ffma` on CUDA Cores (blue bar) while FP16 mostly uses either `hmma.1688` (light orange bar) or

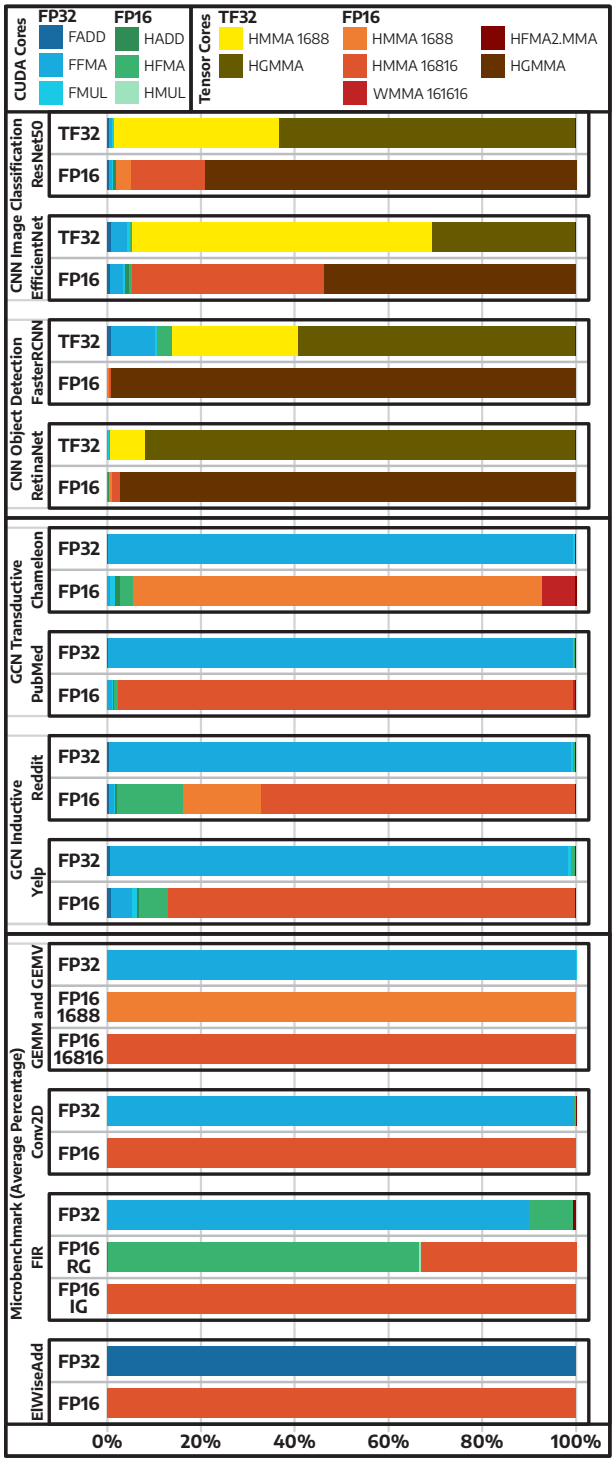


Figure 5: The floating-point instruction mix of CNNs, GCNs, and microbenchmark kernels on H100 utilizing CUDA libraries (e.g., cuDNN, cuBLAS, CUTLASS). Note that full precision training in CNN will, by default, use TF32 instead of FP32. Yellow/brown/orange/red run in Tensor Cores and blue/green run in CUDA Cores.

`hmma.16816` (dark orange bar) on Tensor Cores. Since Conv2D is decomposed to implicit GEMM, it follows the behavior of GEMM.

- **FIR:** The fp32.ar implementation uses `ffma` and `hfma` that runs on CUDA Cores. On the other hand, the fp16.rg implementation still has the majority of the FP16 instructions executed in CUDA Cores as `hfma` (green bar) while some of the instructions are executed in Tensor Cores with `hmma.16816` instructions. Finally, the fp16.ig implementation spends the majority of the instructions on Tensor Cores as `hmma.16816`. Only a small percentage of Tensor-Core-bound instructions are useful since most of them are due to padding and memory alignment.
- **ElWiseAdd:** The FP32 implementation uses `fadd` on CUDA Cores while the FP16 implementation uses `hmma.16816` on Tensor Cores. Unfortunately, for FP16, most of the instructions are spent on the expensive matrix-multiply operations, which are not useful since the only useful operation is addition.

4.4 How much more performance does H100 provide over A100?

Table 2 shows the theoretical peak performance of H100 is 3.4× in FP32 and FP16 on CUDA Cores and 3.2× in TF32 and FP16 on Tensor Cores compared to A100. The H100 achieves these theoretical performance improvements by having 2.5× higher number of CUDA Cores (16896 vs. 6912), doubling the Tensor Cores throughput per SM per cycle, doubling the memory bandwidth (3.3 TB/s vs. 1.5 TB/s), pushing the TDP higher (700 W vs. 400 W), running at higher sustained clock frequency (1980 MHz vs. 1410 MHz), and having other new features that help with execution efficiency. This section compares the achieved performance improvements of H100 over its predecessor, the A100, for the experimented CNNs, GCNs, and microbenchmark kernels as shown in Figure 6.

4.4.1 CNN Workloads. The H100 achieves an average of 1.96×, 1.96×, and 1.88× speed-up for FP32 on CUDA Cores, TF32 on Tensor Cores, and FP16 on Tensor Cores, respectively, across four CNN workloads over A100 as shown in Figure 6-a.

4.4.2 GCN Workloads. Figure 6-b shows the speed-up achieved by H100 over A100 on GCN. We observed a significantly high speedup on GCN with Yelp and Reddit datasets. For GCN with PubMed and Chameleon datasets, performance improvements on H100 over A100 are insignificant, with an average speed-up of 1.12×. When running GCN training on H100 with CUDA 12.0, the Chameleon mixed precision training flow is broken while its full precision shows double the time needed compared to A100. Reverting back to CUDA 11.8 solves the issue. Interestingly, it is the other way around for both Yelp and Reddit which enjoy significant improvements when using CUDA 12.0 on H100 for two reasons: 1) sparse-matrix multiplication (`spmm` [23]) kernel is being used, which is not found when running on A100; 2) the use of newer `hgmma` instruction.

4.4.3 Microbenchmark Kernels. The microbenchmark kernels that target Tensor Cores use either `hmma.1688` and `hmma.16816` instructions; none of them use the newer `hgmma` instructions supported by H100, which may affect the attainable performance.

- **GEMM, GEMV, and Conv2D:** H100 achieves average speed-up of 3.01×, and 2.36×, and 1.98× for GEMM with FP32 on CUDA Cores, FP16 using `hmma.16816` on Tensor Cores, and FP16 using `hmma.1688` on Tensor Cores, respectively over A100 (Figure 6-c). The speed-up is lower for GEMV with an average of 2.74×, 2.00×,

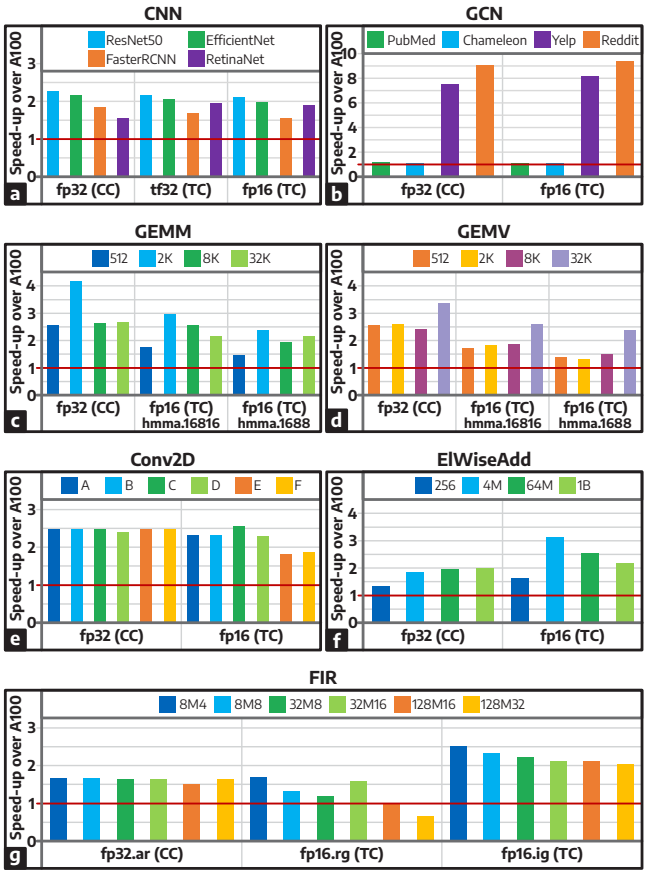


Figure 6: Significant speed-up is achieved by H100 over A100 for most benchmarks. Red line indicates A100 performance (baseline).

and 1.65×, respectively, due to lower arithmetic intensity (Figure 6-d). Finally, H100 reaches an average speed-up of 2.45× and 2.20× on Conv2D for FP32 (CUDA Cores) and FP16 (Tensor Cores) over A100, respectively (Figure 6-e).

- **FIR and ElWiseAdd:** While FIR (Figure 6-f) and ElWiseAdd (Figure 6-e) do not benefit from Tensor Cores, H100 achieved an average speed-up of 1.62×, 1.23×, 2.21×, 1.78×, and 2.37× for FIR fp32.ar, FIR fp16.rg, FIR fp32.ig, ElWiseAdd FP32, and ElWiseAdd FP16, respectively.

4.5 Discussion

4.5.1 Empirical Roofline Toolkit. The ERT [80] is a useful tool for creating a roofline model of the hardware. However, it does not have support to find the roof for Tensor Cores using either `mma` or `wgmma.mma_async`. From the roofline analysis (Figure 4), Conv2D is one of the likely kernels that can be used to measure the roof of Tensor Cores performance, since it can almost reach the theoretical peak throughput of Tensor Cores.

4.5.2 Profiling non-deterministic application. While it is recommended to use application replay when profiling using `ncu` [52] to avoid the overhead of kernel replay, profiling non-deterministic workloads such as ML training flows [87] may need to use kernel replay instead. Although we have followed steps to maintain reproducibility and control randomness in PyTorch [71], `ncu` with application replay is unable to consolidate profiling results due to the mismatch in kernel names and kernel launch parameters, which is an indication that the applications do not take the same execution path every time it runs during the replay.

4.5.3 Reshaping Optimizations. Both FIR and ElWiseAdd will not run on Tensor Cores without reshaping optimization to map them into GEMM-like operations (Section 3.4.2). Unfortunately, reshaping comes with costs due to memory alignment and padding, making the performance benefit of Tensor Cores difficult to come by. Finer control of Tensor Cores (e.g., the ability to skip the multiplication on MMA operations) may be beneficial for element-wise operations that often follow GEMM/GEMV operations by fusing both GEMM kernels and element-wise kernels to significantly reduce data movement and kernel switching overhead.

4.5.4 TensorFloat32. The TensorFloat32 (TF32) was introduced by NVIDIA along with third-generation Tensor Cores (Section 2.1) [8]. TF32 is a 19-bit data type with 8-bit exponent to retain the dynamic range of FP32 and 10-bit mantissa to achieve the same accuracy as FP16, which has been proven to be sufficient for ML workloads. Since TF32 can run on Tensor Cores and gives significant speed-up over FP32 on CUDA Cores, many frameworks that rely on NVIDIA libraries allow the demotion of FP32 to TF32 (e.g., through option `CUBLAS_TF32_TENSOR_OP_MATH` on cuBLAS) if the GPU supports TF32. While this may work fine for many ML workloads, it may cause numerical instability for applications where accuracy is important, such as in HPC applications. Therefore, making sure of precision to use is important (e.g., explicitly configure CUDA libraries to keep using FP32 when needed).

5 CONCLUSION

Tensor Cores provide significant speed-up for applications that have abundant GEMM operations. CNNs yield "Good" improvements with Tensor Cores, exemplified by the average speedups of 1.91× and 2.42× with TF32 and FP16 training, respectively, compared to FP32 training running on the CUDA Cores. Kernels like GEMM, GEMV, and Conv2D also show "Good" advantage of Tensor Cores with an impressive 8.4×, 8.39×, and 6.99× average speed-up, respectively. The Conv2D kernel almost saturates the FP16 theoretical performance of Tensor Cores on H100. On the other hand, FIR and ElWiseAdd kernels show performance degradation when running on Tensor Cores despite code transformations, making them "Ugly" kernels for Tensor Cores. Furthermore, GCN improvement with Tensor Cores can be classified as "Bad" since they only achieved 1.03× average speed-up and are sensitive to the changes in library versions. Finally, H100 gives an impressive 2.33× average speed-up across CNNs, GCNs, and microbenchmark kernels over A100. These speed-ups are mostly due to the H100 having 2.5× more CUDA Cores, double the throughput of Tensor Cores, and double the memory bandwidth compared to the A100.

ACKNOWLEDGMENTS

This research was supported in part by Semiconductor Research Corporation (SRC) Tasks 3015.001 and 3148.001, National Science Foundation (NSF) Grant #2326894, and NVIDIA Applied Research Accelerator Program Grant. Any opinions, findings, conclusions, or recommendations are those of the authors and not of the funding agencies. The authors would also like to thank the reviewers for their constructive feedback and suggestions.

REFERENCES

[1] Advanced Micro Devices. 2020. *AMD CDNA Architecture.* Whitepaper. Advanced Micro Devices, US.

[2] Pedro Martins Basso, Fernando Fernandes dos Santos, and Paolo Rech. 2020. Impact of Tensor Cores and Mixed Precision on the Reliability of Matrix Multiplication in GPUs. *IEEE Transactions on Nuclear Science* 67, 7 (2020), 1560–1565. https://doi.org/10.1109/TNS.2020.2977583

[3] Harun Bayraktar. 2020. How CUDA Math Libraries Can Help You Unleash The Power of The New NVIDIA A100 GPU. *NVIDIA GPU Technology Conference (GTC)* s21681 (May 2020).

[4] Davis Blalock and John Guttag. 2021. Multiplying Matrices Without Multiplying. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, Virtual, 992–1004. https://proceedings.mlr.press/v139/blalock21a.html

[5] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, Yizhen Luo, Zhongming Yu, Hengrui Zhang, Xingcheng Yao, Aohan Zeng, Shiguang Guo, Yuxiao Dong, Yang Yang, Peng Zhang, Guohao Dai, Yu Wang, Chang Zhou, Hongxia Yang, and Jie Tang. 2023. CogDL: A Comprehensive Library for Graph Deep Learning. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. ACM, New York, NY, USA, 747–758. https://doi.org/10.1145/3543507.3583472

[6] Junkyeong Choi, Hyucksung Kwon, Woongkyu Lee, Jieun Lim, and Jungwook Choi. 2022. Understanding and Optimizing INT4 Convolution for Accelerated DNN Inference on Tensor Cores. In *2022 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, Rennes, France, 1–6. https://doi.org/mdrz

[7] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* 43, 3 (2023), 9–17. https://doi.org/10.1109/MM.2023.3256796

[8] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

[9] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and Programmability. *IEEE Micro* 38, 2 (2018), 42–52. https://doi.org/mdrs

[10] Rezaul Chowdhury, Francesco Silvestri, and Flavio Vella. 2020. A Computational Model for Tensor Core Units. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '20)*. ACM, New York, NY, USA, 519–521. https://doi.org/10.1145/3350755.3400252

[11] Yi-Hua Chung, Cheng-Jhih Shih, and Shih-Hao Hung. 2022. Accelerating Simulated Quantum Annealing with GPU and Tensor Cores. In *High Performance Computing*, Ana-Lucia Varbanescu, Abhinav Bhatele, Piotr Luszczek, and Baboulin Marc (Eds.). Springer International Publishing, Cham, 174–191.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Miami, Florida, USA, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[13] Sultan Durrani, Muhammad Saad Chughtai, Abdul Dakkak, Wen-mei Hwu, and Lawrence Rauchwerger. 2021. FFT Blitz: The Tensor Cores Strike Back. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. ACM, New York, NY, USA, 488–489. https://doi.org/10.1145/3437801.3441623

[14] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. ACM, New York, NY, USA, 278–291. https://doi.org/mdr8

[15] Jesun Saharian Firoz, Ang Li, Jiajia Li, and Kevin Barker. 2020. On the Feasibility of Using Reduced-Precision Tensor Core Operations for Graph Analytics. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–7. https://doi.org/10.1109/HPEC43674.2020.9286152

[16] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. ACM, New York, NY, USA, 84–93. https://doi.org/10.1145/3289602.3293906

[17] B. Gallet and M. Gowanlock. 2022. Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 135–144. https://doi.org/mdrr

[18] Chris Gottbrath. 2018. Using TensorRT to Unlock Tensor Core Performance for Inference. *NVIDIA GPU Technology Conference (GTC)* dc8169 (Oct 2018).

[19] Azzam Haidar, Harun Bayraktar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. 2020. Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 476, 2243 (2020), 20200110. https://doi.org/10.1098/rspa.2020.0110

[20] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[22] Negar Heidari, Lukas Hedegaard, and Alexandros Iosifidis. 2022. Chapter 4 - Graph convolutional networks. In *Deep Learning for Robot Perception and Cognition*, Alexandros Iosifidis and Anastasios Tefas (Eds.). Academic Press, Cambridge, Massachusetts, United States, 71–99. https://doi.org/mdrq

[23] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Atlanta, GA, USA, 1–12. https://doi.org/10.1109/SC41405.2020.00076

[24] H. Jiang. 2022. Intel's Ponte Vecchio GPU : Architecture, Systems & Software. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–29. https://doi.org/10.1109/HCS55958.2022.9895631

[25] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 1–14. https://doi.org/gnqdc9

[26] Pau San Juan, Pedro Alonso-Jordá, and Enrique S. Quintana-Ortí. 2021. High Performance and Energy Efficient Integer Matrix Multiplication for Deep Learning. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, Valladolid, Spain, 122–125. https://doi.org/10.1109/PDP52278.2021.00027

[27] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2009. A characterization and analysis of PTX kernels. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Austin, TX, USA, 3–12. https://doi.org/10.1109/IISWC.2009.5306801

[28] Andrew Kerr, Duane Merrill, Julien Demouth, John Tran, Naila Farooqui, Markus Tavenrath, Vince Schuster, Eddie Gornish, Jerry Zheng, and Bageshri Sathe. 2018. CUTLASS: CUDA Template Library for Dense Linear Algebra at All Levels and Scales. *NVIDIA GPU Technology Conference (GTC)* s8854 (Mar 2018).

[29] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907 [cs.LG]

[30] Takumi Kondo, Yoshihiro Maeda, and Norishige Fukushima. 2021. Accelerating Finite Impulse Response Filtering Using Tensor Cores. In *2021 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, Tokyo, Japan, 74–79.

[31] Thorsten Kurth, Shashank Subramanian, Peter Harrington, Jaideep Pathak, Morteza Mardani, David Hall, Andrea Miele, Karthik Kashinath, and Animashree Anandkumar. 2022. FourCastNet: Accelerating Global High-Resolution Weather Forecasting using Adaptive Fourier Neural Operators. arXiv:2208.05419 [physics.ao-ph]

[32] Ang Li and Simon Su. 2021. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2021), 1878–1891. https://doi.org/10.1109/TPDS.2020.3045828

[33] Cheng Li, Abdul Dakkak, Jinjun Xiong, and Wen-mei Hwu. 2020. Benanza: Automatic μBenchmark Generation to Compute "Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, New Orleans, LA, USA, 440–450. https://doi.org/10.1109/IPDPS47924.2020.00053

[34] Guangli Li, Jingling Xue, Lei Liu, Xueying Wang, Xiu Ma, Xiao Dong, Jiansong Li, and Xiaobing Feng. 2021. Unleashing the Low-Precision Computation Potential of Tensor Cores on GPUs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Seoul, South Korea, 90–102. https://doi.org/10.1109/CGO51591.2021.9370335

[35] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal Loss for Dense Object Detection. arXiv:1708.02002 [cs.CV]

[36] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2015. Microsoft COCO: Common Objects in Context. arXiv:1405.0312 [cs.CV]

[37] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia* (Firenze, Italy) *(MM '10)*. ACM, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

[38] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Vancouver, BC, Canada, 522–531. https://doi.org/10.1109/IPDPSW.2018.00091

[39] Matt Martineau, Patrick Atkinson, and Simon McIntosh-Smith. 2019. Benchmarking the NVIDIA V100 GPU and Tensor Cores. In *Euro-Par 2018: Parallel Processing Workshops*, Gabriele Mencagli, Dora B. Heras, Valeria Cardellini, Emiliano Casalicchio, Emmanuel Jeannot, Felix Wolf, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Laura Ricci, Marco Beccuti, Laura Antonelli, José Daniel Garcia Sanchez, and Stephen L. Scott (Eds.). Springer International Publishing, Cham, 444–455.

[40] Paulius Micikevicius. 2018. Training Neural Networks with Mixed Precision: Theory and Practice. *NVIDIA GPU Technology Conference* s8923 (Mar 2018).

[41] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*. Open Review, Vancouver, BC, Canada, 12 pages. https://openreview.net/forum?id=r1gs9JgRZ

[42] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. arXiv:2209.05433 [cs.LG]

[43] Cristóbal A. Navarro, Roberto Carrasco, Ricardo J. Barrientos, Javier A. Riquelme, and Raimundo Vega. 2021. GPU Tensor Cores for Fast Arithmetic Reductions. *IEEE Transactions on Parallel and Distributed Systems* 32, 1 (2021), 72–84. https://doi.org/10.1109/TPDS.2020.3011893

[44] NVIDIA Corporation. 2016. *NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU.* Whitepaper. NVIDIA Corporation, US.

[45] NVIDIA Corporation. 2017. *NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU.* Whitepaper. NVIDIA Corporation, US.

[46] NVIDIA Corporation. 2018. *NVIDIA Turing GPU Architecture: Graphics Reinvented.* Whitepaper. NVIDIA Corporation, US.

[47] NVIDIA Corporation. 2019. NVIDIA CUDA Toolkit Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/#nvprof.

[48] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture: Unprecedented Acceleration at Every Scale.* Whitepaper. NVIDIA Corporation, US.

[49] NVIDIA Corporation. 2022. *NVIDIA H100 Tensor Core GPU Architecture: Exceptional Performance, Scalability, and Security for The Data Center.* Whitepaper. NVIDIA Corporation, US.

[50] NVIDIA Corporation. 2023. cublasAxpyEx(). https://docs.nvidia.com/cuda/cublas/#cublasaxpyex.

[51] NVIDIA Corporation. 2023. GNMT v2 For PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Translation/GNMT

[52] NVIDIA Corporation. 2023. Kernel Profiling Guide. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#kernel-replay.

[53] NVIDIA Corporation. 2023. Nsight Compute CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[54] NVIDIA Corporation. 2023. *NVIDIA Ada GPU Architecture: Designed to deliver outstanding gaming and creating, professional graphics, AI, and compute performance.* Whitepaper. NVIDIA Corporation, US.

[55] NVIDIA Corporation. 2023. *NVIDIA L40S Unparalleled AI and graphics performance for the data center.* Datasheet. NVIDIA Corporation, US.

[56] NVIDIA Corporation. 2023. Parallel Thread Execution ISA Version 8.2. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[57] NVIDIA Corporation. 2023. ResNet-50 v1.5 for MXNet. https://github.com/NVIDIA/DeepLearningExamples/tree/master/MxNet/Classification/RN50v1.5

[58] NVIDIA Corporation. 2023. UNet Industrial Defect Segmentation for TensorFlow. https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Segmentation/UNet_Industrial

[59] Hiroyuki Ootomo and Rio Yokota. 2022. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance. *The International Journal of High Performance Computing Applications* 36, 4 (2022), 475–491. https://doi.org/10.1177/10943420221090256

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG]

[61] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, San Diego, CA, USA, 58–70. https://doi.org/ggtwps

[62] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2016. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. arXiv:1506.01497 [cs.CV]

[63] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. 2021. Multi-Scale Attributed Node Embedding. *Journal of Complex Networks* 9, 2 (2021), 22 pages.

[64] Valerie Sarge. 2020. Tensor Core Performance: The Ultimate Guide. *NVIDIA GPU Technology Conference (GTC)* s21929 (May 2020).

[65] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Magazine* 29, 3 (Sep. 2008), 93. https://doi.org/10.1609/aimag.v29i3.2157

[66] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible Software Profiling of GPU Architectures. *SIGARCH Comput. Archit. News* 43, 3S (jun 2015), 185–197. https://doi.org/10.1145/2872887.2750375

[67] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. 2023. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2023), 246–261. https://doi.org/10.1109/TPDS.2022.3217824

[68] Yufei Sun, Long Zheng, Qinggang Wang, Xiangyu Ye, Yu Huang, Pengcheng Yao, Xiaofei Liao, and Hai Jin. 2022. Accelerating Sparse Deep Neural Network Inference Using GPU Tensor Cores. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–7. https://doi.org/10.1109/HPEC55821.2022.9926300

[69] Mingxing Tan and Quoc V. Le. 2020. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv:1905.11946 [cs.LG]

[70] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. CUTLASS.

[71] The Linux Foundation. 2023. Reproducibility. https://pytorch.org/docs/stable/notes/randomness.html.

[72] Philippe Tillet and David Cox. 2017. Input-Aware Auto-Tuning of Compute-Bound HPC Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 43, 12 pages. https://doi.org/mdr3

[73] Gaurav Verma, Yashi Gupta, Abid M. Malik, and Barbara Chapman. 2021. Performance Evaluation of Deep Learning Compilers for Edge Inference. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Portland, OR, USA, 858–865. https://doi.org/mdr2

[74] Shibo Wang and Pankaj Kanwar. 2019. BFloat16: The secret to high performance on Cloud TPUs. *Google Cloud Blog* 4 (2019), 1 pages.

[75] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: Accelerating Quantized Graph Neural Networks via GPU Tensor Core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. ACM, New York, NY, USA, 107–119. https://doi.org/10.1145/3503221.3508408

[76] Yuxin Wang, Qiang Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Kaiyong Zhao, and Xiaowen Chu. 2020. Benchmarking the Performance and Energy Efficiency of AI Accelerators for AI Training. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, Melbourne, Australia, 744–751. https://doi.org/10.1109/CCGrid49817.2020.00-15

[77] Zhihao Wen, Yuan Fang, and Zemin Liu. 2021. Meta-Inductive Node Classification across Graphs. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, Canada) *(SIGIR '21)*. ACM, New York, NY, USA, 1219–1228. https://doi.org/mdr4

[78] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[79] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. 2015. ArrayFire - A high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire

[80] Charlene Yang. 2015. Berkeley CS Roofline Toolkit. https://bitbucket.org/berkeleylab/cs-roofline-toolkit.

[81] Zhiwei Yang, Lu Lu, and Ruimin Wang. 2022. A batched GEMM optimization framework for deep learning. *The Journal of Supercomputing* 78, 11 (March 2022), 13393–13408. https://doi.org/10.1007/s11227-022-04336-3

[82] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. arXiv:1907.04931 [cs.LG]

[83] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2018. Graph Convolutional Networks: Algorithms, Applications and Open Challenges. In *Computational Data and Social Networks*, Xuemin Chen, Arunabha Sen, Wei Wayne Li, and My T. Thai (Eds.). Springer International Publishing, Cham, 79–91.

[84] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2019. Graph convolutional networks: a comprehensive review. *Computational Social Networks* 6, 1 (Nov. 2019), 23 pages. https://doi.org/10.1186/s40649-019-0069-y

[85] Zhihui Zhang, Jingwen Leng, Lingxiao Ma, Youshan Miao, Chao Li, and Minyi Guo. 2020. Architectural Implications of Graph Neural Networks. *IEEE Computer Architecture Letters* 19, 1 (2020), 59–62. https://doi.org/10.1109/LCA.2020.2988991

[86] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Storrs, CT, USA, 214–225. https://doi.org/10.1109/IISWC53511.2021.00029

[87] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23)*. ACM, New York, NY, USA, 153–164. https://doi.org/10.1145/3543622.3573210

MalleTrain: Deep Neural Networks Training on Unfillable Supercomputer Nodes

Xiaolong Ma
University of Nevada, Reno
Reno, Nevada, USA
xiaolongm@nevada.unr.edu

Feng Yan
University of Houston
Houston, Texas, USA
fyan5@central.uh.edu

Lei Yang
University of Nevada, Reno
Reno, Nevada, USA
leiy@unr.edu

Ian Foster
Argonne National Laboratory
Lemont, Illinois, USA
University of Chicago
Chicago, Illinois, USA
foster@anl.gov

Michael E. Papka
Argonne National Laboratory
Lemont, Illinois, USA
University of Illinois Chicago
Chicago, Illinois, USA
papka@anl.gov

Zhengchun Liu
Argonne National Laboratory
Lemont, Illinois, USA
zhengchun.liu@anl.gov

Rajkumar Kettimuthu
Argonne National Laboratory
Lemont, Illinois, USA
University of Chicago
Chicago, Illinois, USA
kettimuthu@anl.gov

ABSTRACT

First-come first-serve scheduling can result in substantial (up to 10%) of transiently idle nodes on supercomputers. Recognizing that such unfilled nodes are well-suited for deep neural network (DNN) training, due to the flexible nature of DNN training tasks, Liu et al. proposed that the re-scaling DNN training tasks to fit gaps in schedules be formulated as a mixed-integer linear programming (MILP) problem, and demonstrated via simulation the potential benefits of the approach. Here, we introduce `MalleTrain`, a system that provides the first practical implementation of this approach and that furthermore generalizes it by allowing it to be used even for DNN training applications for which model information is unknown before runtime. Key to this latter innovation is the use of a lightweight online job profiling advisor (`JPA`) to collect critical scalability information for DNN jobs—information that it then employs to optimize resource allocations dynamically, in real time. We describe the `MalleTrain` architecture and present the results of a detailed experimental evaluation on a supercomputer GPU cluster and several representative DNN training workloads, including neural architecture search and hyperparameter optimization. Our results not only confirm the practical feasibility of leveraging idle supercomputer nodes for DNN training but improve significantly on prior results, improving training throughput by up to 22.3% without requiring users to provide job scalability information.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645035

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; **Distributed algorithms**; • **Computer systems organization** → **Distributed architectures**.

KEYWORDS

Deep Neural Network, Distributed Deep Learning Training, Supercomputer, Scheduling, Resource Management

ACM Reference Format:
Xiaolong Ma, Feng Yan, Lei Yang, Ian Foster, Michael E. Papka, Zhengchun Liu, and Rajkumar Kettimuthu. 2024. MalleTrain: Deep Neural Networks Training on Unfillable Supercomputer Nodes. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645035

1 INTRODUCTION

Batch-scheduled high-performance computing (HPC) systems typically maintain a queue of runnable jobs, with the order in which queued jobs are run being determined by resource scheduling policies established by administrators to meet higher-level goals. For example, the largest supercomputers often implement policies to encourage capability computing, wherein they prioritize large jobs that cannot run elsewhere. Other criteria, such as job wait time and recent usage by a user or group, may also be considered when determining job priorities. But regardless of policy goals, the fact that jobs are typically given exclusive access to a fixed number of nodes while running means that nodes will be idle whenever the number of free nodes is less than the number needed to run the next job (as identified by policy).

Backfilling [27], a method by which lower-priority, shorter, and/or smaller jobs are run on idle resources ahead of higher-priority jobs

as long as they do not delay the start time of the higher-priority jobs, can reduce, but not eliminate, inefficiencies, which can be substantial. For example, in 2012, a comprehensive analysis of a 12-month workload trace of the Kraken supercomputer showed an average utilization of 94% [37]; a four-year study of the Blue Waters system revealed that monthly utilization rarely exceeded 80%. Jones et al. [18]; and other studies have reported utilizations of around 90% [9, 25, 29]. These numbers can represent thousands of idle GPUs on large supercomputers.

One approach to enhancing utilization in such environments is to devise new approaches for structuring applications in `malleable forms` and for mapping these malleable applications to supercomputer resources. A malleable computation adapts its degree of parallelism at runtime in response to external requests [15], for example by using checkpointing for semi-automated stop/restart [34] or specialized languages and libraries [6, 11–13]. If well managed, malleable applications can improve system utilization and scheduling efficiency and reduce average response times, compared with unmalleable jobs. However, to realize these benefits, (a) malleable jobs need to be able to adapt dynamically to changing resource allocations and (b) job schedulers must be able to expand or shrink their resources to improve system utilization, throughput, and/or response times.

In practice, the rigid nature of both commonly used programming models like MPI and many current schedulers makes writing and running malleable applications a daunting task, which is why few malleable applications exist.

One intriguing source of malleable applications is deep neural network (DNN) training. DNNs are being employed widely in scientific computing [8, 10, 19, 21, 24, 26], and DNN training is becoming a major workload in today's supercomputers. Furthermore, deep learning frameworks such as AdaptDL [31], PyTorch TorchElastic [28], and Elastic Horovod [33] enable scaling up and down the number of workers dynamically during training at modest cost without requiring a restart. A DNN training job is divided into many smaller tasks (mini-steps) that can be fitted into node×time gaps in a supercomputer computing infrastructure. In other words, DNN training workloads can in principle be structured as malleable computations. However, practical realization of this malleability requires the ability to 1) determine, quickly and accurately, what mini-steps should be configured for different batch queue states, and 2) assign resources and computations to run those mini-steps.

Liu et al. recently showed how, given knowledge of scheduler state, the task of identifying mini-steps can be formulated as a deterministic mixed-integer linear programming–based resource allocation problem [25]. However, while they showed via simulation that this "FreeTrain" approach could construct effective schedules for real scheduler traces, they did not address the second task just listed, by providing a practical implementation of their proposed approach. This is a significant obstacle to the effective realization of malleable DNN training due to the need for several system components to coordinate and interact coherently: idle resource management, job progress monitoring, resource negotiation, and resource allocation. These components as well as their coordination are not readily available in today's job schedulers that were designed for unmalleable computing tasks.

A second deficiency of the FreeTrain approach is that it requires users to provide accurate scaling information, such as measured throughput when using different numbers of nodes for DNN training jobs. Providing this information is a substantial challenge because in many modern DNN training workflows, such as neural architecture search (NAS) [23, 32, 39] and hyperparameter tuning (HPO) [22], jobs are generated on the fly based on results produced in previous iterations by methods such as reinforcement learning [39] and Bayesian optimization [14]. Thus, even experienced DNN experts do not know all the model details beforehand, let alone their scalability characteristics.

In the work reported here, we propose and demonstrate solutions to the two obstacles to the practical realization of malleable DNN training just noted. First, we present a malleable DNN training system architecture, `MalleTrain`, which achieves the efficient coordination of the required idle resource management, job progress monitoring, resource negotiation, and resource allocation functions. For instance, in order to make malleable scheduling decisions, the Resource Allocator must first get information about unfillable nodes from the batch scheduler (e.g., PBS [16] or Slurm [36]), profiling information from a profiler, and current running and waiting DNN jobs from the job monitor; then, it needs to control a DNN scaling framework (e.g., Elastic Horovod) to execute the scheduling decisions. Throughout this process, it must also avoid negative impacts on jobs submitted to the main batch scheduler.

Second, we address the challenge of obtaining accurate scaling information by introducing a lightweight job profiling advisor (`JPA`) to obtain automatically the information required for making resource management decisions. `JPA` runs experiments whenever a DNN training task starts, according to a schedule that minimizes associated costs by taking advantage of the fact that removing a node is faster than adding a node in distributed DNN training. By thus obtaining accurate job information at modest cost, `JPA` permits the MILP to make more accurate decisions, with significant benefits in practice. We conducted extensive simulation evaluations using workloads from production supercomputer clusters, alongside experiments on a smaller cluster with synthetic logs derived from real Summit cluster logs. Our findings indicate that the more accurate information provided by `JPA` allows `MalleTrain` to achieve performance improvements of up to 22.3% relative to FreeTrain. In addition, it permits the scheduling of malleable DNN training applications, such as NAS and HPO, for which no performance information may be available.

This paper thus makes three important contributions. First, we propose a system architecture for running malleable DNNs on supercomputers, and implement `MalleTrain` according to this architecture. Second, we propose a lightweight online profiler that employs an inverse-order profiling method to obtain accurate scalability information for dynamic DNN jobs. Third, we present results from both simulations with supercomputer traces and real-world executions on a cluster using synthetic traces that demonstrate the efficiency of these methods in harnessing previously idle nodes for DNN training—and thus the feasibility of using what may often be 10% or more of previously unfillable supercomputer nodes for large-scale DNN training.

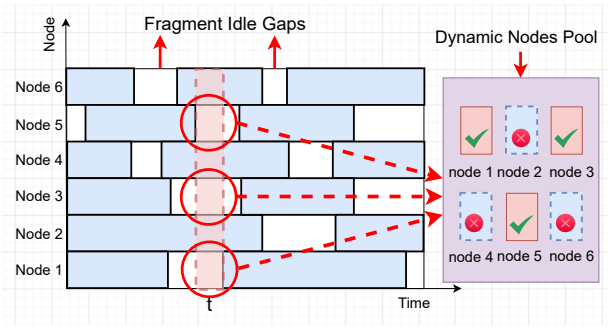


Figure 1: Illustration of dynamic fragment resources on a portion of a cluster. At time t, there are three idle nodes in the `MalleTrain` resource pool.

2 BACKGROUND

We present background information on the methods used by cloud providers to support malleability, fragmented resources in HPC, FreeTrain, and HPC network topologies.

2.1 Cloud-Preemptable Instances

Cloud providers such as AWS [1], Google [2], and Azure [3] make preemptable compute capacity available at a reduced cost via mechanisms such as AWS Spot Instances. For AWS, Spot Instances enable strategic utilization of surplus capacity; for users, they provide an opportunity to reduce their cloud expenses. To make use of such resources, however, users must be flexibile in their application runtime and tolerance for interruptions.

Spot Instances are particularly well suited for certain noncritical tasks such as data analysis, batch processing, and background operations. As noted, their costs are typically lower than for regular instances; on the other hand, they do not provide a time guarantee, introducing the possibility of unexpected interruption due to the cloud provider reclaiming running instances. To mitigate the potential impact of such interruptions, cloud providers often grant a brief time window and prior notification to clients. This advance notice enables clients to reconfigure their workload distribution, effectively rebalancing the workload across available resources. By reallocating tasks and resources in response to an impending reclamation, clients can minimize disruptions and maintain a good level of user experience. Spot Instance resources in cloud environments resemble the preemptible HPC nodes addressed by `MalleTrain`.

2.2 Fragment Resources on HPC

As explored in recent research [9, 25, 29], leadership supercomputer clusters such as Mira, Theta, and Summit exhibit utilization rates of around 90%. Considering the substantial scale of these leadership supercomputer clusters, the unutilized resources become a significant concern. To put this in perspective, 10% idle capacity corresponds to 460 nodes on the 4608-node Summit and more than 1000 nodes on the 10,624-node Aurora.

Resource allocation within supercomputer clusters is typically managed by main schedulers such as Slurm [36] or PBS [16]. These schedulers administer multiple queues to prioritize resource assignments for user requests. As depicted in Figure 1, inevitable fragmentary resources emerge. These fragments may not always be backfilled, and (a portion of them) may remain unassigned. However, these seemingly negligible fragments are well suited for scalable and/or fault-tolerant workloads. The nature of these unassigned fragment resources resembles that of Spot VMs, as discussed in §2.1. In subsequent sections we will refer to these fragment resources within supercomputer clusters as *preemptible nodes*. Their allocation timing lacks guarantees, rendering them unsuitable for typical fixed-size supercomputer workloads. Nonetheless, the paradigm of malleable applications, exemplified by DNN training, aligns seamlessly with this computational context. This suitability is underscored by several key factors: (1) DNN training demands substantial time and computational resources; (2) the distributed data-parallel training paradigm is inherently scalable; (3) leading DNN training frameworks, such as Horovod Elastic [33] and TorchElastic [28], adeptly support elastic training; and (4) DNN training often involves exhaustive searches for optimal neural network architectures and hyperparameters, consuming extensive computational resources.

The objective of `MalleTrain` is to empower users to effectively leverage the unfilled fragments in supercomputers. Some supercomputers have a preemptable queue (the jobs submitted to this queue may be preempted anytime) explicitly to encourage the use of the unfilled nodes. A preemptable queue can be designated for `MalleTrain` to which the users will submit adaptable DNN training jobs. `MalleTrain` will optimally manage the allocation of unfilled nodes by dynamically expanding and shrinking these adaptable DNN jobs. To incentivize the adoption of this preemptable queue, benefits such as reduced charges, in terms of either monetary cost or node-time consumption, can be extended to users.

Table 1: Queue types and their characteristics. *Queue* is the queue type name on the Polaris cluster, the *Min* and *Max* columns give minimum/maximum number of nodes, and time, allowed per job request, and *Priority* is the priority for jobs in the queue.

	Nodes		Time		
Queue	Min	Max	Min	Max	Priority
debug	1	2	5 min	1 hr	debug
debug-scaling	1	10	5 min	1 hr	debug
demand	1	56	5 min	1 hr	High
prod	10	496	5 min	24 hr	High
preemptable	1	10	5 min	72 hr	Low

Table 1 displays the different queue types in the Argonne Leadership Computing Facility (ALCF) Polaris cluster [4]. The low job priority means that nodes allocated for the job in the preemptable queue could be reclaimed.

2.3 FreeTrain

As noted earlier, FreeTrain [25] introduces an approach to dynamically allocating idle resources in which nodes and running job information are taken as inputs and user-defined metrics such as throughput or scalability are adopted as optimization objectives. By formulating the problem using MILP, FreeTrain is able to compute

an optimal allocation of idle resources to DNN training jobs, subject to constraints such as allowed job size, feasible resource allocation, job scale information, and job migration overhead.

However, several practical challenges must be overcome before this approach can be realized into a production environment:

(1) **Expecting users to provide specific runtime job details can be a significant burden to users.** The MILP algorithm requires users to supply precise job-specific information, such as model training throughput and scalability, since these details serve as essential inputs for the optimization process. This requirement will significantly increase the burden on users.

(2) **Job runtimes often correlate closely with specific hardware capability and configurations.** Thus, to attain accurate job runtime information, users would have to prerun their jobs under nearly identical system settings and hardware configurations. However, this approach would be prohibitively time-consuming and resource-consuming for most supercomputer users.

(3) **In some cases, heuristic algorithms rely on current models to predict future executions, making it impractical to preprofile all potential models.** The majority of HPO/NAS algorithms are heuristic [14, 23, 32, 39], which implies that the models to be evaluated are not predetermined until the current models have completed their execution. Thus, users will not be able to provide accurate job runtime information, a situation that will lead to an invalid resource allocation plan and will largely downgrade the performance of the system.

To overcome these challenges, an intelligent online profiling mechanism is needed. Such a mechanism should accurately collect job runtime information while minimizing disruptions to the regular execution of jobs.

`MalleTrain` also employs MILP to do the allocation optimization but emphasizes practical deployment aspects in supercomputer clusters. JPA can be integrated seamlessly into the workflow, orchestrating automatic profiling and obviating the need for manual input. As a result, the profiling procedure becomes an inherent facet of the process, efficiently alleviating the user from the need to provide such details beforehand. This dynamic profiling process operates in real time, eliminating the need to halt any ongoing jobs. While the profiling phase may occasionally lead to suboptimal cluster performance, we mitigate potential overhead through the implementation of a carefully designed online profiling mechanism. Thus our design is able to obtain accurate profiling information without excessive operational costs.

2.4 Topology

The network topology in a supercomputer cluster plays an important role in facilitating efficient communication, seamless data transfer, and effective management of network resources. Today, the dragonfly [20] and fat-tree [7] topologies are widely utilized in supercomputer clusters due to their ability to deliver high bandwidth and low latency. These features make them adept at meeting the demanding requirements of modern high-performance computing environments. The Polaris cluster and upcoming Aurora cluster in the ALCF both use the dragonfly network topology, and the Summit cluster uses fat-tree. A major concern for fragmented idle resources in a supercomputer is that such resources will often

Figure 2: Example of fragment resources distribution on Polaris (27th in the TOP500 supercomputer list on Nov. 2023). Red stars mark fragmented idle resources scattered on the cluster. Note: For clarity in presentation, the figure depicts a majority of the cluster rather than its entirety.

be scattered and distant from each other, as shown in Figure 2. Each color represents a job; the nodes with same color were allocated to the same job. To fully utilize the inter connection bandwidth and reduce the latency, schedulers tend to assign the nodes into the same group or make them close to each other. For fragmented resources, however, usually the nodes are scattered into different topology groups. This scattering will have two major impacts. First, long distance usually means more hops are needed, which means the connections could have a higher fluctuation and cause a downgrade in the DNN training performance. Second, long distance could increase the end-to-end latency and cause more network resource contentions. We perform extensive evaluation and show that the topology is not a critical bottleneck for the design of `MalleTrain`.

3 SYSTEM DESIGN AND REALIZATION

`MalleTrain` manages the residual resources of a supercomputer cluster, in other words, those that at any particular moment have not been allocated directly by the main scheduler. Two major challenges for `MalleTrain` arise in utilizing such residual resources: (1) their availability varies dynamically, and (2) they are preemptible. The `MalleTrain` design enables these resources to be utilized fully for parallel DNN training. `MalleTrain` seamlessly integrates with mainstream schedulers such as Slurm or PBS on supercomputer clusters. It operates without impacting the main scheduler, exclusively controlling the non-trivial, dynamic, residual resources that the main scheduler cannot utilize.

3.1 System Architecture Overview

Figure 3 shows the `MalleTrain` architecture and its five primary components, which we describe in the following:

Scavenger detects and collects idle nodes from the main job scheduler for `MalleTrain`. Two primary approaches could be employed: an event-driven mechanism, whereby the main scheduler alerts `MalleTrain` to idle nodes, or a proactive strategy, in which Scavenger periodically polls to find available unused (but preemptible when the main scheduler needs them) resources. The latter approach, preferred for its autonomy, requires no additional action

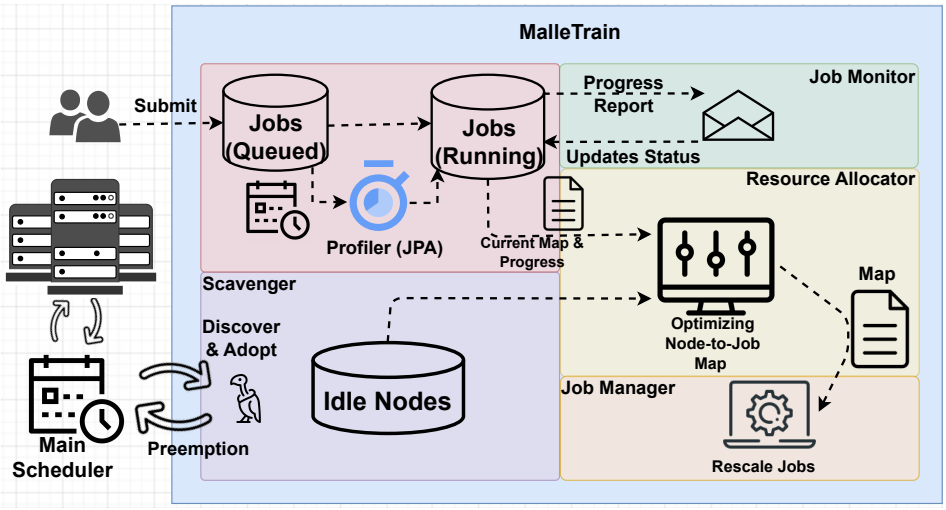


Figure 3: Schematic of the `MalleTrain` architecture. Scavenger adopts idle nodes, Resource Allocator determines a map of nodes to jobs, Job Manager rescales jobs according to the map, Job Monitor tracks job progress, and Job Profiling Advisor manages the online profiling process.

from the main scheduler, ensuring seamless and efficient use of idle nodes by `MalleTrain`.

Resource Allocator maps nodes to DNN jobs in such a way as to optimize a given metric such as throughput or scaling efficiency. The allocation task can be formulated as a mathematical programming problem. In this paper we adopt the formulation of Liu et al. [25] for resource allocation. The Resource Allocator is event-driven, with four types of events being considered: new nodes joining `MalleTrain`, nodes being recalled by the batch scheduler (i.e., the corresponding jobs are preempted), arrival of new `MalleTrain` jobs, and `MalleTrain` jobs completing.

Job Manager manages all jobs and implements the jobs-to-nodes mapping made by the Resource Allocator.

Job Monitor tracks job progress by consuming (current global batch size, timestamp) records generated by DNN training jobs via one line of `MalleTrain`-supplied code added to the training loop. The Monitor module then computes the current throughput as well as the cost incurred for each rescale operation and updates that information in a job records table to be used by the Resource Allocator.

Job Profiling Advisor manages the online profiling process, as described in §3.3. The JPA is an independent component that starts work before the job entering the Resource Allocator.

When nodes cannot be backfilled by the main scheduler, they are redirected to the Scavenger for utilization. Jobs submitted by users to `MalleTrain` await the availability of nodes. As nodes become available, the jobs at the front of the queue commence execution. The running jobs transmit progress updates to the Job Monitor via a socket client. The system's architecture ensures continuous reporting of both cluster node statuses and job execution information to the Resource Allocator. The Allocator then employs MILP based on the current job distribution and number of nodes in the Scavenger. The MILP algorithm devises a strategic plan, which is represented by a map and subsequently conveyed to the Job Manager. The Job Manager then implements this plan to adjust resources accordingly. The events described in §3.2 will trigger the Resource Allocator to run MILP and generate a new adjustment plan.

Users are provided with the option to explicitly indicate whether their job requires profiling. If so, the JPA consults with the Resource Allocator to assess the availability of necessary node resources for profiling. Should resources be insufficient, the jobs are returned to the queue. Conversely, if adequate resources are available for profiling, the job proceeds through the profiling process. This process uniquely involves an inverse order of node numbers; further details are given in §3.3. When the profiling process is done, the profiled job information will be an input to the MILP to find the optimal allocation.

3.2 Event-Driven Resource Adjustment

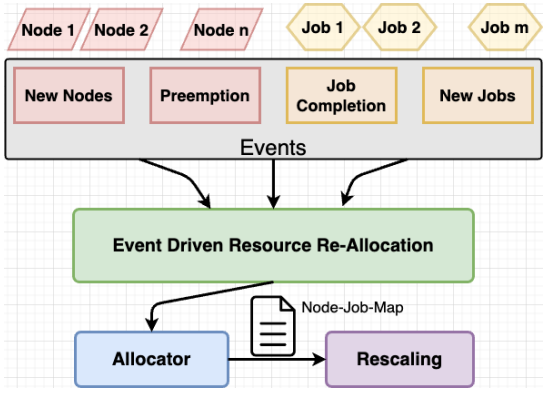


Figure 4: Event-driven resource allocation process.

Our event-driven resource management architecture is shown in Figure 4. There are four types of events:

New Nodes indicates that one or more nodes have become available to `MalleTrain`.

Preemption can be initiated at any time by the main scheduler without any prior notification. The jobs being run by `MalleTrain` on the preempted nodes are terminated and the nodes returned to the main batch scheduler.

Job Completion. `MalleTrain` picks a maximum of the top (first come, first serve, FCFS) jobs from its queue to prevent excessive hunger of low-priority jobs (e.g., low-throughput jobs when sample processed per second is the target to optimize). All the selected jobs are launched by `MalleTrain` via spawning a process using a `subprocess` module of Python in a nonblocking fashion. The exit/completion of a job is thus notified from the Job Monitor module of `MalleTrain`.

A **New Jobs** event can trigger resource allocation only when the number of currently running jobs, $Njrun$, is less than the jobs number threshold allowed in `MalleTrain`, $Pjmax$. When more than one job is submitted as a batch (e.g., grid search of a hyperparameter search), $Pjmax - Njrun$ jobs will be added to the running list as a batch to reduce the rescaling cost. When the number of arrving jobs $Njarrive$ is larger than $Pjmax - Njrun$, the $Njarrive - (Pjmax - Njrun)$ jobs will be put into the FCFS queue for future execution.

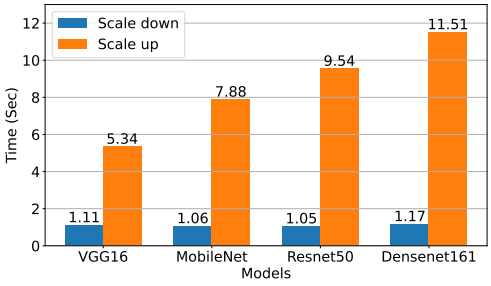
Table 2: Example jobs-to-nodes map, as determined by MILP. Each row corresponds to a job, with scale given by the sum of the cells in the row; each column corresponds to a node, with at most one cell in the column with value 1 indicating the job to which the node is allocated.

	N_1	N_2	N_3	N_4	N_5	N_6	N_7	...	...	N_n
J_1	0	0	1	0	0	0	1	0	0	0
J_2	0	0	0	0	0	0	0	0	1	1
...	1	0	0	1	1	0	0	0	0	0
J_4	0	1	0	0	0	1	0	1	0	0

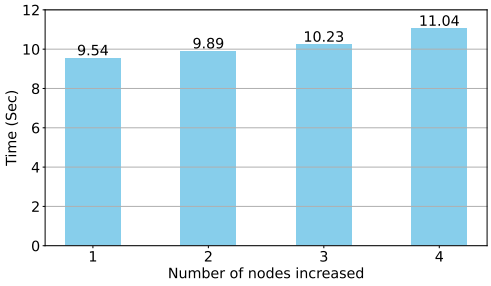
The node-job map shows the allocation plan, and Table 2 demonstrates an example map of the allocation plan. The MILP optimizer takes the input and gives a new node-job map to the Allocator to do the reallocation. We give more details in §3.3.

3.3 Job Profiling Advisor

In contrast to traditional profiling methods that necessitate dedicated resources, our online profiling process is integrated into the training process. This approach ensures the uninterrupted operation of worker processes during profiling. The strategic design of node adjustment sequences, as depicted in Figure 6, to avoid scale-up operations, effectively minimizes additional overhead. Each job is equipped with a lightweight reporter (socket client), responsible for reporting job progress to the Job Monitor (socket server). This approach facilitates the automatic aggregation by the Job Monitor of the training process information that is then used for optimization purposes. Consequently, the JPA is enabled to make precise and timely adjustments, thereby maximizing resource utilization.



(a) 1-node scale-up and scale-down costs.



(b) Scale-up times vs. number of nodes: ResNet-50.

Figure 5: Rescaling overhead costs on Polaris A100 GPU nodes: (a) Time to scale up and down a single node, for different models; (b) Time to scale up different numbers of nodes, for ResNet-50 model.

We noted in §2.3 the necessity for online profiling in order to permit accurate MILP solutions and to handle tasks for which profile information is not available before their execution. Here we shift focus to an in-depth examination of the design elements of JPA. In our proposed design the profiling function runs concurrently with jobs. Thus we want it to be:

Prompt, meaning that it processes profiling events rapidly so as to ensure efficient utilization of profiling information, and furthermore completes rapidly so as to minimize overhead and limit disruption to other tasks;

Fair, meaning that its design incorporates principles of fairness, and that in instances where job interruption is unavoidable, a Least Recently Used (LRU) strategy is employed to ensure equitable distribution of interruptions; and

Efficient, meaning that it prioritizes minimal disruption to other tasks, adhering to two key principles: (1) avoiding the interruption of multiple jobs simultaneously and (2) preventing the complete cessation of any single job.

Accurate MILP requires that we know, or can rapidly determine, the time that will be required to run any training mini-task on any possible number of nodes. While obtaining this information may sound intractable, in practice the regular nature of DNN computations makes it feasible to obtain good estimates. As in FreeTrain, we assume a fixed per-node minibatch size (when training, we employ a learning rate scheduler to adjust learning rate according to the

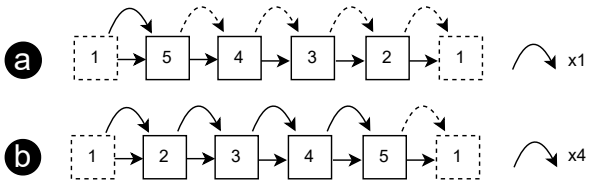


Figure 6: Inverse-order rescaling sequence. The solid curve represents scale-up and the dashed curve scale-down. JPA aims to minimize the number of scale-up operations in order to reduce overhead.

global batch size [17, 38]). We then need simply to measure the time per epoch for that minibatch size on different numbers of nodes, from a specified minimum to a specified maximum.

A useful optimization when performing those measurements derives from the observation that, as shown in Figure 5a, the cost of scaling up is consistently multiple times greater than that of scaling down. Furthermore, Figure 5b illustrates that the overhead incurred during scale-up remains relatively constant regardless of the number of nodes involved; even as the number of nodes increases, the increase in scale-up time is marginal. Consequently, in our profiling of the rescaling process, we should minimize the need for scaling up and prioritize scaling down wherever feasible. As an example, consider the two situations illustrated in Figure 6. If the initial number of nodes is 1 and the objective is to profile nodes 2, 3, 4, and 5, we may either: (a) scale up directly to 5 nodes and then scale down to 1, thereby gathering scalability data for all nodes using a single scale-up operation, or (b) incrementally scale up from 1 to 5, which requires four separate scale-up operations. The first approach is significantly more efficient than the first, since it requires only one scale-up operation.

The JPA architecture (Figure 7) resembles that of `MalleTrain` but with several distinctions: (1) JPA exclusively processes new job events, since only these require profiling; in contrast, the trainer instance accepts multiple events, as described in §3.2. (2) The node adjustment in JPA is decided by our profiling algorithm instead of by the MILP program. Users retain the discretion to decide whether their jobs undergo profiling. Upon receiving a profiling request from a user, a profiling event is triggered, which initiates a process whereby the Resource Allocator assesses the availability of sufficient resources for profiling. If resources are deemed adequate, a profiling job is started, temporarily preempting nodes from other jobs. Upon completion of profiling, the MILP process is engaged to make adjustments based on the newly collected information. The gathered scale information is then reported and recorded by the job manager, contributing to future optimization efforts.

3.4 Cluster Configuration

MILP is an NP-hard problem and the cost of the MILP computation required to determine a mapping of jobs to idle nodes scales rapidly with the number of runnable jobs and available nodes. Thus, it can be preferable to partition a supercomputer into disjoint subsets and run multiple trainers in parallel, one per subset. This approach restricts the maximum number of nodes to which any one job can scale, but has the advantages of reducing delays due to training and of permitting different trainers to optimize for different metrics appropriate for different task types, such as computer vision models and language models.

With multiple trainers, the question arises of whether it is advisable from a performance perspective to run more than one on a single node. Our preliminary investigation into the effects of running multiple MILP processes concurrently on the same node revealed that the processing time begins to increase only when the number of concurrent trainers exceeds the number of cores, as illustrated in Figure 8. This suggests that deploying multiple trainers and running the associated MILP processes concurrently on a single head node can diminish overheads without adversely affecting the performance of standard jobs.

4 EVALUATION AND DISCUSSION

We conducted an extensive experimental evaluation to validate the effectiveness and robustness of our framework with real logs of supercomputer clusters. We also validated `MalleTrain` on a small cluster in a real production environment.

4.1 Experiment Setup

We examined trace logs from two supercomputers listed in the TOP500 as of November 2023: Summit, ranked 7th, and Polaris, ranked 27th [5]. The Summit log spans 14 days from February 10 to February 24, 2021, while the Polaris log covers a 7-month duration, from January 1 to July 28, 2023.

Figure 9 depicts event traces from the Summit and Polaris supercomputers. We see that Polaris has more shorter gaps than Summit, with indeed over 50% of its event gaps being shorter than 10 seconds. A key factor contributing to this difference is Summit's policy favoring large jobs. Such jobs generally have longer durations, leading to fewer but more extended resource occupations. Conversely, without a similar policy favoring large jobs, Polaris experiences more frequent, shorter gaps between events due to the prevalence of smaller jobs. However, because of the unavailability of idle node data for the Polaris cluster, we focus on Summit trace data in our log replay simulation evaluation. Figure 10, which shows idle nodes on Summit over a two-week period, shows that the number of idle nodes varies significantly over time.

While plugging `MalleTrain` into the batch scheduler of a real supercomputer would permit accurate evaluation in a real system, we would lose the ability to reproduce the same trace with different strategies, including the baseline allocation policy, for comparative research. Therefore, we instead generate representative traces and replay them on the real system for our experimental evaluation. In contrast to the simulation-based evaluation, experiments here do not rely on any performance modeling: they run the DNN training task on real supercomputer nodes.

A challenge for `MalleTrain` is to optimally utilize fragmented node×time resources to meet a user-specified metric (e.g., throughput in terms of samples processed per second, resource utilization/scaling efficiency). We synthesize traces that are independent and identically distributed with real traces from supercomputers. Figure 11 compares node idle gap lengths from real Summit scheduler logs vs. our synthetic traces. We see that the distribution of

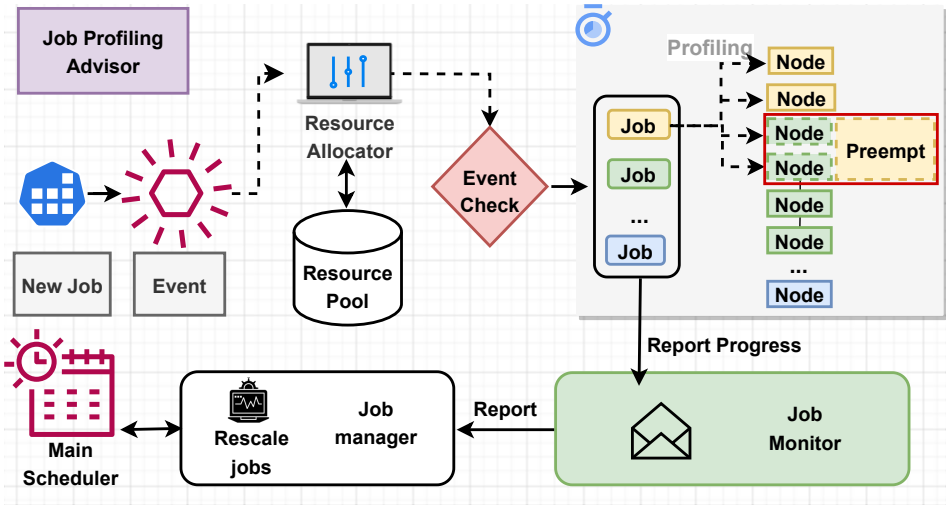


Figure 7: Online profiling process.

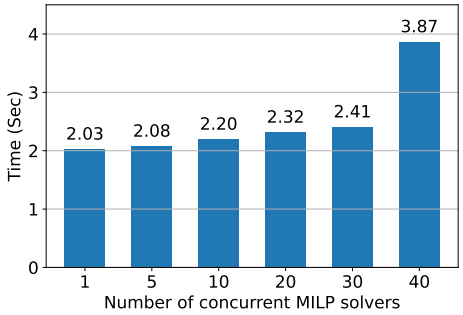


Figure 8: Average time taken for an example MILP computation as the numbers of concurrent MILP computations performed on a single 32-core head node scales from 1 to 40.

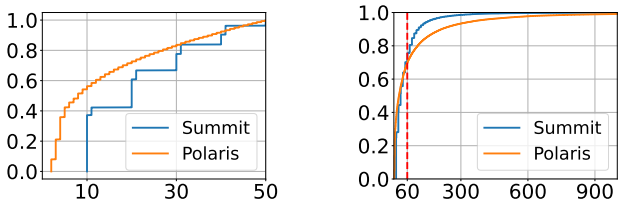


Figure 9: Cumulative histograms of idle gap counts on Summit and Polaris, for short gaps (0–50 secs: left) and longer gaps (0–3600 secs: right). Polaris has more shorter gaps (≤60 secs) while Summit has more gaps in the range from 60 to 600 secs.

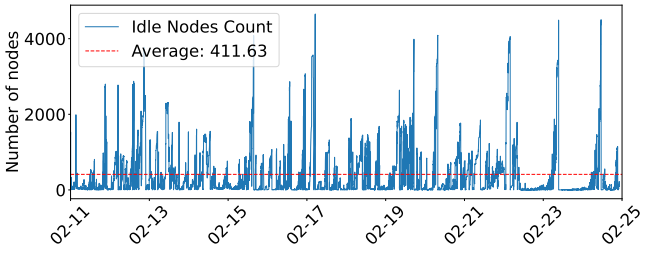


Figure 10: Idle nodes on Summit over two-week period.

synthetic traces is close to those of the real logs, confirming the representativeness of our synthetic traces.

4.1.1 Workload. NASBench101 [35] is a neural architecture search (NAS) benchmark dataset created to permit systematic, reproducible, and accessible evaluation of NAS algorithms. It was introduced to address the challenges associated with the high computational cost of evaluating NAS algorithms, which traditionally require training thousands of neural network architectures from scratch to find the most efficient one for a given task. We conducted our experiment within the search space of NASBench101, which comprises 423,624 computationally unique neural architectures. The image size for our training is 224×224×3. We use randomly generated tensors instead of the real dataset to remove the potential I/O impact on our experiments. We note that our focus here is not on the accuracy of the models but rather on assessing throughput and scalability. Varieties of deep learning models that do the HPO tasks were also evaluated in the same context as the NAS workload; the models were randomly selected from models listed in Figure 14.

4.1.2 Testbed. We conducted experiments on a 32-node cluster in which each node is equipped with four A100 GPUs. The GPUs are interconnected via NVLink within each node, and nodes are connected via InfiniBand. The synthesized traces, as depicted in Figure 11, were instrumental in simulating the preemptive actions undertaken by the main scheduler.

4.2 Performance Evaluation

We conducted experiments to benchmark our system against the FreeTrain framework for preemptible resource allocation on HPC

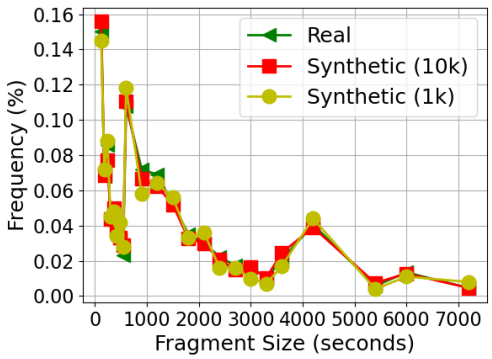


Figure 11: Comparison histogram of fragment length between real logs and synthetic. Synthetic (10k) shows the statistics for 10k fragments, and Synthetic (1k) shows the statistics for 1k fragments. The synthetic traces keep the same distribution as that of the real log.

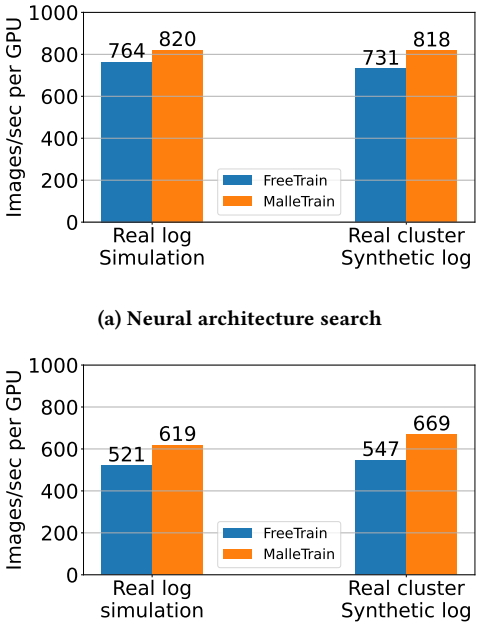
clusters. Our evaluation comprised NAS and HPO training workloads. Notably, the NAS workload exhibited more variability in training speed and scalability compared with HPO tasks.

Our primary metric for comparison was the overall training throughput of the system. We ran both frameworks under identical workloads to ensure a fair comparison. For the NAS model sampling process, we randomly selected models. To maintain consistency, we set the same seed value for both frameworks, ensuring that the sequence of model training remained identical across the experiments. We conducted the simulation with the two-week log and conducted the experiments for 12 hours with the synthetic trace. The average throughput is shown in Figure 12 with the NAS workload and HPO workload. We see that `MalleTrain` outperforms FreeTrain in various settings.

4.3 Topology Impact Analysis

The dynamic and randomly scattered nature of fragmented resources across the cluster raises concerns about potential declines in the overall performance of training jobs. To address these concerns, we conducted experiments on the Polaris cluster with the dragonfly network topology. Our study involved comparing the performance of nodes confined within a single dragonfly group versus those distributed across multiple dragonfly groups. Figure 13 shows that the physical distribution of nodes, whether scattered or closely situated, has minimal impact on NAS/HPO DNN training speed. Figure 14 indicates robust scalability of models even at the 32-node level, each node equipped with 4 NVIDIA A100 GPUs, encompassing a total of 128 A100 GPUs.

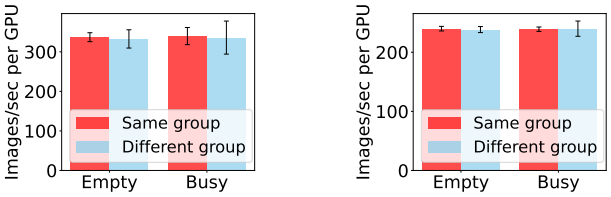
The underlying reasons for these observations are multifaceted. First, leadership-class supercomputer clusters are typically outfitted with high-performance network devices. For instance, Polaris is equipped with the HPE Slingshot 11 interconnect, offering up to 200 Gb/s point-to-point bandwidth. Second, the networking infrastructure in these clusters is often highly overprovisioned, mitigating network contention among applications running on different nodes.



(a) Neural architecture search

(b) Hyperparameter optimization

Figure 12: FreeTrain vs. `MalleTrain` performance for the NAS and HPO applications, as measured both with real logs on a simulator and synthetic logs on a real cluster.



(a) ResNet-50 **(b) VGG-19**

Figure 13: We analyzed training performance for sample `MalleTrain` jobs under four different scenarios: *Same Group, Empty* (where all nodes are located within the same Dragonfly group and the cabinet is empty), *Same Group, Busy* (where all nodes are within the same Dragonfly group but are collocated with other jobs), *Different Group, Empty* (where nodes are distributed across two Dragonfly groups with the two cabinets empty), and *Different Group, Busy* (where nodes are distributed across two Dragonfly groups and collocated with other jobs). The results demonstrate consistent training speeds for both models across all scenarios. The error bars for the *Different Group, Busy* scenario reveal higher variances in training speed, indicating fluctuations occur primarily in this scenario. However, the average training speed remains consistent despite these fluctuations.

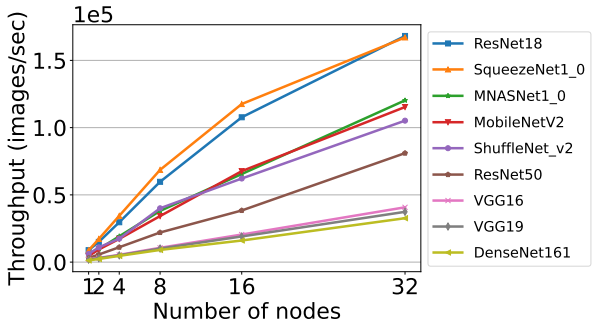


Figure 14: Trend analysis of model scalability on 32 Polaris nodes, each with 4 A100 GPUs.

Third, modern distributed deep learning frameworks, such as PyTorch [28] and Horovod [33], effectively overlap computing and communication tasks. This overlapping functionality reduces the network's impact on training speed, thereby diminishing the sensitivity to network conditions.

5 RELATED WORK

We have already referred to the pioneering work of Liu et al. on FreeTrain [25], while noting also that certain assumptions and strict requirements make it fall short in the real production environment. FreeTrain heavily relies on users to provide accurate runtime information from training jobs, which increases the burden to the users, making it impractical for use. Indeed, in some widely used heuristic NAS/HPO algorithms, FreeTrain has to guess a configuration or provide information solely based on user experience; the inaccurate or out-of-date information might largely downgrade the overall performance of the MILP optimization algorithm. In contrast, `MalleTrain` integrates automatic profiling components into the process and doing the profiling automatically.

Pollux [30] is a resource-adaptive DNN training and scheduling framework designed to efficiently rearrange distributed deep learning processes, particularly in dynamic-resource environments such as shared clusters and cloud infrastructures. This framework employs Kubernetes for efficient scheduling, rescaling, and reconfiguring of job batch sizes and learning rates, thus maximizing training performance and optimizing resource utilization. Pollux operates on a fixed-size cluster, however, whereas `MalleTrain` can handle dynamically varying cluster sizes.

6 CONCLUSION

We have introduced `MalleTrain`, a system that we demonstrate can employ idle fragmented nodes on batch-scheduled HPC systems for large-scale DNN training. `MalleTrain` defines a workable architecture for efficient use of such idle nodes, and via its job profiling advisor, which efficiently gathers accurate job execution data at runtime with minimal interference to ongoing tasks, enables idle nodes to be employed efficiently even for dynamic workloads such as neural architecture search and hyperparameter optimization. Detailed performance studies involving both simulations and experiments validate the effectiveness of the approach and show that `MalleTrain` achieves >20% more training throughput than was reported, on the basis of simulation studies alone, for a precursor system. `MalleTrain` thus opens up the feasibility of both improving the utilization of large HPC systems and increasing the resources delivered to DNN applications. Moreover, the methodologies developed in this study have potential applications beyond their current scope. They could be adapted, for example, to infrastructure management tasks, such as scheduling in Kubernetes clusters and other cloud computing platforms.

ACKNOWLEDGMENT

We are grateful to the reviewers for their valuable feedback and comments and Gail Pieper for helping us edit our paper. This work was sponsored in part by NSF grant CAREER-2048044 and by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] 2023. https://aws.amazon.com/ec2/spot/. Accessed: 2023-10-23.
[2] 2023. https://cloud.google.com/spot-vms. Accessed: 2023-10-23.
[3] 2023. https://azure.microsoft.com/en-us/products/virtual-machines/spot/. Accessed: 2023-10-23.
[4] 2023. https://www.alcf.anl.gov/polaris. Accessed: 2023-10-23.
[5] 2023. https://www.top500.org/lists/top500/2023/11/. Accessed: 2023-11-15.
[6] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, et al. 2014. Parallel programming with migratable objects: Charm++ in practice. In *SC'14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 647–658.
[7] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review* 38, 4 (2008), 63–74.
[8] Ahsan Ali, Hemant Sharma, Rajkumar Kettimuthu, Peter Kenesei, Dennis Trujillo, Antonino Miceli, Ian Foster, Ryan Coffee, Jana Thayer, and Zhengchun Liu. 2022. fairDMS: Rapid model training by data and model reuse. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 394–405.
[9] William Allcock, Paul Rich, Yuping Fan, and Zhiling Lan. 2017. Experience and practice of batch scheduling on Leadership Supercomputers at Argonne. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1–24.
[10] Selin Aslan, Zhengchun Liu, Viktor Nikitin, Tekin Bicer, Sven Leyffer, and Doga Gursoy. 2020. Distributed optimization with tunable learned priors for robust ptycho-tomography. *arXiv preprint arXiv:2009.09498* (2020).
[11] Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. 2015. Malleable Invasive Applications. In *Software Engineering (Workshops)*. 123–126.
[12] Ewa Deelman, Karan Vahi, Mats Rynge, Rajiv Mayani, Rafael Ferreira da Silva, George Papadimitriou, and Miron Livny. 2019. The evolution of the Pegasus workflow management software. *Computing in Science & Engineering* 21, 4 (2019), 22–36.
[13] Travis Desell, Kaoutar El Maghraoui, and Carlos A Varela. 2007. Malleable applications for scalable high performance computing. *Cluster Computing* 10, 3 (2007), 323–337.
[14] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*. PMLR, 1437–1446.
[15] Dror G Feitelson and Larry Rudolph. 1996. Toward convergence in job schedulers for parallel supercomputers. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1–26.
[16] Hanhua Feng, Vishal Misra, and Dan Rubenstein. 2007. PBS: a unified priority-based scheduler. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems*. 203–214.
[17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[18] Matthew D Jones, Joseph P White, Martins Innus, Robert L DeLeon, Nikolay Simakov, Jeffrey T Palmer, Steven M Gallo, Thomas R Furlani, Michael Showerman, Robert Brunner, et al. 2017. Workload analysis of Blue Waters. *arXiv preprint arXiv:1703.00924* (2017).

[19] Julian Kates-Harbeck, Alexey Svyatkovskiy, and William Tang. 2019. Predicting disruptive instabilities in controlled fusion plasmas through deep learning. *Nature* 568, 7753 (2019), 526–531.

[20] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. 2008. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 77–88.

[21] Christopher Kleman, Shoaib Anwar, Zhengchun Liu, Jiaqi Gong, Xishi Zhu, Austin Yunker, Rajkumar Kettimuthu, and Jiaze He. 2023. Full Waveform Inversion-Based Ultrasound Computed Tomography Acceleration Using Two-Dimensional Convolutional Neural Networks. *Journal of Nondestructive Evaluation, Diagnostics and Prognostics of Engineering Systems* 6, 4 (2023), 041004.

[22] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-Tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A system for massively parallel hyperparameter tuning. *Proceedings of Machine Learning and Systems* 2 (2020), 230–246.

[23] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[24] Zhengchun Liu, Tekin Bicer, Rajkumar Kettimuthu, and Ian Foster. 2019. Deep learning accelerated light source experiments. In *IEEE/ACM Third Workshop on Deep Learning on Supercomputers*. IEEE, 20–28.

[25] Zhengchun Liu, Rajkumar Kettimuthu, Michael E Papka, and Ian Foster. 2023. FreeTrain: A Framework to Utilize Unused Supercomputer Nodes for Training Neural Networks. In *IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 299–310.

[26] Zhengchun Liu, Hemant Sharma, Jun-Sang Park, Peter Kenesei, Jonathan Almer, Rajkumar Kettimuthu, and Ian Foster. 2020. BraggNN: Fast X-ray Bragg Peak Analysis Using Deep Learning. *arXiv preprint arXiv:2008.08198* (2020).

[27] Ahuva W. Mu'alem and Dror G. Feitelson. 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems* 12, 6 (2001), 529–543.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[29] Tirthak Patel, Zhengchun Liu, Rajkumar Kettimuthu, Paul Rich, William Allcock, and Devesh Tiwari. 2020. Job Characteristics on Large-Scale Systems: Long-Term Analysis, Quantification and Implications. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 1186–1202.

[30] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning.. In *OSDI*, Vol. 21. 1–18.

[31] Aurick Qiao, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2020. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. *arXiv preprint arXiv:2008.12260* (2020).

[32] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 33. 4780–4789.

[33] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[34] Sathish S Vadhiyar and Jack J Dongarra. 2003. SRS: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters* 13, 02 (2003), 291–312.

[35] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. 2019. NAS-Bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*. PMLR, 7105–7114.

[36] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple Linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.

[37] Haihang You and Hao Zhang. 2012. Comprehensive workload analysis and modeling of a petascale supercomputer. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 253–271.

[38] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet training in minutes. In *47th International Conference on Parallel Processing*. 1–10.

[39] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.

Leftovers for LLaMA

Ravi Kumar Singh
TCS Research
ravik.singh2@tcs.com

Likhith Bandamudi
TCS Research
likhith.bandamudi1@tcs.com

Shruti Kunde
TCS Research
shruti.kunde@tcs.com

Mayank Mishra
TCS Research
mishra.m@tcs.com

Rekha Singhal
TCS Research
rekha.singhal@tcs.com

ABSTRACT

In recent years, large language models (LLMs) have become pervasive in our day-to-day lives, with enterprises utilizing their services for a wide range of NLP-based applications. The exponential growth in the size of LLMs poses a significant challenge for efficiently utilizing these models for inference tasks, which require a substantial amount of memory and compute. Enterprises often possess multiple resources (workers, nodes, servers) with unused (leftover) capacity, providing an opportunity to address this challenge by distributing large models across these resources.

Recent work such as Petals, provides a platform for distributing LLM models in a cluster of resources. Petals require that users use their discretion to distribute blocks on a given cluster, consequently leading to a non-optimal placement of blocks. In this paper, we propose LLaMPS - a large language model placement system that aims to optimize the placement of transformer blocks on the available enterprise resources, by utilizing the leftover capacity of the worker nodes. Our approach considers leftover memory capacity along with available CPU cores, when distributing transformer blocks optimally across worker nodes. Furthermore, we enhance the scalability of the system by maximizing the number of clients that can be served concurrently. We validate the efficacy of our approach by conducting extensive experiments using open-source large language models - BLOOM (1b, 3b, and 7b parameters), Falcon, and LLaMA. Our experiments demonstrate that LLaMPS facilitates optimal placement of transformer blocks by utilizing leftover resources, thus enabling enterprise-level deployment of large language models.

CCS CONCEPTS

• **Computing methodologies** → **Distributed artificial intelligence**.

KEYWORDS

LLMs, Leftover capacity, Distributed inference, Optimal block placement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05…$15.00
https://doi.org/10.1145/3629526.3645045

ACM Reference Format:
Ravi Kumar Singh, Likhith Bandamudi, Shruti Kunde, Mayank Mishra, and Rekha Singhal. 2024. Leftovers for LLaMA. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3629526.3645045

1 INTRODUCTION

Large Language Models (LLMs) have become pervasive, finding widespread applications in businesses, such as natural language processing [11] and recommender systems [9] [15] for inference tasks. LLMs [14] [10] have been instrumental in enabling decision-making and facilitating various aspects of day-to-day business operations in enterprises. With the continuous evolution of LLMs, a notable challenge that has emerged is their increasing size. As LLMs grow in scale [5], they require a substantial amount of memory and compute resources for effective deployment. These requirements may pose a constraint in enterprises having limited infrastructure, impeding their ability to fully leverage the potential of LLMs for inference tasks.

Many businesses deploy LLM-powered chatbots [16] [4] for customer support. These chatbots require substantial memory and computational resources for handling natural language queries efficiently. Recommender systems [13] powered by LLMs, like those used by streaming services, need to process vast amounts of user data and perform complex language-based recommendations, demanding considerable computational resources. Besides LLMs have been steadily increasing in size over time to improve their performance, and this growth has led to greater resource requirements. Larger models generally require more memory and computational power for efficient inference. As businesses grow and their workloads increase, they may find it challenging to scale their inferencing infrastructure to meet the demand. This can lead to performance bottlenecks and delays.

To address these challenges enterprises do have multiple options such as leveraging cloud-based LLM services to help mitigate some of the infrastructure and resource constraints. Cloud providers offer pre-configured LLM models and scalable infrastructure, which may not necessarily be a cost-effective solution.

Enterprises however have a latent opportunity comprising of multiple worker (server) nodes having leftover capacity in terms of memory and processing cores. We believe this capacity can be effectively utilized to deploy LLMs in a distributed manner. There are some efforts in the literature such as Deepspeed [18], Petals [6] etc that enable the distribution of transformer blocks across multiple workers. DeepSpeed enables distributed fine-tuning and inference

of transformer blocks by on the underlying GPU cluster, by utilizing maximum possible resources. Petals [5] is one such effort that enables the distribution of the transformer blocks across multiple workers for fine-tuning or inference. Petals utilize leftover capacity from the workers in a cluster to seamlessly enable the distribution of transformer blocks across worker nodes. However, Petals requires that users use their discretion to distribute blocks on a given cluster, consequently leading to a non-optimal placement of blocks. This may also lead to sub-optimal system performance in terms of inference latency.

In this paper, we address the above-mentioned challenges by proposing a system that enables optimal placement of transformer blocks in a given cluster. Our contributions are as follows:

- We propose an Optimal block Placement Algorithm (OPA), used in LLaMPS s.t., multiple models may be served on a single server, while optimally utilizing the leftover capacity of workers in the cluster.
- We evaluate the efficacy of LLaMPS on an enterprise CPU-cluster on multiple variants of open source transformer-based large language models, namely BLOOM [23], LLaMA [21] and Falcon [20].

The rest of the paper is structured as follows. Section 2 provides a background for our proposed work. We present the LLaMPS system in Section 3. Our experiment setup is discussed in Section 4. We present results, ablation studies, and related work in Sections 5, 6 and 7. We conclude with directions for future work in Section 8.

2 BACKGROUND

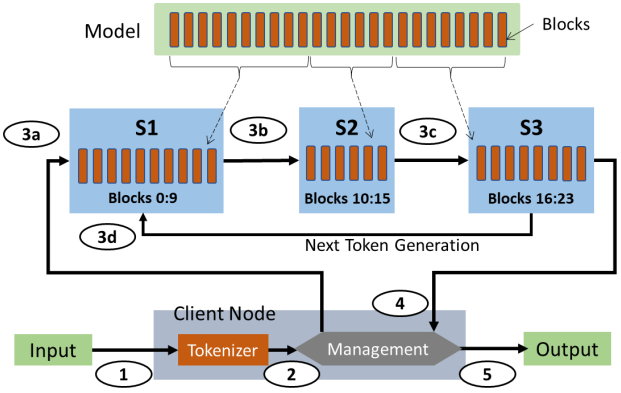


Figure 1: Inference

The resource constraints imposed by the massive scale of transformer models render them unsuitable for deployment on a single server. Consequently, this has been an active area of research for exploring solutions to harness the potential of these models in a distributed manner. One approach to address the resource challenge is to leverage distributed frameworks that enable the allocation of transformer blocks across multiple servers. These frameworks aim to maximize the computational resources available for distribution, thus achieving low-latency inference. However, this strategy, while effective in reducing response times, can lead to sub-optimal utilization of resources. An alternative school of thought advocates for making efficient use of the leftover capacity of servers, ensuring that no computational power goes to waste. One such framework is Petals, which capitalizes on the untapped resources of servers.

As shown in Figure 1 a typical enterprise may have servers s1, s2, and s3, each with different degrees of leftover capacity. Transformer model blocks are distributed across these servers. Server 1 hosts Blocks 0-9, Server 2 handles Blocks 10-15, and Server 3 manages Blocks 16-23. During an inference cycle, the input is initially tokenized at the client node (Steps 1 and 2), and the tokenized input is then relayed to server 1. The input traverses the allocated transformer blocks on server 1, and the intermediate output is sequentially passed to server 2, and so on, until it has traversed all the blocks of the transformer (Steps 3a, 3b, 3c). Finally, the output from the last server in the sequence is transmitted back to the client, where the required output is generated.

Petals, although making effective use of available block capacities, do not inherently guarantee an optimal distribution of these blocks across servers. In response to this limitation, we introduce an Optimal Placement Algorithm (OPA) within the LLaMPS framework, offering an enhanced method for strategically allocating blocks across servers. OPA's primary objective is to accommodate multiple models efficiently on a single server while exploiting any residual server capacity. The LLaMPS system can be seamlessly integrated with various open-source distributed frameworks, providing flexibility and compatibility. In this paper, we assess the performance of the Optimal Placement Algorithm within the LLaMPS framework when applied to the open-source Petals framework. To conduct our evaluation, we established a controlled lab environment within an enterprise setting, featuring a cluster composed of four CPU servers with heterogeneous capacities. Since Petals supports three different Large Language Models (LLMs), namely LLaMa, BLOOM, and Falcon; our analysis encompasses all three LLMs, offering a comprehensive assessment of the algorithm's efficacy across a range of scenarios.

3 LLAMPS

In this section, we present our system LLaMPS. As shown in Figure 2, the LLaMPS system takes the size of the transformer model as input. This input is crucial for accurately calculating the total memory requirement for accommodating the model on the available resources. LLaMPS leverages the residual resource capacity available within an enterprise cluster pool. We quantify this *leftover* capacity in terms of memory and number of cores (i.e., compute capacity). LLaMPS then runs the Optimal Placement Algorithm (OPA) to determine the optimal distribution of transformer blocks across nodes in the enterprise resource cluster. The optimal placement algorithm is designed to maximize the overall leftover capacity within the resource cluster to enable multiple clients to be served concurrently. Once the plan is ready, the transformer model is automatically chunked into blocks and the blocks are distributed to the identified servers. The application is then ready to start the process of inference. In contrast, Petals [5] distributed framework requires the user to decide the exact distribution of blocks across the available servers. This distribution may lead to a sub-optimal utilization of the enterprise resources. The Optimal Placement Algorithm of LLaMPS ensures that the leftover capacity of the enterprise

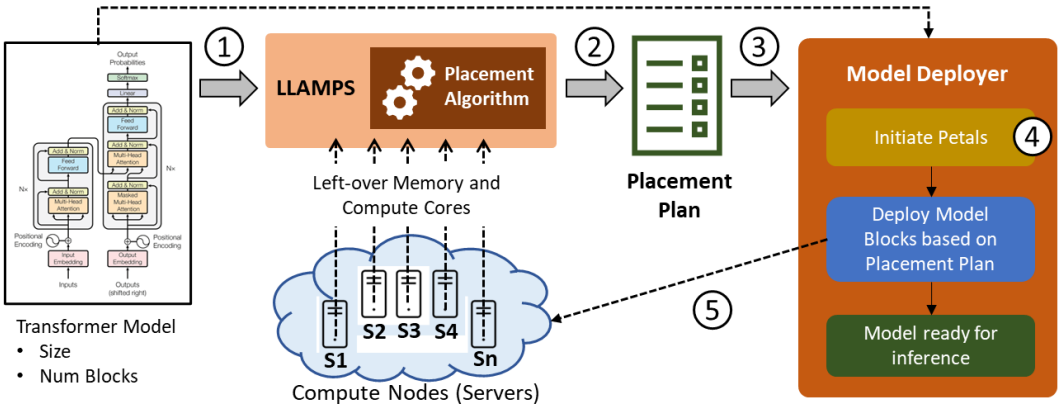


Figure 2: Architecture of LLaMPS

cluster is optimally utilized, subsequently serving multiple clients and enabling multiple models to be supported on individual worker resources in the enterprise cluster.

The Optimal Placement Algorithm is outlined in algorithm 1. As depicted in the Optimal Placement Algorithm 1, LLaMPS determines the available memory and cores on each worker in the enterprise cluster. Next, OPA calculates the leftover memory capacity of each worker (Step 2). Since we use the open-source Petals framework for the distribution of the transformer block, OPA needs to deduct the memory overhead imposed by Petals on the worker node. OPA is a generic approach that enables optimal distribution of transformer blocks on an enterprise cluster, hence the underlying distributed framework can be replaced by any other framework. It is expected that the memory overhead imposed by the framework will be deducted from the available memory at a worker node. Once the available memory and cores at each worker are determined, OPA then follows the Analytical Hierarchy Process (AHP) technique [12] - a popular approach for systematically determining weights based on the preferences of decision-makers. AHP helps in structuring the decision-making process and deriving relative weights through pairwise comparisons. Weights are based on preference scores. Since we compare OPA with GMA which is a memory-based approach, we have assigned higher preference to memory. In LLaMPS we have two objectives - memory and cores. Decision-makers assign a preference score to indicate how much one objective (memory) is more important than another (cores) (Step 6). After collecting preference scores, we normalize them to form a consistent comparison matrix (Step 7). Then, the weighted average of each objective is calculated based on the normalized comparison matrix (Step 8,9). The final weighted scores for each worker node are determined as shown in Step 10. Once the scores for all worker nodes are determined, the list of worker nodes is sorted in descending order based on the scores (Step 12). OPA then determines the number of blocks to be loaded on each server, by dividing the leftover memory at each worker by the size of each transformer block (Step 13). The blocks are then loaded on each server and the Petals framework is leveraged to perform downstream tasks.

Algorithm 1 OPA - Optimal Placement Algorithm

Require:
- TB_s - Size of transformer model block
- P_o - Petals Framework Overhead
- S - Cluster of enterprise workers
- AM_S - Available memory for each worker
- AC_S - Available cores for each worker
- $Nb(S)$ - Number of blocks on a worker

Ensure: Optimal block placement on servers

1: **for** each worker in the cluster $S_i \subseteq S$ **do**
2: $LM(S_i) \leftarrow (AM_{S_i} - P_o)$ ▷ Left over memory capacity
3: $AC(S_i) \leftarrow$ Available cores of worker
4: **end for**
5: **for** each worker in the cluster $S_i \subseteq S$ **do**
6: Assign preference scores for memory and cores
7: $M[Scores_i] \leftarrow$ Normalized matrix of scores
8: $W_{m_i} \leftarrow$ Weighted memory score using $M[Scores_i]$
9: $W_{c_i} \leftarrow$ Weighted core score using $M[Scores_i]$
10: $[W_{Score_i}] \leftarrow (W_m \times LM(S_i)) + (W_{c_i} \times AC(S_i))$
11: **end for**
12: $S[W_{Score_i}]_{sorted} \leftarrow$ List of sorted servers based on scores
13: Assign blocks to each server from $S[W_{Score_i}]_{sorted}$ list s.t. $Nb(S_i) \leftarrow LM(S_i)/TB_s$
14: Distribute blocks on each server for inference

Algorithm 2 GMA - Greedy Memory Algorithm

Require:
- TB_s - Size of transformer model block
- P_o - Petals Framework Overhead
- S - Cluster of enterprise workers
- AM_S - Available memory for each worker
- $Nb(S)$ - Number of blocks on a worker

Ensure: Optimal block placement on servers

1: **for** each worker in the cluster $S_i \subseteq S$ **do**
2: $LM(S_i) \leftarrow (AM_{S_i} - P_o)$ ▷ Left over memory capacity
3: **end for**
4: $[LM(S_i)]_{sorted} \leftarrow$ List of sorted servers by leftover memory
5: Assign blocks to each server from $[LM(S_i)]_{sorted}$ list s.t. $Nb(S_i) \leftarrow LM(S_i)/TB_s$
6: Distribute blocks on each server for inference

Table 1: Regression Model Comparison: Predicting Petals' Memory Overhead

Model	MSE	R-Squared	Petals' Memory Overhead Equation
Linear Regression	263.7	0.773	$1.88 \cdot \text{Number of Blocks} + 0.60 \cdot \text{Model parameters} - 15.05$
LGBM Regression	722.25	0.394	$73443.43 \cdot \text{Number of Blocks} + 148798.14 \cdot \text{Model parameters}$
Polynomial Regression	6.74	0.994	$0.03 \cdot \text{Number of Blocks} \cdot \text{Model parameters} + 0.65 \cdot (\text{Model parameters})^2 + 0.14$
Decision Tree Regression	19.75	0.998	Decision Tree

Algorithm 2 depicts GMA - Greedy Memory Algorithm. The GMA determines the leftover memory capacity (Step 2). A user is more likely to use a greedy memory approach to select resources for block placement. Hence we have used GMA as the baseline algorithm in our experiments. However, GMA does not guarantee optimal block placement while OPA does. Then it sorts the list of servers in the descending order of memory capacity (Step 5). The blocks are then assigned to the servers in the sorted order of the list based on the leftover memory capacity of each worker (Step 5). GMA does not take into consideration the cores when assigning blocks which may lead to a sub-optimal placement of blocks on the servers, subsequently affecting system performance adversely. We present detailed experiments and ablation studies to compare the OPA algorithm of LLaMPS with GMA.

4 EXPERIMENT SETUP

We designed multiple experiments to validate the efficacy of the optimal placement algorithm of LLaMPS. We aimed to place transformer blocks in an enterprise cluster while optimally utilizing the leftover capacity of the workers in the cluster. We believe that the optimal placement of blocks enables serving a maximum number of possible clients simultaneously within an enterprise. We now present some details of our experiment setup. Our experiment setup comprises multiple large language models, and worker nodes having heterogeneous configurations within an enterprise cluster.

4.1 Large Language Models

Our experiments are conducted on 3 different open-source models, namely BLOOM [23], LLaMA [21] and Falcon [20]. BLOOM is a multi-lingual large language model that has the ability to generate text in 46 natural languages and 13 programming languages. The BLOOM model comes in multiple variants. In this paper, we have conducted our experiments on 3 variants - namely 560 million, 3 billion, and 7 billion parameter versions of BLOOM. The multiple variants enable us to deeply evaluate the efficacy of our proposed system. Another model we have used to evaluate LLaMPS is LLaMA (Large Language Model Meta AI), which is a family of LLMs released by Meta AI [8]. LLaMA has multiple variants and we conduct our experiments on the 70 billion version of the LLaMA model. Additionally, we also conduct experiments on Falcon [20] which is a generative large language model. Falcon comes in multiple variants and we evaluate LLaMPS on the 40 billion version of Falcon. Through our experiments on multiple variants of multiple open-source models in the literature, we reinforce the efficacy of our proposed system LLaMPS.

4.2 Heterogeneous Enterprise Cluster

We create a set-up in our enterprise lab, utilizing leftover capacities of servers utilized by members of the labs. We pick 5 CPU-based servers and the configuration of these servers is depicted in Table 2. All servers have heterogeneous configurations in terms of operation system, OS version, cores, and memory, which is a typical real-world setup in any enterprise. We leverage this leftover capacity to optimally distribute blocks of large language models such that the number of users can be maximized. Table 2 provides details of the different servers and also distinguishes between the "Client" and four different servers ("Server 1," "Server 2,","Server 3" and "Server 4") and their respective configurations.

4.3 Distributed Framework Overhead

The LLaMPS system functions on the Petals distributed framework, utilizing its services for various tasks, including the creation and initialization of the Distributed Hash Table (DHT) and the loading of transformer blocks. The overhead and impact of these factors on OPA's performance within LLaMPS led us to conduct a series of experiments for a systematic analysis. In this investigation, Petals serves as the open-source foundation for LLaMPS, supporting block distribution and other DHT-related operations. It is important to note that although LLaMPS currently leverages Petals, it is not limited to using this particular framework and can seamlessly integrate with any alternative distributed framework for execution.

To predict the memory overhead linked to Petals (**Petals' Memory Overhead**), we formulated an equation with two variables: the *number of blocks* and *number of model parameters*. This predictive model proves to be valuable in the Block Placement algorithm.

Table 1 displays the outcomes of four regression models—Linear Regression, LGBM Regression, Polynomial Regression, and Decision Tree Regression—applied to 150 data points. Our experiments reveal that the Decision Tree Regression model outperforms the other three, demonstrating an R-squared value of 0.998 and an MSE of 19.75. Consequently, we employed the Decision Tree Regression model for predicting Petals' Memory Overhead. Choosing simpler machine learning models other than DDN and LLMs is a more cost-effective and resource-efficient approach.

Three factors within the distributed framework contribute to the overall overhead: the overhead determined by the Decision Tree Regressor model, and the Attention cache's size. The size of the attention cache is calculated as twice the model's hidden size multiplied by 4096 times the tensor size (4 bytes for CPU). Together, these factors collectively constitute the **Petals Framework Overhead**, quantified as:

$$\text{Petals Memory Overhead} + \text{Attention Cache size} \quad (1)$$

Table 2: Server Configuration Information

Server	OS Version	Kernel	Total Memory(GB)	Cores
Client	Ubuntu 18.04.6	5.4.0	62.8	48
Server 1	Ubuntu 18.04.6	5.4.0	252	56
Server 2	Ubuntu 22.04.3	5.19.0	992	88
Server 3	CentOS 7	3.10.0	504	56
Server 4	CentOS 7.8	3.10.0	256	56

LLaMPS system uses this formula to calculate Petals Framework Overhead for block placement.

5 EXPERIMENTS

We performed various experiments on GMA and OPA by varying the model parameters, servers, clients, cores, batch size, and token length.

Given below are the details of various parameters used in experiments.

- **Model)**: The experiments were conducted using the "bloom-560m," "bloom-3b," "bloom-7b1," "falcon-40b," and "llama-70b" models.
- **Block Distribution**: The "Block_distribution" varied between "[24]" for "bloom-560m" and "[30]" for "bloom-3b" and "bloom-7b1." In the "falcon-40b" and "llama-70b" models, the block distribution was specified as "(32,28)" and "(35,25,20)" respectively.
- **Memory**: The "Memory" column indicates the memory configurations used for each experiment. For example, "[12, 14, 14, 14]" in "bloom-560m" refers to the memory allocated on a single server.
- **Cores**: The "Cores" column specifies the number of cores allocated to each server. It varied for different models and experiments.
- **Selected Servers**: This column lists the selected servers for each experiment. It may include server configurations and the number of servers, like "[14,2]" or "(35,25,20)".
- **Clients**: C1, C2, and C4 represent a number of clients being 1, 2, and 4.
- **Block Execution Time C1, C2, C4**: The "Block Execution Time" columns represent the execution time for different clients (C1, C2, C4) under both the "GMA " and "OPA" approaches.
- **Batch Size**: Batch size refers to the quantity of input data grouped together. In the context of text generation with a transformer model, a batch size of 1 corresponds to generating text based on a single input sentence. Conversely, a batch size of 'n' involves generating text for 'n' input sentences concurrently, sending all 'n' sentences to the transformer simultaneously.
- **Token**: The "number of tokens" refers to the quantity of output tokens produced by a transformer model. For instance, in text generation using a transformer model, having 100 tokens means generating a sequence of 100 words.

5.1 Varying the cores -> pick best the core

Assigning servers based on a combination of memory and cores is a basic premise of OPA in LLaMPS. In our experiments across various versions of the Bloom model (Figure 3), a notable observation emerged: the optimal performance was achieved at 8 cores, i.e., additional cores beyond 8 did not give any significant performance improvement. To validate our observations, we systematically conducted experiments with varying configurations, namely 2, 4, 8, 16, 32, and 56 cores. As shown in Figure 3, beyond the 8-core threshold, the block execution stabilized, revealing a clear knee point in the performance curve. Our results are consistent across multiple flavors of the BLOOM model and also support multiple concurrent clients.

5.2 Model fits on a single server

Table 3 depicts experiments conducted for varying flavors of BLOOM, LLaMA, and the Falcon models. We compare our approach with the GMA for the choice of servers selected for block distribution. We measure the block execution time in each case for single and concurrent clients. We experimented with OPA block placement algorithms on a single server, adjusting client loads across 1, 2, and 4 concurrent clients. These trials were conducted on three distinct models. Notably, when we refer to '1 server,' it implies that all transformer blocks are positioned on a single server, while '1 client' signifies a lone user sending input queries to the server. Conversely, '2 concurrent clients' and '4 concurrent clients' denote two and four concurrent users sending input queries to the server. The servers are named as S1, S2, S3 and S4. There server tuple is represented as S(available memory in GBs, Available cores). Our aim is to ensure the optimal distribution of transformer blocks such that the number of clients served is maximized.

(1) For the **Bloom-560m model**, we utilized four servers—S1, S2, S3,and S4—with heterogeneous memory and core specifications. GMA opted for S2 to distribute all 24 blocks, whereas OPA selected server S4. Comparing block execution times using a batch size of 16 and token size of 10, inference using GMA took 3.09 seconds for 1 client, 4.87 seconds for 2 concurrent clients, and 9.73 seconds for 4 concurrent clients. Conversely, inference using OPA resulted in execution times of 2.02 seconds for 1 client, 3.01 seconds for 2 concurrent clients, and 4.56 seconds for 4 concurrent clients. Notably, the block execution times for OPA with 1, 2, and 4 concurrent clients were lower than GMA with a single client. Therefore, OPA exhibited better performance for 1 and 2 concurrent clients compared to GMA for 1 client, while OPA across

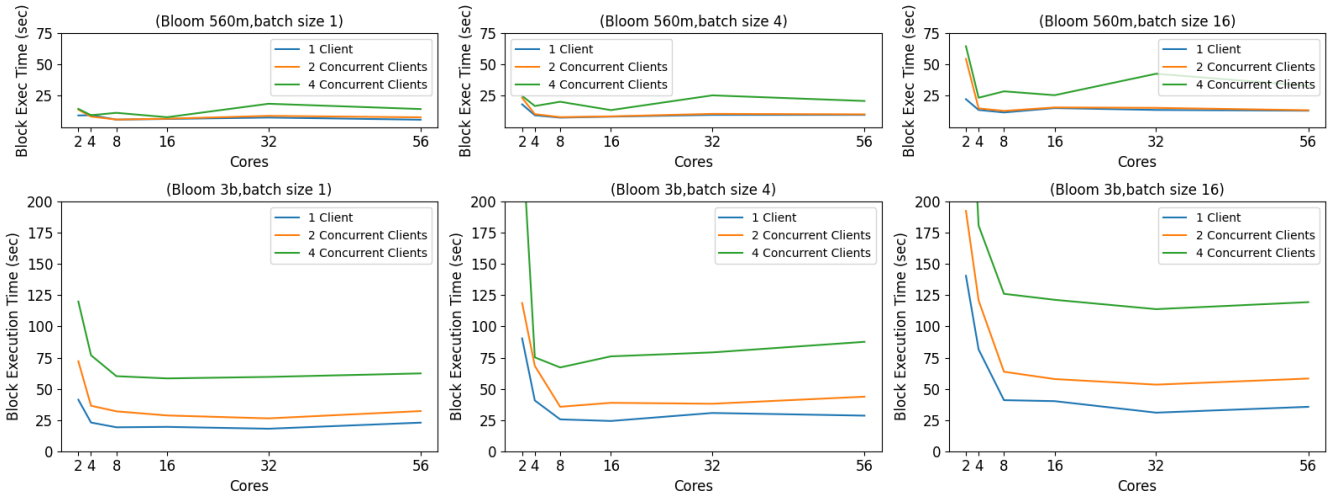


Figure 3: Knee Point on varying Cores

1, 2, and 4 concurrent clients outperformed GMA with 2 concurrent clients.

(2) For the **Bloom-3b model**, similar to the Bloom-560m model, four servers (S1, S2, S3, and S4) were utilized, each with distinct memory and core configurations. GMA selected S2 to distribute all 30 blocks, whereas OPA utilized the S4 server. The block execution times using GMA were 17.01 seconds for 1 client, 28.43 seconds for 2 concurrent clients, and 59.46 seconds for 4 concurrent clients. However, with OPA, the execution times were notably lower: 7.12 seconds for 1 client, 10.12 seconds for 2 concurrent clients, and 20.15 seconds for 4 concurrent clients. Once again, OPA exhibited better performance for 1 and 2 concurrent clients compared to GMA for 1 client, while outperforming the GMA across 1, 2, and 4 concurrent clients.

(3) Lastly, for the **Bloom-7b model**, with the same set of servers (S1, S2, S3, and S4) and their corresponding memory and core specifications, GMA opted for S2 to distribute all 30 blocks, whereas OPA selected the use of the S4 server. The block execution times using the GMA were 45.02 seconds for 1 client, 83.21 seconds for 2 concurrent clients, and 179.01 seconds for 4 concurrent clients. However, employing OPA resulted in significantly lower execution times: 20.1 seconds for 1 client, 34.5 seconds for 2 concurrent clients, and 67.44 seconds for 4 concurrent clients. Similar to the previous models, OPA demonstrated superior performance for 1 and 2 concurrent clients compared to GMA for 1 client, and across 1, 2, and 4 concurrent clients compared to GMA with 2 concurrent clients.

We were unable to conduct experiments with Falcon and LLaMA models as the leftover capacity of any single server was insufficient to load all the blocks of Falcon and LLaMA. In the next set of experiments, we focus on scenarios where the leftover capacity of servers is insufficient even to load the smallest version of the BLOOM model.

5.3 Models fits on 2 servers

As shown in Table3 we experimented with OPA block placement algorithms across two servers, varying client loads with 1, 2, and 4 concurrent clients. These trials encompassed two different models, including three versions of the Bloom model, and the Falcon model. The leftover capacity was insufficient to fit LLaMA on any two given servers and hence LLaMA was not a part of this experiment. When a single server couldn't accommodate all transformer blocks, we utilized two servers. Additionally, with a batch size of 16 and a fixed token length of 10, the OPA algorithm indicated an increase in block execution time as the number of concurrent clients increased, specifically with 2 concurrent clients.

In Table 3

(1) We allocated blocks of the Bloom 560m model using both GMA and OPA methods across servers S1, S2, S3, and S4. S1 has 13.5 GB memory with 8 cores, S2 has 12 GB memory with 2 cores, while S3 and S4 possess 12 GB memory with 4 and 8 cores, respectively. The GMA picked S1 (14 blocks) and S2 (10 blocks), while OPA selected S1 and S4. Block execution times with the GMA were 3.73 seconds for 1 client, 4.49 seconds for 2 concurrent clients, and 8.82 seconds for 4 concurrent clients. In contrast, OPA resulted in notably lower execution times: 2.75 seconds for 1 client, 2.88 seconds for 2 concurrent clients, and 3.11 seconds for 4 concurrent clients. Similar to prior models, OPA demonstrated superior performance for 1 and 2 concurrent clients compared to the GMA for 1 client, and across 1, 2, and 4 concurrent clients compared to the GMA with 1 client.

(2) For the Bloom 3b model, we distributed its blocks using both GMA and OPA methods across servers S1, S2, S3, and S4. S1 offers 17.5 GB memory with 8 cores, S2 has 16 GB memory with 2 cores, while S3 and S4 possess 16 GB memory with 4 and 8 cores, respectively. The GMA selected S1 (16 blocks) and S2 (14 blocks), while OPA chose S1 and S4. Block execution times with the GMA were 18.11 seconds for 1 client,

Table 3: OPA vs GMA on 1,2,3 servers

Bloom 560m - (batch size=16, tokens=10)							
Servers: (GB,cores) [S1(12,2), S2(14,2), S3(14,4), S4(14,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S2)	[24]	3.89	4.87	9.73
OPA	16	10	(S4)	[24]	2.02	3.01	4.56
Servers: (GB,cores) [S1(13.5,8), S2(12,2), S3(12,4), S4(12,8)]							
GMA	16	10	(S1,S2)	[14,10]	3.73	4.49	8.82
OPA	16	10	(S1,S4)	[14,10]	2.43	2.59	3.65
Servers: (GB,cores) [S1(12.15,8), S2(11.65,8), S3(11.15,2), S4(11.15,4), S5(11.15,8)]							
GMA	16	10	(S1,S2,S3)	[12,8,4]	2.75	3.01	3.65
OPA	16	10	(S1,S2,S5)	[12,8,4]	2.68	2.88	3.11
Bloom 3b - (batch size=16, tokens=10)							
Servers: (GB,cores) [S1(23,8), S2(23.5,2), S3(23.5,4), S4(23.5,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S2)	[30]	17.01	28.43	59.46
OPA	16	10	(S4)	[30]	7.12	10.12	20.15
Servers: (GB,cores) [S1(17.5,8), S2(16.7,2), S3(16.7,4), S4(16.7,8)]							
GMA	16	10	(S1,S2)	[16,14]	18.11	28.62	54.44
OPA	16	10	(S1,S4)	[16,14]	8.6	9.97	19.13
Servers: (GB,cores) [S1(16.7,8), S2(15,2), S3(13.3,2), S4(13.3,4), S5(13.3,8)]							
GMA	16	10	(S1,S2,S3)	[14,10,6]	10.93	12.09	14.11
OPA	16	10	(S1,S2,S5)	[14,10,6]	10.28	11.23	13.91
Bloom 7b1 - (batch size=16, tokens=10)							
Servers: (GB,cores) [S1(42,8), S2(44,2), S3(44,4), S4(44,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S2)	[30]	45.02	83.21	179.01
OPA	16	10	(S4)	[30]	20.1	34.5	67.44
Servers: (GB,cores) [S1(31,8), S2(30,2), S3(30,4), S4(30,8)							
GMA	16	10	(S1,S2)	[16,14]	46.78	70.12	160.12
OPA	16	10	(S1,S4)	[16,14]	23.08	24.98	42.01
Servers:(GB,cores) [S1(30,8), S2(26,8), S3(22,2), S4(22,4),S5(22,8)]							
GMA	16	10	(S1,S2,S3)	[14,10,6]	27.79	29.12	38.35
OPA	16	10	(S1,S2,S5)	[14,10,6]	22.12	23.59	36.12
Falcon-40b - (batch size=16, tokens=10)							
Servers:(GB,cores) [S1(98.3,8), S2(88,2), S3(88,4), S4(88,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S1,S2)	[32,28]	491.2	536.12	840.02
OPA	16	10	(S1,S4)	[32,28]	211.37	305.12	682.01
Servers: (GB,cores) [S1(93.2,2), S2(67.5,8), S3(42,2), S4(42,4), S5(42,8)]							
GMA	16	10	(S1,S2,S3)	[30,20,10]	260.12	266.72	352.12
OPA	16	10	(S1,S2,S5)	[30,20,10]	258.19	263.89	345.26
Llama2-70b - (batch size=16, tokens=10)							
Servers: (GB,cores) [S1(155,8), S2(113.5,2), S3(83.5,2), S4(83.5,4),S5(83.5,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S1,S2,S3)	[35,25,20]	396.11	408.12	798.12
OPA	16	10	(S1,S2,S5)	[35,25,20]	279.13	304.11	588.76

28.63 seconds for 2 concurrent clients, and 54.44 seconds for 4 concurrent clients. However, OPA resulted in significantly lower execution times: 8.6 seconds for 1 client, 9.97 seconds for 2 concurrent clients, and 19.13 seconds for 4 concurrent clients. Similar to previous models, OPA exhibited superior performance for 1 and 2 concurrent clients compared to the GMA for 1 client, and across 1, 2, and 4 concurrent clients compared to the GMA with 2 concurrent clients.

(3) In the case of the Bloom 7b1 model, we utilized both GMA and OPA methods to distribute its blocks across servers S1,

S2, S3, and S4. S1 has 31 GB memory with 8 cores, S2 has 30 GB memory with 2 cores, while S3 and S4 have 30 GB memory with 4 and 8 cores, respectively. The GMA opted for S1 (16 blocks) and S2 (14 blocks), whereas OPA chose S1 and S4. Block execution times with the GMA Approach were 46.78 seconds for 1 client, 70.12 seconds for 2 concurrent clients, and 160.12 seconds for 4 concurrent clients. However, OPA resulted in notably lower execution times: 23.1 seconds for 1 client, 24.98 seconds for 2 concurrent clients, and 42.01 seconds for 4 concurrent clients. As with previous models, OPA showcased better performance for 1 and 2 concurrent clients compared to the GMA for 1 client, and across 1, 2, and 4 concurrent clients compared to the GMA with 1 client.

(4) For the Falcon model, we employed both GMA and OPA methods to distribute all 60 blocks across servers S1, S2, S3, and S4. S1 offers 98.3 GB memory with 8 cores, S2 has 88.2 GB memory with 2 cores, while S3 and S4 possess 88.2 GB memory with 4 and 8 cores, respectively. The GMA selected S1 (14 blocks) and S2 (10 blocks), whereas OPA chose S1 and S4. Notably, for Client 1, GMA exhibited a block execution time of 491.2 seconds, while OPA showed 211.37 seconds for Client 1 and 305.5 seconds for Client 2, signifying an improvement over Client 1 using GMA. This observed trend remained consistent across other models as well.

5.4 Model fits on 3 servers

Next, we experimented with OPA block placement algorithms across three servers, varying the client loads between 1, 2, and 4 concurrent clients. These trials encompassed three different models, including three versions of the Bloom model, the Falcon model, and the LLama 2 model. When a leftover capacity of a single or two servers was insufficient to accommodate all transformer blocks we designed this experiment using 3 servers. Notably, when employing the OPA algorithm with 2 concurrent clients, the block execution time increased with a higher number of concurrent clients, all the while maintaining a batch size of 16 and a fixed token length of 10.

In the context of the Falcon model presented in Table 3, we distributed blocks using both GMA and OPA methods across servers S1, S2, S3, S4, and S5. These servers possess varying memory and core configurations, with GMA selecting S1, S2, and S3, while OPA opted for S1, S2, and S5. Specifically, for Client 1 in GMA, the block execution time recorded was 260.12 seconds. However, utilizing OPA, the block execution time was 258.19 seconds for Client 1 and 263.89 seconds for Client 2, demonstrating comparable performance to that of Client 1 in GMA.

(1) bloom 560m: Utilizing both GMA and OPA techniques, we allocated all 24 blocks among servers S1, S2, S3, S4, and S5. S1 has 12.15 GB memory and 8 cores, S2 holds 11.65 GB memory with 8 cores, while S3, S4, and S5 share 11.15 GB memory but differ in core count—2, 4, and 8 cores respectively. GMA selected S1, S2, and S3, whereas OPA favored S1 (12 blocks), S2 (8 blocks), and S5 (4 blocks). Under the GMA, block execution times for 1 client were 2.75 seconds, 2 concurrent clients took 3.01 seconds, and 4 concurrent clients demanded 3.65 seconds. Meanwhile, employing OPA resulted in execution times of 2.68 seconds for 1 client, 2.88 seconds for 2 concurrent clients, and 3.11 seconds for 4 concurrent clients. OPA showcased superior performance across 1, 2, and 4 concurrent clients in contrast to the GMA.

(2) bloom 3b: All 30 blocks were allocated across servers S1, S2, S3, S4, and S5 using both GMA and OPA methodologies. S1 possesses 16.7 GB memory and 8 cores, S2 holds 15 GB memory with 8 cores, while S3, S4, and S5 share 13.3 GB memory, differing in core count—2, 4, and 8 cores respectively. GMA favored S1, S2, and S3, while OPA distributed 14 blocks to S1, 10 blocks to S2, and 6 blocks to S5. Block execution times under GMA for 1 client were 10.93 seconds, 12.09 seconds for 2 concurrent clients, and 14.11 seconds for 4 concurrent clients. With OPA, execution times were 10.28 seconds for 1 client, 11.23 seconds for 2 concurrent clients, and 13.91 seconds for 4 concurrent clients. OPA demonstrated superior performance across 1, 2, and 4 concurrent clients compared to the Greedy Approach.

(3) bloom 7b1: Distributing all 30 blocks across servers S1, S2, S3, S4, and S5 was achieved using GMA and OPA methods. S1 boasts 30 GB memory and 8 cores, S2 holds 26 GB memory with 8 cores, while S3, S4, and S5 share 22 GB memory but differ in core count—2, 4, and 8 cores respectively. GMA selected S1, S2, and S3, whereas OPA allocated 14 blocks to S1, 10 blocks to S2, and 6 blocks to S5. Using GMA, block execution times were 27.79 seconds for 1 client, 29.12 seconds for 2 concurrent clients, and 38.35 seconds for 4 concurrent clients. Meanwhile, OPA showed execution times of 22.12 seconds for 1 client, 23.59 seconds for 2 concurrent clients, and 36.12 seconds for 4 concurrent clients. OPA showcased superior performance across 1, 2, and 4 concurrent clients compared to the GMA.

(4) Falcon model: All 60 blocks were distributed across servers S1, S2, S3, S4, and S5 using GMA and OPA methodologies. S1 possesses 92.2 GB memory and 8 cores, S2 has 67.7 GB memory with 8 cores, while S3, S4, and S5 share 42 GB memory, differing in core count—2, 4, and 8 cores respectively. GMA selected S1, S2, and S3, whereas OPA opted for S1, S2, and S5. Under GMA, block execution times were 260.12 seconds for 1 client, 266.72 seconds for 2 concurrent clients, and 352.12 seconds for 4 concurrent clients. OPA demonstrated execution times of 258.19 seconds for 1 client, 263.89 seconds for 2 concurrent clients, and 345.26 seconds for 4 concurrent clients. OPA showcased superior performance across 1, 2, and 4 concurrent clients compared to the GMA Approach.

(5) LLama 2 model: Employing both GMA and OPA methods, we distributed all 80 blocks among servers S1, S2, S3, S4, and S5. S1 boasts 155 GB memory and 8 cores, S2 holds 113.5 GB memory with 8 cores, while S3, S4, and S5 share 83.5 GB memory, differing in core count—2, 4, and 8 cores respectively. S1, S2, and S3, while OPA favored S1 (35 blocks), S2 (25 blocks), and S5 (20 blocks). Block execution times under GMA for 1 client were 396.11 seconds, 408.12 seconds for 2 concurrent clients, and 798.12 seconds for 4 concurrent clients. Meanwhile, employing OPA resulted in execution times of 279.13 seconds for 1 client, 404.11 seconds for 2 concurrent clients, and 588.76 seconds for 4 concurrent clients.

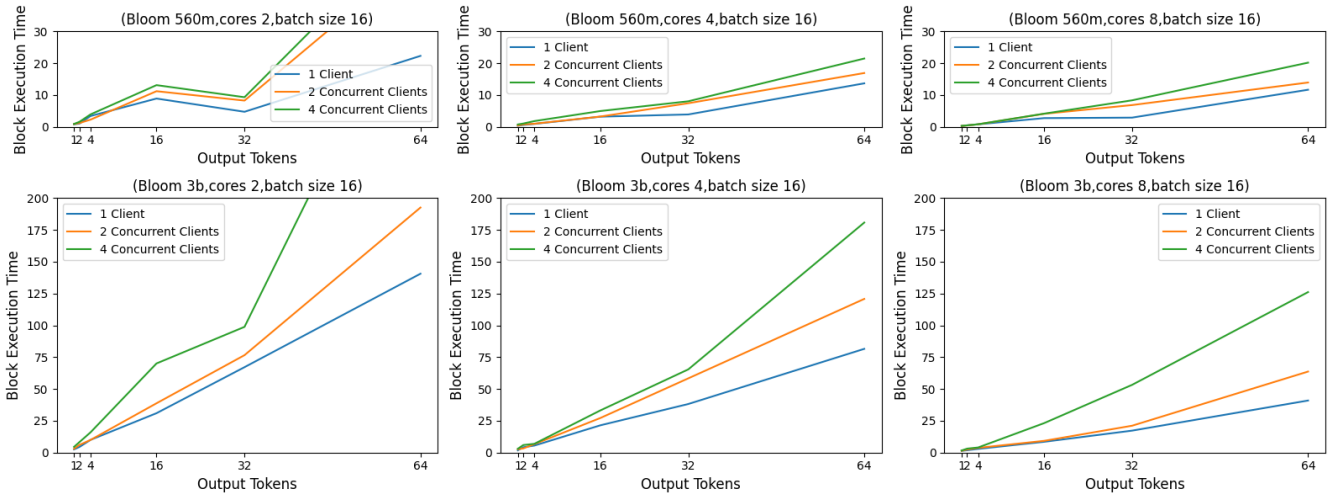


Figure 4: Block execution time for 1, 2, 4 concurrent clients

OPA demonstrated superior performance across 1, 2, and 4 concurrent clients compared to the GMA.

As observed from the results in Table 3, OPA outperforms GMA in almost all the cases, with a lower block execution time as compared to GMA. These results are consistent across different models. BLOOM, Falcon and LLaMA 2 with varying parameters and number of clients.

6 ABLATION STUDIES

In this section we discuss ablation studies to better understand and analyze the performance of OPA in LLaMPS. Figure 4 illustrates the behavior of OPA upon increasing the number of output tokens while keeping the batch size fixed. When the cores are limited to 2, the block execution time shows a steep increase as the number of output tokens increases. This is expected behavior as the compute increases, and so does the block execution time. This is particularly observed with an increase in concurrency (clients 2 and 4) and hence increase in a number of client requests. Upon adding more cores (4 and 8) we observe that the overall block execution time shows a decreasing trend as more compute becomes available. This trend remains consistent across the BLOOM 3b model, thus corroborating the efficacy of the OPA approach.

Previously, in Figure 3, we discussed the *knee-point* of the curve by varying the number of cores. These graphs illustrate that with increasing batch size, adding cores is beneficial till a certain point (i.e. 8 cores). Beyond 8 cores, we do not observe a significant improvement as some tasks performed during inference may not be parallelizable beyond 8 degrees. For example, in Bloom 560m with a batch size of 1, batch execution time is less than 20 seconds, for a batch size of 4, it is slightly over 20 seconds, and for a batch size of 16, it is less than 40 seconds. With the increase in cores, concurrency is more efficiently handled and this pattern holds for the Bloom 3b model as well.

Thus, OPA shows expected behavior across varying parameters (cores, batch sizes, number of output tokens) and the trend in the results is consistent across models with varying parameters. The efficacy of these results has also been validated on a heterogeneous cpu-based cluster setup in an enterprise.

7 RELATED WORK

In recent years, the deployment of large language models has seen widespread adoption owing to their versatile applications in real-world scenarios, predominantly in the field of natural language processing (NLP). As these models increase in size and complexity, their computational demands have necessitated distributed computing solutions, given their impracticality to fit on a single machine. Models such as OPT-175B[25], BLOOM-176B [23], and LLama-70b[22] demand substantial accelerator memory, exceeding 350 GB for inference. The computational requirements impose a barrier, necessitating multiple high-end GPUs for downstream tasks.

There have been efforts to distribute transformer blocks across multiple servers. Deepspeed [3] [17][18] is a distributed inference solution that supports large transformer-based language models. It enables parallel inference and achieves tensor parallelism by leveraging multiple GPUs. Deepspeed utilizes maximum available resources on multiple servers, enhancing overall performance and scalability.

Another approach by Deepspeed [3][17][18], namely Deepspeed Zero inference[19] involves "offloading" model parameters to more economical memory sources like RAM or SSD and subsequently executing them on the GPU layer by layer. While this strategy enables the deployment of LLMs on a single GPU, it introduces inherent trade-offs, such as increased latency due to the time-consuming offloading of layers. Additionally, the large size of layers and the substantial transfer overhead between the GPU and CPU further contribute to these challenges.

FastServe [24] is a system for distributed inference serving of LLMs. The system is built on NVIDIA FasterTransformer and exploits the autoregressive pattern of LLM inference to enable preemption at the granularity of each output token. FastServe utilizes

high-end GPUs and similar to DeepSpeed leverages maximum resources available to minimize the job (JCT) completion time.

Another avenue to enhance LLM accessibility involves leveraging public inference APIs for example deploying the LLM on Sagemaker [1] which is AWS [2] service later accessed model using API. However, the pricing structure of these APIs can render certain research projects prohibitively expensive. This economic constraint poses a limitation on the widespread use of LLMs for various research endeavors and potential applications.

Petals [7] works as a decentralized framework designed for fast inference of transformer-based models. It splits any given model into several blocks that are hosted on different servers. These servers can be spread out across continents, and anybody can connect their own GPU! In turn, users can connect to this network as a client and apply a chosen model to their data.

In our context, our paper introduces a novel perspective, focusing on the inference of large language models within enterprise setups. We seek to leverage the residual (leftover) capacity present on servers, harnessing untapped resources for efficient model inference. Our work explores the optimization of block assignment, which stays unexplored in related literature at this point. Through this exploration, we aim to push the boundaries of efficiency in large language model inference, ultimately maximizing the number of clients that can be accommodated, and thereby ensuring distributed inference of LLMs in a cost-effective manner for enterprises by utilizing leftover capacity.

8 CONCLUSIONS AND FUTURE WORK

Our paper introduces LLaMPS, a Large Language Model Placement System designed to address the challenge of efficiently deploying large language models (LLMs) within enterprise setups LLaMPS. Our approach focuses on the placement of transformer blocks, optimizing the utilization of enterprise resources by utilizing leftover capacity in worker nodes in an enterprise setup. The Optimal Placement Algorithm (OPA) maximizes the number of clients that can be served concurrently by optimally placing transformer blocks on residual resources. Through extensive experimentation with open-source large language models such as BLOOM (with 1b, 3b, and 7b parameters) Falcon and LLaMA, our results consistently demonstrate the efficacy of LLaMPS in facilitating optimal transformer block placement. By leveraging leftover resources, LLaMPS paves the way for enterprise-level deployment of large language models. LLaMPS can also maintain sustainability by optimally utilizing leftover capacity of machines within an enterprise; consequently saving the cost of buying extra machines/resources to host the entire LLM model. As a part of future work, we plan to investigate the feasibility of a system that dynamically identifies and utilizes leftover capacities across cloud instances. LLaMPS can be used in Retrieval Augmented Generation (RAG) applications for doing the inference or other downstream tasks using LLMs when latency is not critical.

REFERENCES

[1] [n. d.]. Amazon SageMaker. https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html.

[2] [n. d.]. AWS. https://aws.amazon.com/.

[3] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 46, 15 pages.

[4] Debarag Banerjee, Pooja Singh, Arjun Avadhanam, and Saksham Srivastava. 2023. Benchmarking LLM powered Chatbots: Methods and Metrics. *arXiv preprint arXiv:2308.04624* (2023).

[5] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2022. Distributed Inference and Fine-tuning of Large Language Models Over The Internet. (2022).

[6] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2022. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188* (2022).

[7] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2023. Petals: Collaborative Inference and Fine-tuning of Large Models. arXiv:2209.01188 [cs.LG]

[8] Jennifer D'Souza. [n. d.]. A Review of Transformer Models. ([n. d.]).

[9] Wenqi Fan, Zihuai Zhao, Jiatong Li, Yunqing Liu, Xiaowei Mei, Yiqi Wang, Jiliang Tang, and Qing Li. 2023. Recommender systems in the era of large language models (llms). *arXiv preprint arXiv:2307.02046* (2023).

[10] A Shaji George and AS Hovan George. 2023. A review of ChatGPT AI's impact on several business sectors. *Partners Universal International Innovation Journal* 1, 1 (2023), 9–23.

[11] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. (2023).

[12] AE Hefnawy and Ahmed Soliman Mohammed. 2014. Review of different methods for deriving weights in the Analytic Hierarchy Process. *International Journal of the Analytic Hierarchy Process* 6, 1 (2014), 92–123.

[13] Xu Huang, Jianxun Lian, Yuxuan Lei, Jing Yao, Defu Lian, and Xing Xie. 2023. Recommender AI Agent: Integrating Large Language Models for Interactive Recommendations. *arXiv preprint arXiv:2308.16505* (2023).

[14] Nian Li, Chen Gao, Yong Li, and Qingmin Liao. 2023. Large language model-empowered agents for simulating macroeconomic activities. *arXiv preprint arXiv:2310.10436* (2023).

[15] Jianghao Lin, Xinyi Dai, Yunjia Xi, Weiwen Liu, Bo Chen, Xiangyang Li, Chenxu Zhu, Huifeng Guo, Yong Yu, Ruiming Tang, et al. 2023. How Can Recommender Systems Benefit from Large Language Models: A Survey. *arXiv preprint arXiv:2306.05817* (2023).

[16] Keivalya Pandya and Mehfuza Holia. 2023. Automating Customer Service using LangChain: Building custom open-source GPT Chatbot for organizations. *arXiv preprint arXiv:2310.05421* (2023).

[17] Samyam Rajbhandari, Conglong Li, Zhewei Yao, and Minjia Zhang. 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International Conference on Machine Learning*. PMLR, 18332–18346.

[18] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[19] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, and Chen. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. (2023).

[20] Ah-Hwee Tan. 2004. FALCON: A fusion architecture for learning, cognition, and navigation. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*, Vol. 4. IEEE, 3297–3302.

[21] Hugo Touvron, Thibaut Lavril, and Gautier Izacard. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[22] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[23] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Castagné, François Yvon, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).

[24] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. *arXiv preprint arXiv:2305.05920* (2023).

[25] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]

Processing Natural Language on Embedded Devices: How Well Do Transformer Models Perform?

Souvika Sarkar*
szs0239@auburn.edu
Auburn University
Alabama, USA

Mohammad Fakhruddin Babar*
m.babar@wsu.edu
Washington State University
Pullman, USA

Md Mahadi Hassan
mzh0167@auburn.edu
Auburn University
Alabama, USA

Monowar Hasan
monowar.hasan@wsu.edu
Washington State University
Pullman, USA

Shubhra Kanti Karmaker Santu
sks0086@auburn.edu
Auburn University
Alabama, USA

ABSTRACT

Transformer-based language models such as BERT and its variants are primarily developed with compute-heavy servers in mind. Despite the great performance of BERT models across various NLP tasks, their large size and numerous parameters pose substantial obstacles to offline computation on embedded systems. Lighter replacements of such language models (e.g., DistilBERT and TinyBERT) often sacrifice accuracy, particularly for complex NLP tasks. Until now, it is still unclear *(a)* whether the state-of-the-art language models, viz., BERT and its variants are deployable on embedded systems with a limited processor, memory, and battery power and *(b)* if they do, what are the "right" set of configurations and parameters to choose for a given NLP task. This paper presents a *performance study of transformer language models* under different hardware configurations and accuracy requirements and derives empirical observations about these resource/accuracy trade-offs. In particular, we study how the most commonly used BERT-based language models (viz., BERT, RoBERTa, DistilBERT, and TinyBERT) perform on embedded systems. We tested them on ***four*** off-the-shelf embedded platforms (Raspberry Pi, Jetson, UP^2, and UDOO) with 2 GB and 4 GB memory (i.e., a total of ***eight*** hardware configurations) and ***four*** datasets (i.e., HuRIC, GoEmotion, CoNLL, WNUT17) running various NLP tasks. Our study finds that executing complex NLP tasks (such as "sentiment" classification) on embedded systems is *feasible* even without any GPUs (e.g., Raspberry Pi with 2 GB of RAM). Our findings can help designers understand the deployability and performance of transformer language models, especially those based on BERT architectures.

*Both authors contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645054

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**;
• **Computer systems organization** → **Embedded hardware**.

KEYWORDS

Transformers; Embedded Systems; NLP; Language Models

ACM Reference Format:
Souvika Sarkar, Mohammad Fakhruddin Babar, Md Mahadi Hassan, Monowar Hasan, and Shubhra Kanti Karmaker Santu. 2024. Processing Natural Language on Embedded Devices: How Well Do Transformer Models Perform?. In *Proceedings of the 15th ACM/SPEC Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645054

1 INTRODUCTION

The natural language processing (NLP) domain and the emergence of large language models rapidly transform how we interact with technology. With the proliferation of IoT-specific applications, the demand for voice-controlled services that can perform tasks by responding to spoken commands is growing. Use cases of NLP applications, especially those that need embedded and mobile systems, include home automation, healthcare, industrial control, and automotive infotainment. To design a dialogue-based interaction system for a target device, we need models that are feasible to *(a)* run on the hardware and *(b)* meet the desired level of accuracy. Although some services such as digital assistants (e.g., Alexa, Siri, Cortana) may leverage cloud resources for processing human voices, there exist applications (e.g., offline home/industrial robots, automotive infotainment, battlefield/military equipment) that may not have network connectivity, thus require to synthesize NLP tasks on the embedded device itself. The challenge is understanding the *feasibility* of running a large language model on *resource-limited* devices.

Transformer-based architectures [1], especially BERT-based models [2], have established themselves as popular state-of-the-art baselines for many NLP tasks, including *Intent Classification (IC)*, *Sentiment Classification (SC)*, and *Named Entity Recognition (NER)*. However, a well-known criticism of BERT-based architectures is that they are data-hungry and consume a lot of memory and energy; therefore, deploying them in embedded systems is challenging. In fact, due to their excessive size (431 MB) and parameters (110 M), deploying a pre-trained BERT model (called

$BERT_{Base}$) in resource-constrained embedded devices is often impractical, especially at the production level with certain minimum accuracy/performance requirements. Lighter versions of BERT (e.g., DistilBERT [3] and TinyBERT [4]) often result in accuracy losses. The degree of degradation in performance depends on the difficulty of the task, especially since those models often cannot perform well on complex NLP tasks, including emerging entity [5] or mixed emotion detection [6]. Therefore, designers must make an inevitable trade-off between an accurate model and one that can run smoothly in a resource-constrained environment. Unfortunately, developers often have little idea about this trade-off and have to spend a lot of time conducting trial-and-error experiments to find a suitable architecture that is feasible for the target (resource-constrained) hardware and meets a desired level of accuracy.

From a developer's perspective, it is still unclear what is the "right" BERT-based architecture to use for a given NLP task that can strike a suitable *trade-off between the resources available and the minimum accuracy* desired by the user. Due to the staggering size of the $BERT_{Base}$ model, we experiment with different "distilled" BERT models (e.g., DistilBERT and TinyBERT) for IC, SC, and NER tasks. However, existing ready-to-use distilled models perform poorly on some SC and NER datasets (Sec. 4). Hence, there is a need to *explore other models that can better optimize the efficiency/accuracy trade-offs.*

This research performs an *exploratory study of BERT-based models*[1] *under different resource constraints and accuracy budgets to derive empirical data about these resource/accuracy trade-offs.* We aim to answer the following questions: *(a)* how can we determine the suitable BERT architecture that runs on a target hardware and meets user-defined performance requirements (accuracy, inference time)? *(b)* what are the trade-offs between accuracy and model size as we perform optimizations (such as *pruning*) to run them on limited embedded device memory? and *(c)* what are the implications of performing pruning on accuracy and corresponding resource usage, including memory, inference time, and energy consumption? In answer to those questions, we observe the overhead of running various BERT architectures on four different hardware (viz., Raspberry Pi [9], Jetson Nano [10], UP^2 [11], and UDOO [12]). Our experiments suggest that some BERT models (specifically those that are "distilled") failed to achieve desired performance goals (e.g., F1 score) for various NLP tasks. Further, although pruning can reduce model size, it does not significantly help in energy efficiency.

Contributions. Our study fills the gap between simulation-based studies and real-world scenarios, as no prior work has deployed these models on embedded platforms. The findings of this work can help designers choose alternative BERT-based architectures under given resource constraints, thus saving development time and hassle. To ensure reproducibility, our implementation and related documentation is **publicly available** [13].

We made the following contributions in this paper.

- Our study systematically investigates the performance of BERT-based language models on four off-the-shelf embedded platforms (Raspberry Pi, Jetson, UP^2, and UDOO) with two different memory variants (2 GB and 4 GB RAMs). We analyzed the trade-offs between complexity and accuracy across multiple NLP tasks. (Sec. 3-Sec. 4).
- We explore the feasibility of deploying complex NLP tasks on embedded systems and analyze them under three *metrics*: *(a)* inference time, *(b)* memory usage, and *(c)* energy consumption. We developed a lookup table through empirical observations that will be useful for system designers to decide suitable model configurations for the target platform (Sec. 4).

Our key findings. The observations from executing the NLP tasks on our test platforms are as follows: *(a)* simpler NLP tasks such as IC can result in a relatively high (90%+) F1 score; *(b)* the time required to perform inference proportional to the size of the trained model and trimming them result in some accuracy loss (e.g., 60% of reduction in model size could reduce the accuracy by 50%); *(c)* the energy consumption on our test hardware remains relatively consistent, whether we prune the model or not; and *(d)* while GPUs play a role in decreasing inference time, the unavailability of GPUs can be compensated by faster CPUs.

We now start with the problem statement and present selected datasets (Sec. 2). Section 3 describes our experiment setup before we discuss our findings in Sec. 4.

2 PROBLEM STATEMENT & DATASETS

We aim to study *how language models can be optimally deployed* to accomplish dialog processing in embedded devices. The core technical challenge of any dialog system is to accurately understand and interpret user "utterances" and perform the right "action" accordingly. At a fundamental level, user utterance understanding relies on the following three basic NLP tasks[2]: *(a) Intent classification (IC)* — to understand the need of the user, *(b) Sentiment Classification (SC)* — to understand user emotions, and *(c) Named-entity Recognition (NER)* — to extract related entities such as persons or objects.

Figure 1 presents the workflow of the dialogue-based systems considered in this work. The user initiates the interaction by providing a spoken command to the voice-controlled device (marker ① in Fig. 1). The device employs existing automatic speech recognition techniques [14, 15] to convert the user's speech into texts as most language models take textual input (②). The system then runs an "intent classifier" (Sec. 2.1.1) and analyzes the extracted text to determine the user's *intention* (③). The classifier identifies the relevant user intentions for the given command (④). For instance, the intents for the given command *"Can you please go to my study room and turn off the lights?"* could be identified as *"Motion"* and *"Change operational state,"* as it instructed the device to move from its current position. Simultaneously, a "sentiment classifier"(Sec. 2.1.2) is employed to extract the user *sentiment* (⑤). In this case, the extracted sentiment is *"Neutral"*

[1] Note: there exist other large language models, such as GPT [7] from OpenAI and LaMDA [8] from Google. However, they are even more resource-hungry than BERT (thus less suitable for embedded deployment), and some are close-sourced. Hence, our initial study limits on BERT-based architectures.

[2] Section 2.1 formally presents these three NLP tasks.

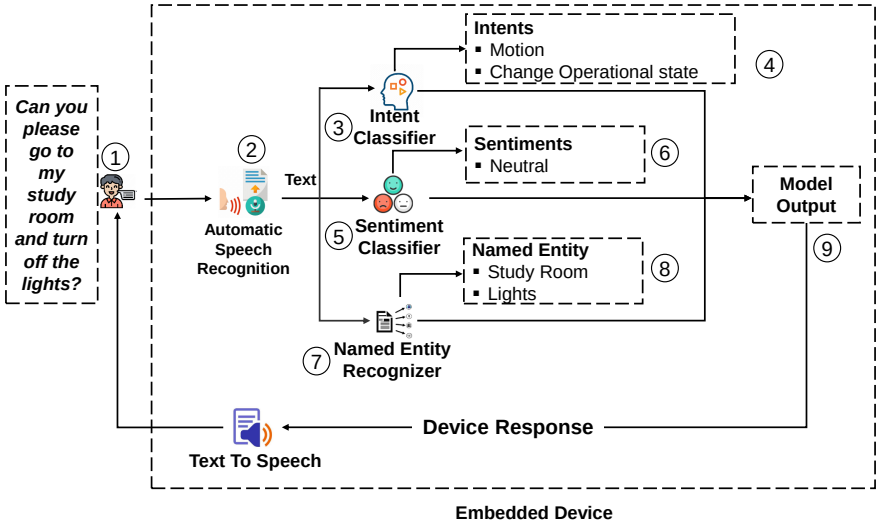


Figure 1: Utterance processing steps of a voice-controlled embedded device.

as the command does not express any emotion (⑥). Sentiment classification helps the voice-controlled device grasp the emotional context behind user utterances, enabling it to respond appropriately. In addition, the dialog system utilizes a "named entity recognizer" (Sec. 2.1.3) to identify specific "entities" (⑦-⑧). NER is crucial for accurately identifying user-specified entities like locations and objects, ensuring precise execution of commands in this scenario. For example, the entities, in this case, are *"study room"* (location) and *"lights"* (object). Once the user's intention and relevant entities are identified (⑨), the control application running on the embedded device carries out the specified task and sends a response back to the user. In this paper, we focus on understanding how the language models perform on embedded platforms for IC, SC, and NER tasks (e.g., steps ③-⑧).

2.1 NLP Tasks under Consideration

Recall from our earlier discussion that intent/sentiment classification and named-entity recognition are fundamental NLP tasks for any voice-controlled interactive system, chatbots, and virtual assistants. We now formally introduce IC, SC, and NER tasks.

2.1.1 Intent Classification. To produce an accurate response, reduce backtracking, and minimize user frustration, Intent Classification (IC) is needed to identify which subsequent action a robot needs to perform depending on the user's utterance. A formal definition of IC can be given as follows:

Definition 1. *Given a collection of user utterances* $U = \{u_1, u_2, ..., u_n\}$*, and a set of intent labels* $I_x = \{i_1, i_2, ..., i_m\}$*, classify each utterance* $u_j \in U$ *with one to more intents labels from* I_x*.*

Importantly, a user might have more than one intent while speaking to a robot and understanding the implicit or explicit intent expressed in a statement is essential to capturing the user's needs. For example, consider the following command: "*Can you please go to my study room and turn off the lights?*" The command wants the robot to turn off the lights in the study room; the relevant intent here is "*Change Operational State*". However, the statement also expressed another intent related to "*Motion*", as the command requires the robot to change location. Without identifying all the underlying intents, the system cannot perform the right next step. Hence, recognizing and understanding *all* types of intents stated in an utterance is crucial for accomplishing the eventual goal.

2.1.2 Sentiment Classification. Sentiment analysis is regarded as an important task for accurate user modeling in natural dialog-based interactions, where user utterances are usually classified into multiple emotion/sentiment labels. A formal definition can be given as follows:

Definition 2. *Given a collection of user utterances* $U = \{u_1, u_2, ..., u_n\}$*, and a set of sentiment labels* $S_x = \{s_1, s_2, ..., s_m\}$*, classify each expression* $u_j \in U$ *with one to more sentiment labels from* S_x*.*

For example, the following user utterance, *"OMG, yep!!! That is the final answer. Thank you so much!"* will be classified with sentiment labels "gratitude" and, "approval". Similarly, statements such as *"This caught me off guard for real. I'm actually off my bed laughing"* will be labeled as "surprise" and "amusement."

2.1.3 Named-entity Recognition. Named entity recognition (NER) — often referred to as entity chunking, extraction, or identification — is a sub-task of information extraction that seeks to locate and classify named entities mentioned in unstructured text. An entity can be expressed by a single word or a series of words that consistently refer to the same thing. Each detected entity is further classified into a predetermined category. The formal definition of the NER task can be given as follows:

Definition 3. *Given a collection of statements/texts* $S = \{s_1, s_2, ..., s_n\}$*, and a set of entity labels* $E_x = \{e_1, e_2, ..., e_m\}$*, all the words/tokens in the text will be classified with an entity label* $e_i \in E_x$*.*

NER can be framed as a sequence labelling task that is performed in two steps, first, detecting the entities from the text, and second, classifying them into different categories. A named entity recognizer model classifies each word/phrase representing an entity into one of the four types: *(a)* persons (PER), *(b)* objects (OBJ), *(c)* locations (LOC), and *(d)* miscellaneous names (MISC).

2.2 Datasets

Our study includes the following datasets: *(a)* HuRIC (for IC), *(b)* GoEmotion (for SC), and *(c)* CoNLL and WNUT17 (for NER), as we present below.

2.2.1 Intent Classification: HuRIC. For IC, we use Human Robot Interaction Corpus (HuRIC) [16], which is the state-of-the-art single-class classification dataset. The basic idea of HuRIC is to build a reusable corpus for human-robot interaction in natural language for a specific application domain, i.e., house service robots. HuRIC includes a wide range of user utterances given to a robot representing different situations in a house environment. The motivation behind selecting HuRIC for our intent classification task stems from our specific interest in utilizing a dataset that captures human-robot conversations. The HuRIC dataset allows us to train and evaluate intent classification models on realistic dialogues between humans and robots in real-world scenarios. Table 1 presents some statistics of HuRIC.

Statistic	Count
Number of examples	729
Number of intent labels	11
Size of training dataset	583
Size of test dataset	146

Table 1: Statistics of HuRIC dataset.

2.2.2 Sentiment Classification: GoEmotion. We use GoEmotion [6] dataset from Google AI for the SC task. GoEmotion is a human-annotated dataset of 58,000 Reddit comments extracted from popular English-language subreddits and labeled with 27 emotion categories. As the largest fully annotated English language fine-grained emotion dataset to date, the GoEmotion taxonomy was designed with both psychology and data applicability in mind Table 2 presents some statistics of GoEmotion. We chose the GoEmotion dataset for SC task to ensure rigorous testing of our models. With its comprehensive emotion coverage and nuanced labeling, GoEmotion serves as a challenging yet realistic benchmark for evaluating sentiment detection performance.

2.2.3 Named-entity Recognition: CoNLL & WNUT17. For NER we consider two datasets, viz., CoNLL [17] and WNUT17 [5].

CoNLL. CoNLL-2003 [17] was released as a part of CoNLL-2003 shared task: language-independent named entity recognition. The English corpus from this shared task consists of Reuters news stories between August 1996 and August 1997, each annotated with the entities associated with them. The data set consists of a training file, a development file, and a test file. The details of CoNLL-2003 are presented in Table 3.

Statistic	Count
Number of labels	27 + Neutral
Maximum sequence length in overall datasets	30
Size of training dataset	43,410
Size of test dataset	5,427
Size of validation dataset	5,426

Table 2: Statistics of GoEmotion dataset.

Statistic	Articles	Sentences	Tokens
Training set	946	14,987	203,621
Development set	216	3,466	51,362
Test set	231	3,684	46,435

Table 3: Statistics of CoNLL dataset.

WNUT17. While the CoNLL corpus is based on news stories, we wanted to select a dataset that contains user utterances such as those available on HuRIC. Unfortunately, we could not find such a NER dataset but discovered a very similar corpus (WNUT2017 [5]) that contains user-generated text. The WNUT2017 dataset's shared task focuses on identifying unusual, previously-unseen entities in the context of emerging discussions. Identifying entities in noisy text is really challenging, even for human annotators, due to novel entities and surface forms. In this dataset, user comments were mined from different social media platforms because they are large, and samples can be mined along different dimensions, such as texts from/about geo-specific areas, about home aid, and particular topics and events. Table 4 summarizes the dataset properties.

Statistic	Count
Number of examples	5690
Number of labels	6
Size of training dataset	3394
Size of test dataset	1287
Size of validation dataset	1009

Table 4: Statistics of WNUT17 dataset.

3 EXPERIMENTAL SETUP

We now summarize BERT architectures and configurations used in our experiments (Sec. 3.1 and Sec. 3.2). We selected four off-the-shelf platforms from different chip vendors (Intel, AMD, ARM, NVIDIA) to understand the feasibility of using NLP tasks on a variety of architectures (Sec. 3.3). To measure the performance of each task (IC, SC, NER), we use three popular metrics (i.e., Precision, Recall, and F_1 score). Section 3.4 lists the design questions explored in our investigation. The blueprints of our implementation, including related code/documentation, is **publicly available** for community use [13].

	BERT	RoBERTa	DistilBERT	TinyBERT
Number of Layers	12	12	6	4
Attention Heads	12	12	12	12
Hidden Layer Size	768	768	768	312
Feed-Forward Layer Size	3072	3072	3072	1200
Vocabulary Size	30522	30522	30522	30522

Table 5: Attributes of the BERT variants used in our study.

3.1 Off-the-Shelf BERT Variants

We use a pre-trained base variant of BERT [18], RoBERTa [19], DistilBERT [3], TinyBERT [4] model from Huggingface[3] and finetune the models on respective datasets (i.e., HuRIC, GoEmotion, CoNLL, and WNUT17). Table 5 listed the parameters of the BERT variants and Table 6 presents the hyper-parameter used in our experiments.

Hyperparameter	NER	IC/SC
Number of epochs	3	3
Batch size	64	64
Learning rate	$[e^{-6}, e^{-4}]$	$[e^{-6}, e^{-4}]$
Weight decay	$[0.01, 0.3]$	$[0.01, 0.3]$
Optimizer	Adam	Adam
Adam epsilon	$1e^{-8}$	$1e^{-8}$
Max sequence length	64	128

Table 6: Hyperparameter values for finetuning BERT on IC/SC and NER tasks.

3.2 Pruning and Custom Configurations

We also experiment with custom, smaller BERT configurations. Due to the resource constraints (e.g., memory and energy limits) of embedded devices, it is necessary to explore different variants of BERT-based models that can be optimized to run on the device. We can reduce the model size on two fronts: *(a)* by reducing the layer size and *(b)* by pruning various attributes.

In our study, we experiment with *different layer combinations* of BERT models and test their performance on *different hardware configurations*, which are presented in Table 12, and 13. With two layers of $BERT_{Base}$ (instead of 12), the model size reduces significantly, but so does the accuracy (in terms of F_1 score). Still, these models give better accuracy than the distilled models on complex NLP tasks. Also, where $BERT_{Base}$ model with 12 layers cannot run on a resource-constrained device, using a lesser number of layers enable a model to execute on tiny devices with good accuracy compared to the distilled methods (DistilBERT and TinyBERT).

For further shrinking of the model, pruning can be applied to weights, neurons, layers, channels, and attention heads, depending on the heuristic used. In this paper, we focus on pruning *attention heads*, which plays an important role in the model's performance and contributes a large number of parameters. Although multi-headed attention is a driving force behind many recent state-of-the-art models, Michel et al. [20] finds that even if models have been trained using multiple heads, in practice, a large percentage of attention heads can be removed without significantly impacting performance. In fact, in some layers, *attention heads* can even be reduced to a single head.

Based on this fact, we experiment with reducing the size of $BERT_{Base}$ by dynamically pruning the *attention heads*. Each attention head provides a distribution of attention for the given input sequence. For each attention head, we calculate the head importance by calculating the *entropy* of each head. After that, we mask all the attention heads with the lowest entropy. This process is repeated for each layer of the encoder. After masking the heads, we calculate the overall F_1 score of the masked model and determine the drop in F_1 score compared to the original unpruned model. If the drop is less than a predefined threshold, we prune the masked attention heads. We repeat the process until the drop in F_1 score reaches the predefined threshold. This pruning procedure reduces the model size significantly while maintaining the desired model performance.

3.3 Evaluation Platforms

We evaluate the BERT models on heterogeneous setup, viz., x86 (Intel and AMD) and ARM platforms, including devices with or without GPU. In particular, we used the following four different embedded platforms: *(a)* Raspberry Pi 4 Model B [9], *(b)* Jetson Nano [10], *(c)* UP^2 [11], and *(d)* UDOO Bolt [12]. Table 7 lists the hardware configurations used in our setup. Among them, only Jetson board equipped with GPU (128 NVIDIA CUDA cores). We used 2 GB and 4 GB memory configurations for each of the four boards. The SoC (System on Chip) of ARM-based boards (Raspberry Pi and Jetson) configured with soldered memory — hence, we used two different boards for each with 2 GB and 4 GB RAM configurations. The x86-based boards (i.e., UP^2 and UDOO) are modular, so we used the same board but two different DDR4 RAMs (2 GB and 4 GB). Hence, our evaluation setup consists of *eight different hardware configurations* running on four ARM and two x86 boards. For energy measurements during the inference steps, we used UM25C energy meter [21]. We performed all experiments on Linux kernel 5.15.0. The NLP models were developed using PyTorch library (version 1.13).

3.4 Design Challenges & Research Questions

We conducted extensive experiments to investigate the following research questions (RQs).

- **RQ 1.** Given specific user-defined constraints, such as system resources (processor, memory) and performance budgets (accuracy, inference time), what is the optimal (if any) BERT-based architecture satisfying those constraints?
- **RQ 2.** What is the accuracy vs. model-size trade-off as we prune the models?
- **RQ 3.** What are the trade-offs of accuracy and corresponding resource usage (e.g., memory, inference-time, energy consumptions) as we perform pruning?
- **RQ 4.** Does GPU aid in inference time?

[3] https://huggingface.co/.

Embedded Platform		Architecture	CPU	GPU	Memory
	Raspberry Pi	ARM	Quad-core Cortex-A72	✖	2 GB and 4 GB
	Jetson Nano	ARM	Quad-core Cortex-A57	128-core NVIDIA Maxwell	2 GB and 4 GB
	UP^2	x86	Dual-core Intel Celeron N6210	✖	2 GB and 4 GB
	UDOO Bolt	x86	Quad-core AMD Ryzen V1605B	✖	2 GB and 4 GB

Table 7: Embedded platforms used in our evaluations: *(a)* Raspberry Pi [9], *(b)* Jetson Nano [10], *(c)* UP^2 [11], and *(d)* UDOO Bolt [12]. We used 2 GB and 4 GB memory configurations for each board.

- **RQ 5.** What are the energy consumption differences among various architectures (x86 and ARM)? Does the presence of GPUs impact energy usage?

4 RESULTS

In this section, we report our results for the three basic NLP tasks, i.e., IC, SC, and NER for both existing (e.g., BERT, RoBERTa, DistilBERT, and TinyBERT) and custom BERT architectures.

4.1 Experience with Existing BERT Variants

Intent Classification (IC). Recall from our earlier discussion (Sec. 2.2.1) that we used the HuRIC dataset for IC. Table 8 presents our findings after running IC tasks using the HuRIC dataset on different hardware. We observed that all the models performed similarly on this dataset, achieving more than 90% F_1 Score.

Multi-label Sentiment Classification (SC). We next analyze the performance of multi-label SC tasks on Raspberry Pi, Jetson, UP^2, and UDOO board. As mentioned in Sec. 2.2.2, we used GoEmotion [6] dataset for this task. GoEmotion includes direct user-to-user conversation text and labels them with many user emotions. Table 9 summarizes the performance of all the models for this task. Interestingly, for this task, DistilBERT and TinyBERT failed drastically, as they achieved a very low F_1 Score. The failure of distilled models can be attributed to the difficulty of the task. Multi-label SC requires each utterance to be classified with more than one sentiment. Therefore, this dataset is not a straightforward positive-negative sentiment detection.

Named-entity Recognition (NER). As we mention in Sec. 2.2.3, we use two different datasets to test the NER task. Table 10 summarizes the performance over both the NER datasets and shows that for the CoNLL dataset. In this setup, all models performed comparatively the same. However, the performance of distilled models dropped sharply for the WNUT17 dataset (which focuses on identifying unusual, previously-unseen entities). This drop tends to be due to the difficulty of analyzing this task, as NER evaluates the ability to detect and classify novel, emerging, singleton-named entities in noisy inputs.

In summary, our findings are as follows.

- All models achieved decent F1 scores (>90%) for IC task.
- DistilBERT and TinyBERT struggled with the multi-label SC task as none achieved an F1 score of more than 15%.
- All BERT models excelled in the NER task on the CoNLL dataset and accurately recognized named entities (resulting in >90% F1 scores).
- Distilled models showed a performance decline for the NER task on the WNUT17 dataset with the F1 score dropping to less than 5%, indicating difficulty with dataset intricacies.

4.2 Exploration with Custom Architectures

Based on our experiment results (Tables 8–10), we further explore different alternative BERT-based architectures by reducing the layers and pruning the attention heads from the original $BERT_{Base}$ model. For this exploration, we primarily focus on the challenging tasks, i.e., multi-label SC and NER where off-the-shelf models (e.g., DistilBERT and TinyBERT) failed to perform.

Table 11–Table 13 present our exploration findings.[4] We discuss our observations in Sec. 4.2.3 and provide answers to the research questions posed in Sec. 3.4. Before we proceed with the discussion, we present a brief overview of the attributes and metrics used in our evaluation.

4.2.1 Model Attributes. In the evaluation, we vary the following model attributes.

- F_1 *Threshold* (θ): The θ-cells represents what percentage of the F_1 score (with respect to $BERT_{Base}$) is retained by the models. In our experiment, we varied θ between 50% to 90% and reported

[4] **Note:** We omit the results for IC on custom BERT architectures as existing models suffice to perform this task.

Intent Classification Task (Dataset: HuRIC)											
BERT			**RoBERTa**			**DistilBERT**			**TinyBERT**		
Precision	**Recall**	F_1	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1
0.943	0.985	0.961	0.975	0.952	0.962	0.951	0.903	0.927	0.912	0.897	0.902

Table 8: Performance of BERT, RoBERTa, DistilBERT, and TinyBERT for the IC task.

Multi-label Sentiment Detection Task (Dataset: GoEmotion)											
BERT			**RoBERTa**			**DistilBERT**			**TinyBERT**		
Precision	**Recall**	F_1	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1
0.77	0.37	0.490	0.731	0.452	0.567	0.174	0.121	0.134	0.060	0.030	0.031

Table 9: Performance of BERT, RoBERTa, DistilBERT, and TinyBERT for the SC task.

Named-entity Recognition Task											
Dataset: CoNLL											
BERT			**RoBERTa**			**DistilBERT**			**TinyBERT**		
Precision	**Recall**	F_1	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1
0.891	0.963	0.926	0.882	0.955	0.917	0.906	0.967	0.935	0.872	0.958	0.911
Dataset: WNUT17											
BERT			**RoBERTa**			**DistilBERT**			**TinyBERT**		
Precision	**Recall**	F_1	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1
0.671	0.295	0.410	0.537	0.315	0.397	0.316	0.014	0.028	0.0	0.0	0.0

Table 10: Performance of BERT, RoBERTa, DistilBERT, and TinyBERT for the NER task.

Task	**Metrics**	**F_1 Score Threshold (θ)**																			
		θ_{50}				θ_{60}				θ_{70}				θ_{80}				θ_{90}			
IC	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	MS	144.3	N/A	N/A	N/A	148.3	N/A	N/A	N/A	N/A	195.6	N/A	282.2	154.2	198.7	246	N/A	N/A	211.3	268	303.5
	Params	35.1				37.1					48.9		70.5	38.6	49.7	61.5			52.8	67	75.9
	Pruning	68%				66%					56%		36%	65%	55%	44%			52%	39%	31%
NER	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	MS	136.4	N/A	N/A	N/A	147.4	N/A	N/A	N/A	N/A	185.2	233.3	N/A	N/A	201	249.8	268	N/A	N/A	260.8	289.2
	Params	34.1				36.9					46.3	58.3			46.3	62.4	67			65.2	72.3
	Pruning	67%				65%					57%	45%			45%	57%	38%			39%	32%

Table 11: Performance of SC and NER tasks for the GoEmotion and WNUT17 datasets on various configurations. In metrics column, MS= Model Size (MB), Params= Parameters (Million).

the model details in respective columns. For example, θ set to 80% implies the θ_{80} column.

- *Platform*: Indicates the memory capacity of the different hardware we used in our exploration.
- *Layer*: Represents the number of layers retained.
- *Model Size*: The size of the saved model after training. We stored the saved model on the disk which is then loaded on the memory for inference.
- *Parameters*: This metric indicates the total number of parameters in the saved model. We obtained the model parameters using the `model.parameters()` method in PyTorch [22].
- *Pruning*: Pruning percentage represents the reduction in size from the $BERT_{Base}$ model. For example, a pruning percentage of 70% implies that the pruned model is 70% smaller than $BERT_{Base}$.

4.2.2 Performance Metrics. We consider the three metrics to benchmark the NLP models: *(a)* inference time, *(b)* memory usage, and *(c)* energy consumption, as we present below.

- *Memory Consumption*: Maximum memory usage (in megabytes) of the corresponding NLP task running on different boards during the inference time. We used Python `memory_profiler` for each input to get the memory usage.
- *Inference Time*: Depending on the specific task, the 95th percentile time required for the model to infer the appropriate *Intent, Sentiment or Entity* from a given command.
- *Energy Consumption*: The average energy consumed (in watts) by the different hardware during the inference of a given command. We measured both the *rest time* (i.e., when the system is idle) and *inference time* energy consumption.

Plat.	Metrics	F_1 Score Threshold (θ)																			
		θ_{50}				θ_{60}				θ_{70}				θ_{80}				θ_{90}			
Pi 2 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	4.24	N/A	N/A	N/A	4.28	N/A	N/A	N/A	N/A	4.32	N/A	4.37	4.35	4.3	4.48	N/A	N/A	3.91	4.45	4.34
	MC	394.3				391.5					440.4		563	401.9	446.2	497.3			462.9	521.5	559.4
	IT	0.52				0.51					1.1		2.17	0.54	1.01	1.69			1.3	1.81	2.35
Pi 4 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	N/A	4	6	8
	EC	4.91	N/A	N/A	N/A	5.05	N/A	N/A	N/A	N/A	4.67	N/A	5.09	4.98	4.72	4.8	N/A		4.77	4.78	4.95
	MC	397.5				401					446.3		569.2	404.2	453.2	507.7			464.6	528.8	563.4
	IT	0.44				0.44					0.82		1.5	0.5	0.81	1.21			1.00	1.46	1.55
Jetson 2 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	5.87	N/A	N/A	N/A	5.96	N/A	N/A	N/A	N/A	6.01	N/A	6.28	5.87	6.14	6.26	N/A	N/A	6.13	6.22	6.24
	MC	299.2				320.1					345.4		478.5	329.7	352.6	415			367.5	429.8	463.9
	IT	0.29				0.26					0.49		0.88	0.30	0.51	0.68			0.56	.797	1.02
Jetson 4 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	6.27	N/A	N/A	N/A	6.25	N/A	N/A	N/A	N/A	6.21	N/A	6.43	6.27	6.31	6.36	N/A	N/A	6.39	6.4	6.47
	MC	355.5				359.7					410.1		532.9	366	408.8	459.4			423	488.4	524
	IT	0.27				0.27					0.45		0.86	0.30	0.46	0.66			0.50	0.76	0.87
UP^2 2 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	10.897	N/A	N/A	N/A	10.9	N/A	N/A	N/A	N/A	10.94	N/A	10.79	10.96	11.05	10.78	N/A	N/A	10.81	10.73	10.76
	MC	670.9				708.6					606.8		741.7	696.8	693	742.6			741.1	707.4	782.1
	IT	0.12				0.12					0.21		0.39	0.13	0.21	0.30			0.23	0.31	0.42
UP^2 4 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	10.17	N/A	N/A	N/A	11.05	N/A	N/A	N/A	N/A	10.04	N/A	11.51	10.99	11.20	11.38	N/A	N/A	11.01	11.38	11.16
	MC	738.3				653.2					715.7		870	736.8	753.4	792.9			736.2	798.3	875.3
	IT	0.12				0.12					0.21		0.40	0.13	0.22	0.30			0.24	0.30	0.41
UDOO 2 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	23.08	N/A	N/A	N/A	23.12	N/A	N/A	N/A	N/A	23.43	N/A	23.64	23.28	23.38	23.62	N/A	N/A	23.57	23.56	23.46
	MC	379.3				424.6					361.5		422	377.9	348.1	461			360.9	468.2	453.1
	IT	0.06				0.06					0.10		0.12	0.06	0.09	0.13			0.10	0.14	0.17
UDOO 4 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	23.076	N/A	N/A	N/A	23.12	N/A	N/A	N/A	N/A	22.78	N/A	23.18	22.98	23.05	22.99	N/A	N/A	23.13	22.96	23.15
	MC	450.1				445.4					487.3		596.8	443.8	505.9	547.8			505.7	544.7	601.9
	IT	0.07				0.07					0.11		0.16	0.08	0.11	0.15			0.12	0.16	0.16

Table 12: Performance of SC task for the GoEmotion dataset on various configurations. In metrics column, EC=Energy Consumption (W), MC= Memory (MB), and IT= Inference Time (s). The gray cells highlight the best-case scenario for each F_1 threshold (θ).

4.2.3 Observations. We now discuss our major observations and address the research questions introduced in Sec. 3.4.

Selecting "suitable" model subject to given constraints [RQ 1]. We can address this specific research question by inspecting Table 12 and 13. Note that Table 12 and 13 provide information on the model size, performance, parameters, and pruning for the SC and NER tasks, respectively. Let us assume a system designer is looking for suitable NER models for a 2 GB embedded platform that maintains approximately 70% of BERT's accuracy (θ_{70}). In this case, we can *(a)* scan through the NER performance metrics (i.e., Table 13), and *(b)* observe from Pi 2 GB *Platform* row and θ_{70} column that a six-layered and pruned (45% reduced) BERT model can run on a Pi 2 GB platform and attain 70% of $BERT_{Base}$'s original F_1 score. Hence, our exploration (and similar experiments along this line) can *aid the designers to select appropriate models with desired performance guarantees.*

Accuracy and model-size trade-offs for pruned architectures [RQ 2]. Table 12 and Table 13 further provide insights on the pruning vs. F_1 score trade-off. For example, in Table 12, the Pi 2 GB *Platform* row shows a set of models that can run on that system. The same row and θ_{80} (80% F_1 *score threshold*) column show that even pruning 55% of a $BERT_{Base}$ model with four layers can retain 80% of original F_1 score of $BERT_{Base}$, while the model size can be reduced to 198.7 MB from 441.55 MB. Tables 12 and 13 indicate that although *pruning has only a minor impact on memory consumption, it does not have a significant effect on energy consumption.* Furthermore, our analysis of Table 12 suggests that *inference time is directly proportional to the size of the model, implying that decreasing the model size leads to a decrease in inference time.*

Accuracy vs. system resource trade-offs for pruned architectures [RQ 3]. If a user has precise requirements for inference time and memory consumption for a given hardware, one can scan Table 12 and 13 to pick the optimum model that meets those requirements. For instance, if we want to find NER models that can make inferences in less than 0.56 seconds on Raspberry Pi that has 4 GB of memory, the corresponding *Platform* row Pi (4 GB) in Table 13, shows us the model parameters that can satisfy this requirement(e.g., two-layered, four-layered). Since both of them are feasible for the chosen platform, designers can choose any of them based on the required application performance. As an example, if we pick a two-layered BERT model, the accuracy is 60%, and the memory consumption is 698.3 MB. In contrast, if we select the four-layered BERT model, it can achieve 80% accuracy, with a cost of higher memory consumption of 699.1 MB. Hence, at the expense of slightly higher memory consumption, it is possible to get 20%

Plat.	Metrics	F_1 Score Threshold (θ)																			
		θ_{50}				θ_{60}				θ_{70}				θ_{80}				θ_{90}			
Pi 2 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	4.28	N/A	N/A	N/A	4.36	N/A	N/A	N/A	N/A	4.3	4.51	N/A	N/A	3.95	4.35	4.51	N/A	N/A	4.38	4.53
	MC	676.3				709					721.3	678.4			736.8	700.3	678.4			704.7	692.2
	IT	0.55				0.55					1.08	0.61			1.19	0.61	0.61			0.56	0.65
Pi 4 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	4.48	N/A	N/A	N/A	4.53	N/A	N/A	N/A	N/A	4.62	4.63	N/A	N/A	4.59	4.85	4.82	N/A	N/A	4.9	4.88
	MC	675				698.3					683.6	698.9			699.1	706	686			706.6	709
	IT	0.49				0.516					0.57	0.57			0.55	0.57	0.63			0.60	0.65
Jetson 2 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	5.74	N/A	N/A	N/A	5.8	N/A	N/A	N/A	N/A	5.96	5.7	N/A	N/A	6.05	5.77	5.7	N/A	N/A	5.8	5.71
	MC	314.6				348.2					388.7	357.7			392.4	359	362.7			362.3	368.9
	IT	0.29				0.29					0.45	0.29			0.50	0.31	0.29			0.33	0.29
Jetson 4 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	5.74	N/A	N/A	N/A	5.76	N/A	N/A	N/A	N/A	6.23	5.95	N/A	N/A	6.04	6.12	6.04	N/A	N/A	6.05	6.08
	MC	368.3				365.8					418.5	368.7			424.4	365.5	361.7			363.5	366.6
	IT	0.29				0.27					0.497	0.29			0.45	0.33	0.29			0.32	0.30
UP^2 2 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	11.108	N/A	N/A	N/A	11.2	N/A	N/A	N/A	N/A	11.18	11.44	N/A	N/A	11.21	10.77	10.72	N/A	N/A	10.63	10.7
	MC	512.5				597.6					652.9	574			650.8	601.8	603.5			591.3	591.1
	IT	0.13				0.12					0.19	0.18			0.20	0.12	0.12			0.12	0.13
UP^2 4 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	10.99	N/A	N/A	N/A	11.21	N/A	N/A	N/A	N/A	11.36	11.33	N/A	N/A	11.32	11.40	11.35	N/A	N/A	11.16	11.3
	MC	599.8				601.8					651.4	658.1			653.7	652.7	657.7			657.7	652.9
	IT	0.12				0.11					0.19	0.20			0.19	0.19	0.22			0.22	0.21
UDOO 2 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	24.02	N/A	N/A	N/A	24.08	N/A	N/A	N/A	N/A	24.09	24.07	N/A	N/A	24.14	24.06	24.02	N/A	N/A	24.11	24.07
	MC	437.8				438.6					491.2	490.8			490.9	490.5	503.4			490.3	492.4
	IT	0.07				0.05					0.09	0.09			0.09	0.09	0.09			0.09	0.08
UDOO 4 GB	Layer	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
	EC	22.79				22.86					23.07	23.57			23.09	23.49	23.39			23.58	23.50
	MC	445.6				443					500.1	496.4			497.5	499.5	510.1			495.2	499.8
	IT	0.06				0.05					0.08	0.09			0.09	0.08	0.09			0.08	0.08

Table 13: Performance of NER task for the WNUT17 dataset on various configurations. In metrics column, MS= Model Size (MB), Params= Parameters (Million), EC=Energy Consumption (Watt), MC= Memory (MB), and IT= Inference Time (s). The gray cells highlight the best-case scenario for each F_1 threshold (θ).

more accuracy. Such a lookup-based approach allows the designers to perform a desired *cost-benefit analysis*.

The case for GPUs [RQ 4]. Intuitively, GPUs aid in any learning-enabled tasks. We used one GPU-enabled hardware (Jetson Nano) in our design space exploration. As Table 12 and Table 13 illustrate Jetson reduces inference time compared to the Raspberry Pi board (no GPU). However, GPUs alone cannot provide faster inference. For instance, x86 boards (UP^2 and UDOO) do not have GPUs but are equipped with a faster processor, and hence result in better inference times (i.e., took less time to process the queries). Systems with better hardware (CPU/GPU/memory) can output inference decisions faster, which may increase power consumption, as we discuss next.

Energy consumption on various architectures and model configurations [RQ 5]. Since many embedded platforms used for NLP tasks (e.g., voice-controlled robots, voice assistants, IoT devices) are battery-operated, energy consumption for inferring user commands is a crucial parameter. Hence, we also analyze the energy usage of the NLP tasks. For any selected BERT model, one can find the system energy consumption from Table 12 and 13, for two different tasks, respectively. As the table shows, (for a given hardware) during the inference of a given command, *energy consumption does not vary significantly for various models.*

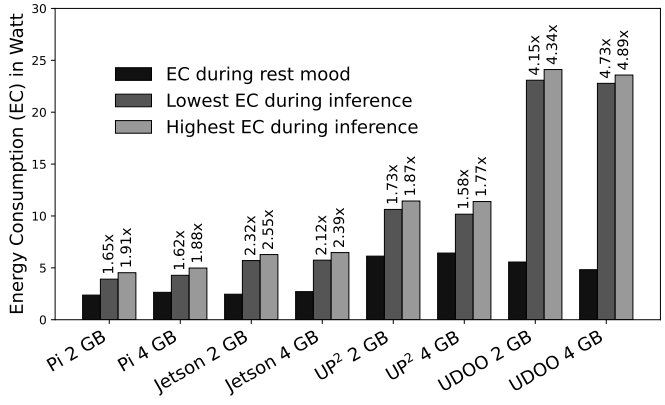


Figure 2: Energy consumption during rest mode and inference period. ARM devices use less energy compared to x86 systems.

To understand the energy usage of the various NLP tasks on our test platforms, we also measured the energy consumption of each board during rest mode and inference period (see Fig. 2). We obtained the rest mode energy usage by idling the device

for 10 minutes and took the average value. For inference energy consumption, we tested with 40 SC and 40 NER queries and repeated each of them 100 times (i.e., a total of $2 \times 40 \times 100 = 8000$ samples. We report the maximum (lighter gray) bar and minimum (darker gray) energy consumption values of the 8000 trials.

As Fig. 2 shows, the inference energy consumption increases by a factor of 1.73 to 4.89 times compared to the rest mode energy usage. Besides, ARM architectures (Raspberry Pi and Jetson) consume less energy than the x86 architectures (UP^2 and UDOO). Another interesting observation is that even though Jetson boards use GPU, they are more power efficient for performing NLP tasks compared to some CPU-only x86 systems. Our experiments show that the AMD Ryzen platform (UDOO) performs poorly in terms of energy usage. However, as Table 12 and Table 13 indicate, UDOO boards output faster inference time (since they have relatively faster CPU than the others). Hence, there exists a *trade-off* between inference time and energy usage.

4.2.4 Summary of Findings. Our key findings for custom BERT architectures are listed below.

Model Size & Pruning.

- Pruning helps in reduction in size (upto 67%) while maintaining at least 50% of $BERT_{Base}$'s F1 score.
- The time required for inference is directly related to the size of the model, i.e., a smaller model size results in a reduction in inference time.
- Pruning of attention heads does not reduce memory usage.
- Pruning attention heads does not improve energy consumption significantly.

System Artifacts.

- Faster x86 (Intel and AMD) platforms outperform ARM SoCs (e.g., Pi and Jetson boards) wrt. inference time, but their energy consumption is significantly higher (e.g., at least 2.60 times) than ARM counterparts.
- GPUs aid in performance (e.g., inference times are approx. 2-times faster in Jetson than Raspberry Pi) but GPUs alone in Jetson boards cannot outperform a relatively faster CPU (i.e., those used in UP^2 and UDOO boards).
- Powerful processor can decrease inference time (as expected) but comes with a cost (increased power consumption: 2.60-5.90 times higher).

5 DISCUSSION

We explore different custom architectures of BERT-based language models and test their deployment feasibility in low-power embedded devices. We conducted extensive performance evaluations on four embedded platforms from various vendors with varying computing capabilities to cover a wide range of application scenarios. We show that it is *not always feasible to shrink* the size of "finetuned" $BERT_{Base}$ model that can satisfy specific user-defined accuracy/performance budgets. We also report which models are deployable to resource-constrained devices with given user requirements. We believe our empirical findings will help the developers quickly narrow down the plausible BERT-based architecture for target applications, thus saving development time and effort. While we tested the NLP models on four embedded platforms (a total of 8 hardware configurations) in a Linux environment, they can be ported to other systems, such as smartphones/tablets running different OSes (such as Android). Thus our empirical study is applicable in broader human-centric application domains, including chatbots [23–25], virtual assistants [26, 27], and language translation [28, 29].

Our study is limited to BERT-based models for four existing datasets (i.e., may not generalize to other language models and datasets). However, our evaluation framework is modular and can be retrofitted to other architectures/datasets without loss of generality. While shrinking models have made it possible to deploy them on resource-constrained embedded devices, their performance on new datasets or tasks is often limited. One potential solution to mitigate this issue is to utilize continual learning techniques [30, 31], as they allow models to continuously learn and evolve based on increasing data input while retaining previously acquired knowledge. Our future work will explore the feasibility of employing continual learning for embedded devices.

One of the challenges to figuring out the *optimal* BERT-based architecture is the lack of application-specific (viz., voice-controlled robots for home automation) datasets. Existing datasets either *(a)* do not have enough examples for training deep learning models or *(b)* do not provide complex, practical queries to test the robustness of a given model. Building suitable datasets for IoT-specific human-centric applications such as voice-control home/industrial automation is an interesting open research problem.

6 RELATED WORK

We discuss related research on two fronts: *(a)* BERT-based models and their efficient variants and *(b)* using NLP on embedded devices.

6.1 BERT-based Models and their Variants

The performance of BERT comes at a high computation and memory cost, which makes on-device inference really challenging. To mitigate this issue, researchers have proposed knowledge distillation approaches from the original BERT model, for example, *(a)* "finetune" the BERT model to improve task-specific knowledge distillation [32, 33], *(b)* use Bi-LSTM models [34] for knowledge distillation from BERT, *(c)* leverage single-task models to teach a multi-task model [35], *(d)* distillation of knowledge from an ensemble of BERT into a single BERT [36], *(e)* TinyBERT [4] uses a layer-wise distillation strategy for BERT in both the pre-training and fine-tuning stages, and *(f)* DistilBERT [3] halves the depth of the BERT model through knowledge distillation in the pre-training stage and an optional fine-tuning stage. On a different direction, the *Patient Knowledge Distillation* approach [37] compresses an original large model ("teacher") into an equally-effective lightweight shallow network ("student"). Other BERT models (e.g., SqueezeBERT [38], MobileBERT [39], Q8BERT [40], ALBERT [41]) can also reduce resource consumption than the vanilla BERT. EdgeBERT [42], an algorithm-hardware co-design approach, performs latency-aware energy optimizations for multi-task NLP problems. However, unlike ours, EdgeBERT *(a)* does not apply attention heads pruning, and *(b)* does not report scores on downstream NLP tasks on real-world embedded systems.

6.2 NLP for Embedded Platforms

Researchers have explored NLP techniques to facilitate natural communication between humans and embedded devices, especially in the context of voice-controlled cognitive robots. For example, Megalingam et al. [43] presents a voice recognition tool that compares the user's input commands with the stored data. Zhang et al. [44] propose a ROS-based robot that analyzes commands using an offline grammar recognition library. Megalingam et al. [45] propose a cost-efficient speech processing module running on ROS that can provide natural language responses to the user. There also exists ROS-integrated independent speech recognition packages [46, 47] as well as Arduino-based [48] and custom [49] voice-control robot platforms. House et al. [50] a voice-controlled robotic arm (named VoiceBot) for individuals with motor impairments [51]. However, most of these works focused on rule-based approaches, and we note that transformer architectures are still under-explored in terms of their practical deployment challenges in real-world embedded and robotic devices, which is the focus of this study.

6.3 Uniqueness of Our Work

While existing work can reduce the size of BERT models through distillation and pruning, from a system design perspective, it is still difficult and tedious for a developer to find out the "right" BERT-based architecture to use in an embedded platform. To date, it is also unclear which lighter version of BERT would find the optimal balance between the resources available in an embedded device (e.g., CPU, GPU, memory) and the minimum accuracy desired. We used four off-the-shelf platforms widely used by developers for various IoT and embedded applications and benchmarked state-of-the-art BERT architectures. Our empirical evaluation and design space exploration on heterogeneous platforms (e.g., x86 and ARM, with or without GPU) can help the system and machine learning engineers to pick suitable architectures depending on target system configuration and performance constraints (e.g., accuracy, F_1 score). To the best of our knowledge, this work is *one of the first efforts to study the feasibility of deploying BERT-based models* in real-world resource-constrained embedded platforms.

7 CONCLUSION

This paper presents an empirical study of BERT-based neural architectures in terms of the feasibility of deploying them on resource-constrained systems, which have become ubiquitous nowadays. Our performance evaluation results will assist developers of multiple ubiquitous computing domains, such as voice-controlled home and industrial automation, precision agriculture, and medical robots to determine the deployability of NLP models in their target platform. By using our benchmark data, designers of ubiquitous systems will now be able to select the "right" hardware, architecture, and parameters depending on the resource constraints and performance requirements. This will also save time on the developer's end, as they can make informed choices regarding which BERT-based architecture to use during development based on their NLP application scenario and the available hardware.

ACKNOWLEDGMENTS

This work has been partially supported by the National Science Foundation Standard Grant Award 2302974, Air Force Office of Scientific Research Grant/Cooperative Agreement Award FA9550-23-1-0426, and Washington State University Grant PG00021441. Any findings, opinions, recommendations, or conclusions expressed in the paper are those of the authors and do not necessarily reflect the views of sponsors.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[3] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter," *CoRR*, vol. abs/1910.01108, 2019. [Online]. Available: http://arxiv.org/abs/1910.01108

[4] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling BERT for natural language understanding," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 4163–4174. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.372

[5] L. Derczynski, E. Nichols, M. van Erp, and N. Limsopatham, "Results of the WNUT2017 shared task on novel and emerging entity recognition," in *Proceedings of the 3rd Workshop on Noisy User-generated Text, NUT@EMNLP 2017, Copenhagen, Denmark, September 7, 2017*, L. Derczynski, W. Xu, A. Ritter, and T. Baldwin, Eds. Association for Computational Linguistics, 2017, pp. 140–147. [Online]. Available: https://doi.org/10.18653/v1/w17-4418

[6] D. Demszky, D. Movshovitz-Attias, J. Ko, A. Cowen, G. Nemade, and S. Ravi, "Goemotions: A dataset of fine-grained emotions," *arXiv preprint arXiv:2005.00547*, 2020.

[7] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[8] R. Thoppilan, D. D. Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, Y. Zhou, C. Chang, I. Krivokon, W. Rusch, M. Pickett, K. S. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. H. Chi, and Q. Le, "Lamda: Language models for dialog applications," *CoRR*, vol. abs/2201.08239, 2022. [Online]. Available: https://arxiv.org/abs/2201.08239

[9] R. Pi, "https://www.raspberrypi.com/," 2023, accessed:May 15. [Online]. Available: https://www.raspberrypi.com/

[10] Jetson, 2023, accessed:Sep 15. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[11] U. board, 2023, accessed:Sep 15. [Online]. Available: https://up-board.org/

[12] U. Bolt, 2023, accessed:Sep 15. [Online]. Available: https://www.udoo.org/discover-the-udoo-bolt/

[13] "Transformers bechmarking: Implementation and artifacts," 2024. [Online]. Available: https://github.com/CPS2RL/Benchmarking-BERT-Embedded-Devices-ICPE24-Artifact

[14] B.-H. Juang and L. R. Rabiner, "Automatic speech recognition–a brief history of the technology development," *Georgia Institute of Technology. Atlanta Rutgers University and the University of California. Santa Barbara*, vol. 1, p. 67, 2005.

[15] D. Yu and L. Deng, *Automatic speech recognition*. Springer, 2016, vol. 1.

[16] A. Vanzo, D. Croce, E. Bastianelli, R. Basili, and D. Nardi, "Grounded language interpretation of robotic commands through structured learning," *Artificial Intelligence*, vol. 278, 2020. [Online]. Available: https://doi.org/10.1016/j.artint.2019.103181

[17] E. F. Sang and F. De Meulder, "Introduction to the conll-2003 shared task: Language-independent named entity recognition," *arXiv preprint cs/0306050*, 2003.

[18] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[19] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[20] P. Michel, O. Levy, and G. Neubig, "Are sixteen heads really better than one?" in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 14 014–14 024. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/2c601ad9d2ff9bc8b282670cdd54f69f-Abstract.html

[21] UM25C, 2017. [Online]. Available: https://www.makerhawk.com/collections/frontpage/products/makerhawk-um25c-usb-tester-bluetooth-usb-meter-type-c-current-meter-usb-power-meter-dc-24-000v-5-0000a-usb-cable-tester-1-44-inch-color-lcd-multimeter-voltage-tester-usb-load-qc-2-0-qc-3-0

[22] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[23] E. Adamopoulou and L. Moussiades, "Chatbots: History, technology, and applications," *Machine Learning with Applications*, vol. 2, p. 100006, 2020.

[24] P. B. Brandtzæg and A. Følstad, "Why people use chatbots," in *Internet Science - 4th International Conference, INSCI 2017, Thessaloniki, Greece, November 22-24, 2017, Proceedings*, ser. Lecture Notes in Computer Science, I. Kompatsiaris, J. Cave, A. Satsiou, G. Carle, A. Passani, E. Kontopoulos, S. Diplaris, and D. McMillan, Eds., vol. 10673. Springer, 2017, pp. 377–392. [Online]. Available: https://doi.org/10.1007/978-3-319-70284-1_30

[25] H.-Y. Shum, X.-d. He, and D. Li, "From eliza to xiaoice: challenges and opportunities with social chatbots," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, pp. 10–26, 2018.

[26] R. W. White, "Skill discovery in virtual assistants," *Communications of the ACM*, vol. 61, no. 11, pp. 106–113, 2018.

[27] B. Schmidt, R. Borrison, A. Cohen, M. Dix, M. Gärtler, M. Hollender, B. Klöpper, S. Maczey, and S. Siddharthan, "Industrial virtual assistants: Challenges and opportunities," in *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers*, 2018, pp. 794–801.

[28] A. G. Oettinger, "Automatic language translation," in *Automatic Language Translation*. Harvard University Press, 2013.

[29] P. F. Brown, J. Cocke, S. A. Della Pietra, V. J. Della Pietra, F. Jelinek, R. L. Mercer, and P. Roossin, "A statistical approach to language translation," in *Coling Budapest 1988 Volume 1: International Conference on Computational Linguistics*. John von Neumann Society for Computing Sciences, Budapest, 1988.

[30] M. De Lange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, "A continual learning survey: Defying forgetting in classification tasks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 7, pp. 3366–3385, 2021.

[31] R. Hadsell, D. Rao, A. A. Rusu, and R. Pascanu, "Embracing change: Continual learning in deep neural networks," *Trends in cognitive sciences*, vol. 24, no. 12, pp. 1028–1040, 2020.

[32] I. Turc, M. Chang, K. Lee, and K. Toutanova, "Well-read students learn better: The impact of student initialization on knowledge distillation," *CoRR*, vol. abs/1908.08962, 2019. [Online]. Available: http://arxiv.org/abs/1908.08962

[33] H. Tsai, J. Riesa, M. Johnson, N. Arivazhagan, X. Li, and A. Archer, "Small and practical BERT models for sequence labeling," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 3630–3634. [Online]. Available: https://doi.org/10.18653/v1/D19-1374

[34] R. Tang, Y. Lu, L. Liu, L. Mou, O. Vechtomova, and J. Lin, "Distilling task-specific knowledge from BERT into simple neural networks," *CoRR*, vol. abs/1903.12136, 2019. [Online]. Available: http://arxiv.org/abs/1903.12136

[35] K. Clark, M. Luong, U. Khandelwal, C. D. Manning, and Q. V. Le, "Bam! born-again multi-task networks for natural language understanding," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, A. Korhonen, D. R. Traum, and L. Màrquez, Eds. Association for Computational Linguistics, 2019, pp. 5931–5937. [Online]. Available: https://doi.org/10.18653/v1/p19-1595

[36] X. Liu, Y. Wang, J. Ji, H. Cheng, X. Zhu, E. Awa, P. He, W. Chen, H. Poon, G. Cao, and J. Gao, "The microsoft toolkit of multi-task deep neural networks for natural language understanding," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, ACL 2020, Online, July 5-10, 2020*, A. Celikyilmaz and T. Wen, Eds. Association for Computational Linguistics, 2020, pp. 118–126. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-demos.16

[37] S. Sun, Y. Cheng, Z. Gan, and J. Liu, "Patient knowledge distillation for BERT model compression," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 4322–4331. [Online]. Available: https://doi.org/10.18653/v1/D19-1441

[38] F. N. Iandola, A. E. Shaw, R. Krishna, and K. Keutzer, "Squeezebert: What can computer vision teach NLP about efficient neural networks?" in *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing, SustaiNLP@EMNLP 2020, Online, November 20, 2020*, N. S. Moosavi, A. Fan, V. Shwartz, G. Glavas, S. R. Joty, A. Wang, and T. Wolf, Eds. Association for Computational Linguistics, 2020, pp. 124–135. [Online]. Available: https://doi.org/10.18653/v1/2020.sustainlp-1.17

[39] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "Mobilebert: a compact task-agnostic BERT for resource-limited devices," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, Eds. Association for Computational Linguistics, 2020, pp. 2158–2170. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.195

[40] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8BERT: quantized 8bit BERT," in *Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition, EMC2@NeurIPS 2019, Vancouver, Canada, December 13, 2019*. IEEE, 2019, pp. 36–39. [Online]. Available: https://doi.org/10.1109/EMC2-NIPS53020.2019.00016

[41] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite BERT for self-supervised learning of language representations," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=H1eA7AEtvS

[42] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E. Yang, M. Donato, V. Sanh, P. N. Whatmough, A. M. Rush, D. Brooks, and G. Wei, "Edgebert: Sentence-level energy optimizations for latency-aware multi-task NLP inference," in *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 2021, pp. 830–844. [Online]. Available: https://doi.org/10.1145/3466752.3480095

[43] R. K. Megalingam, R. S. Reddy, Y. Jahnavi, and M. Motheram, "Ros based control of robot using voice recognition," in *2019 Third International Conference on Inventive Systems and Control (ICISC)*, 2019, pp. 501–507.

[44] Y. Zhang, Z. Lu, C. Wang, C. Liu, and Y. Wang, "Voice control dual arm robot based on ros system." 2018 IEEE International Conference on Intelligence and Safety for Robotics (ISR), 2018, pp. 232–237.

[45] R. K. Megalingam, A. H. Kota, and V. K. T. P., "Optimal approach to speech recognition with ros," in *2021 6th International Conference on Communication and Electronics Systems (ICCES)*, 2021, pp. 111–116.

[46] Y. Zhang and S. C. Xu, "Ros based voice-control navigation of intelligent wheelchair," in *Engineering Decisions for Industrial Development*, ser. Applied Mechanics and Materials, vol. 733. Trans Tech Publications Ltd, 4 2015, pp. 740–744.

[47] S. Sharan, T. Q. Nguyen, P. Nauth, and R. Araujo, "Implementation and testing of voice control in a mobile robot for navigation," in *2019 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*. IEEE, 2019, pp. 145–150.

[48] A. Verma, D. Kumar, H. Maurya, A. Kumar, and M. P. Dwivedi, "Voice control robot using arduino," *International Research Journal of Modernization in Engineering Technology and Science*, vol. 2, p. 04, 2020.

[49] X. Lv, M. Zhang, and H. Li, "Robot control based on voice command," in *2008 IEEE International Conference on Automation and Logistics*. IEEE, 2008, pp. 2490–2494.

[50] B. House, J. Malkin, and J. A. Bilmes, "The voicebot: a voice controlled robot arm," in *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, D. R. O. Jr., R. B. Arthur, K. Hinckley, M. R. Morris, S. E. Hudson, and S. Greenberg, Eds. ACM, 2009, pp. 183–192. [Online]. Available: https://doi.org/10.1145/1518701.1518731

[51] J. A. Bilmes, X. Li, J. Malkin, K. Kilanski, R. Wright, K. Kirchhoff, A. Subramanya, S. Harada, J. Landay, P. Dowden, and H. Chizeck, "The vocal joystick: A voice-based human-computer interface for individuals with motor impairments," in *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*. Vancouver, British Columbia, Canada: Association for Computational Linguistics, Oct. 2005, pp. 995–1002. [Online]. Available: https://aclanthology.org/H05-1125

Optimizing Edge AI: Performance Engineering in Resource-Constrained Environments

Giuliano Casale
Department of Computing
Imperial College London
United Kingdom
g.casale@imperial.ac.uk

ABSTRACT

Recent years have witnessed the growth of Edge AI, a transformative paradigm that integrates neural networks with edge computing, bringing computational intelligence closer to end users. However, this innovation is not without its challenges, especially in environments with limited computing, network, and memory constraints, where resource-hungry AI models often need to be partitioned for distributed execution. This issue becomes even more acute in scenarios where post-deployment updates are infeasible or costly, posing a need to accurately reason about the interplay between resource constraints and Quality-of-Service (QoS) in Edge AI systems, so as to optimally design and operate them.

In this keynote talk, I will focus on these challenges, discussing QoS management and deployment problems arising in Edge AI systems. I will review mechanisms such as early exits and DNN partitioning that are distinctive of this problem space, explaining how they could be accounted for and leveraged in system performance and reliability tuning. I will then illustrate how design decisions and the definition of novel runtime control algorithms can be guided by approaches based on both traditional analytical models and emerging data-driven methods based on machine learning models.

CCS CONCEPTS

• **General and reference → Performance**; • **Computing methodologies → Modeling and simulation**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

KEYWORDS

Edge AI, performance, reliability, tuning

ACM Reference Format:
Giuliano Casale. 2024. Optimizing Edge AI: Performance Engineering in Resource-Constrained Environments. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 1 page. https://doi.org/10.1145/3629526.3649131

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3649131

SPEAKER BIO

Giuliano Casale is a Reader in the Department of Computing at Imperial College London. He does research in Quality-of-Service engineering and cloud computing, topics on which he has published more than 150 refereed papers. He has served as program co-chair for several conferences in the area of performance and reliability engineering, such as ACM SIGMETRICS/Performance and IEEE/IFIP DSN. His research work has received best paper awards at ACM SIGMETRICS, IEEE/IFIP DSN and IEEE INFOCOM. During 2019-2023, he has served as ACM SIGMETRICS chair. He serves on the editorial board of ACM TOMPECS and as Editor-in-Chief of Elsevier Performance Evaluation.

TBASCEM - Tight Bounds with Arrival and Service Curve Estimation by Measurements

Christoph Funda
ZF Mobility Solutions GmbH
Test System Development
Ingolstadt, Germany
christoph.funda@zf.com

Thomas Herpel
ZF Mobility Solutions GmbH
System Qualification & Product Test Strategy
Ingolstadt, Germany
thomas.herpel@zf.com

Reinhard German
Friedrich-Alexander-Universität Erlangen-Nürnberg
Computer Networks and Communication Systems
Erlangen, Germany
german@informatik.uni-erlangen.de

Kai-Steffen Jens Hielscher
Friedrich-Alexander-Universität Erlangen-Nürnberg
Computer Networks and Communication Systems
Erlangen, Germany
Kai-Steffen.Hielscher@informatik.uni-erlangen.de

ABSTRACT

This paper aims to solve the challenge of quantifying the performance of Hardware-in-the-Loop (HIL) computer systems used for data re-injection. The system can be represented as a multiple queue and server system that operates on a First-In, First-Out (FIFO) basis. The task at hand involves establishing tight bounds on end-to-end delay and system backlog. This is necessary to optimise buffer and pre-buffer time configurations. Network Calculus (NC) is chosen as the basic analytical framework to achieve this. In the literature, there are different techniques for estimating arrival and service curves from measurement data, which can be used for NC calculations. We have selected four of these methods to be applied to datasets of industrial Timestamp Logging (TL). The problem arises because these conventional methods often produce bounds that are much larger (by a factor of 1000 or more) than the measured maximum values, resulting in inefficient design of HIL system parameters and inefficient resource usage. The proposed approach, called TBASCEM, introduces a reverse engineering approach based on linear NC equations for estimating the parameters of arrival and service curves. By imposing constraints on the equation variables and employing non-linear optimization, TBASCEM searches for a burst parameter estimation which derives tight global delay bounds. In addition, TBASCEM simplifies the run-time measurement process, supporting real-time data acquisition to evaluate and optimise HIL system performance, and enhancing observability to adapt the HIL configuration to new sensor data. The benefits of TBASCEM are clearly that it enables an efficient performance logging of arrival and service curve parameters and with deriving tighter bounds in HIL systems, compared to evaluated state-of-the-art methods, making TBASCEM an invaluable tool for optimising and monitoring streaming applications in non-hard-real-time environments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05…$15.00
https://doi.org/10.1145/3629526.3645031

CCS CONCEPTS

• **General and reference** → **General conference proceedings**; • **Computer systems organization** → **Embedded and cyber-physical systems**; • **Software and its engineering** → *Empirical software validation*; *Operational analysis*; • **Hardware** → *Sensors and actuators*; • **Information systems** → *Computing platforms.*

KEYWORDS

Performance Evaluation and Monitoring, Hardware-in-the-loop Test System, Streaming System, Network Calculus, Arrival and Service Curve Estimation

ACM Reference Format:
Christoph Funda, Thomas Herpel, Reinhard German, and Kai-Steffen Jens Hielscher. 2024. TBASCEM - Tight Bounds with Arrival and Service Curve Estimation by Measurements. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645031

1 INTRODUCTION

Autonomous vehicle technology requires millions of kilometres of recorded sensor data to be replayed with HIL systems to verify the perception technology on the Device Under Test (DUT). This process absorbs a significant amount of time and energy for the computing resources, as the recordings must be replayed in real-time from the HIL to the DUT. In addition, the replay must accurately reproduce a real-world scenario to ensure high quality HIL performance. It is essential that there is no data loss within the HIL system and that the timing to the DUT is accurate and precise, as needed by the DUT. We dive deeper into that topic in the Section 2. However, many HIL systems, beyond their hardware interfaces to the DUT, do not fall under the category of hard real-time systems. Instead, they fall into the realm of soft and non-hard real-time systems.

How can non-hard real-time computing machines meet the hard real-time requirements of the DUT? One approach involves utilizing a playback-buffer on the hard real-time capable interface-card. By filling the buffer with sufficient data before streaming to the DUT, uninterrupted streaming can be achieved.

The challenge lies now in designing the buffer size and pre-buffer time appropriately. To do so, it is crucial to determine the worst-case end-to-end delay and backlog bounds for buffer size dimensions and the pre-buffer time parameter.

Filling the playback buffer creates idle time before streaming to the DUT, leading to wasted time and energy of HIL computing resources. Hence, minimising pre-buffering time is crucial.

On the contrary, it is crucial to avoid an empty playback buffer during re-injection, as it disturbs the precise timing needed by mostly hard real-time DUTs. This hence leads to failed tests and again a loss of time and energy at the HIL computing resources. To achieve an efficient HIL performance, it is essential to adjust the pre-buffer time to be at least equal to or greater than the worst case end-to-end delay. The aim is to fill the playback buffer with a minimum of the burst parameter of the arrival curve of the sensor stream.

The methodology involves measuring the HIL system before operation, designing it accordingly, and monitoring its performance during operation to optimise its parameters if necessary. Using the measured maximum end-to-end delay directly would overestimate the required time, as waiting time in a queue could be included. Hence, it is necessary to consider queuing theory, which is accomplished by employing NC: a deterministic queuing system theory. Using NC, bounds for the backlog and end-to-end delay of a queuing system can be computed from its arrival and service curves. However, NC bounds should be tight, for our use-case and our system, at least the end-to-end delay bound needs to be as tight as possible, this is the major challenge.

There are basic concepts in the NC framework and mathematically defined in all basic literature about NC [13, 14]. In short, an arrival curve is the upper constraint of an input flow, and a service curve is the lower constraint of a flow provided by a service. The basic linear arrival curves and service curves, made used off in this work, are burst-rate curves and rate-latency curves. In hard real-time systems, their parameters can be found easily, they are often directly defined. However, in our non-real-time HIL system under study, they are not. So, they need to be measured, what is challenging. The analytical solutions for streaming devices with NC has been derived by Le Boudec et al. in [14]. We adapted and applied the NC solutions by linear equations on measurement data from a HIL test system in [11]. However, the results in [11] showed us, that there are improvements of the derived bounds needed for the estimation methods of the system service curves, to make them usable in practice.

One of the approaches mentioned in the survey study by Fidler et al. [8] to generate a strict service curve based on TL of the input and output of queuing server systems by Alcuri et al. [1] are used here in this paper. However, the results show, that the bounds are not tight at all with these methods. They often exceed by a factor of 1000.

Additionally, all the measurement methods, which are discussed in the survey, are based upon TL from the ingress and egress of the system. We are specifically interested to reduce the TL, to be able to do run-time measurements at the HIL system during operation, without generating massive data and performance overhead for the network or processor. On one hand, TL are occupying the memory. On the other hand, the TL need to be sent via network, what would occupy the processor and the network during operation.

To close the gap of estimating a service curve by measurements, with the aim to provide tight delay and backlog bounds of a streaming process within a computer system, without producing lots of TL, a new approach has been developed called TBASCEM — Tight Bounds with Arrival and Service Curve Estimation by Measurements. TBASCEM aims to provide a performant and efficient estimation method from TL as well as a technique for reducing the needed measurement data for arrival and service curve estimation, to be able to calculate tight bounds with NC.

In summary, our novel TBASCEM approach fills the gap for an effective and efficient method to measure and estimate service curves by measurements with the aim to calculate tight bounds, while maintaining low impact on CPU performance and memory allocation during operation.

The remainder of this paper is structured as follows: In Section 2 we start with basic technical backgrounds of HIL systems. In Sect. 3 we introduce our TBASCEM approach. We start with the description of the concept idea and the needed reduced measurements, followed by the reverse engineering approach for the estimation arrival and service curve parameters. In the following Section 4, we define our research questions, with a short explanation of the methodology on how to evaluate them and our hypothesis, followed by a description of the experimental setting, and finishing with discussing the results. In the following Section 5 we describe former work in literature and how it relates to our work. Finally, we end up with Section 6, where we conclude with a short summary and the contribution of our work. Our contributions include:

- Evaluating the tightness of delay and backlog bounds derived from four state-of-the-art service curve estimation methods by TL from the HIL test system with industrial workload, which highlights the need for TBASCEM, to analyse the performance of non-hard real-time streaming systems.
- Derivation of the TBASCEM approach by reverse-engineering of NC solution by linear arrival and service curves.
- Setting up of the optimization equation as well as the constraints for the TBASCEM algorithm.
- Implementing the TBASCEM Run-Time Measurement (RTM) in LabVIEW and the TBASCEM service curve generation in MATLAB for a proof-of-concept and performance evaluation.
- Quantitatively comparing the CPU performance of state-of-the-art TL with our TBASCEM RTM.
- Quantitatively comparing the bound tightness of four state-of-the-art approaches for estimating service curves from measurement with our TBASCEM estimation approach.

2 FUNDAMENTALS

HIL test systems have emerged as invaluable tools in computer science research and engineering. Their ability to seamlessly integrate virtual simulations with physical hardware empowers researchers and engineers to evaluate and optimize complex systems efficiently. With applications ranging from software testing to fault diagnosis, HIL test systems continue to shape the future of innovative technologies.

The following subsections of this section will confidently explore important aspects of HIL test systems, with a specific focus on real-time constraints and designing HIL test systems based on NC.

2.1 Real-Time Constraints of HIL Systems

In HIL systems, the processing time of software or network processes is heavily influenced by the computing machine on which the processes run. For hard real-time systems, the worst-case processing time serves as a critical performance indicator, imposing stringent requirements on the computing machine. Hard real-time systems are costly but offer formal evaluation and guarantees of their Worst-Case Execution Time (WCET) through measurements of processing cycles and CPU frequency.

However, many HIL systems, beyond their hardware interfaces to the DUT, do not fall under the category of hard real-time systems. Instead, they fall into the realm of soft and non-real-time systems. For such systems, formal evaluation cannot be done and guarantees for a WCET cannot be given due to various factors. Processing times are influenced by caching mechanisms, memory system hierarchies, CPU frequency fluctuations during run-time, hardware travel times, interrupt requests, context switches, and other features of modern computer and operating systems. Additional delays and processing times may arise when software systems use middleware, making accurate measurement difficult with high variation. For instance, interprocess communication using a localhost network connection in the robot operating system (ROS) [18] exemplifies this complexity, like applied in the HIL test system [10].

In these non-real-time systems, relying solely on an observed WCET can also be overly pessimistic, leading to bottleneck assumptions and infinite bounds in theory, while practical results suggest more leniency, as demonstrated by us in [11]. Despite their limitations, non-real-time systems are more cost-effective, and their software development is simpler and less expensive. For many cases, a service based on mean-rate or even stochastic bounds suffices. The occasional timing overshot does not result in catastrophic events if overshoots are monitored and documented in the test results.

In practice, closed-loop simulations running on HIL systems often employ monitoring mechanisms to detect simulation service performance during run-time. Any task overruns are logged as warnings or errors, prompting re-evaluation of tests in case of excessive, unwanted overruns [7].

In the case of open-loop re-injection, the HIL system functions as a streaming device. It streams measured input data like previously captured sensor data to the DUT. The DUT processes the data, and the output is streamed back to the HIL system for evaluation as test results. This setup allows for comprehensive testing and evaluation of complex cyber-physical systems, providing insights into their behaviour and performance in real-world scenarios.

2.2 Design of HIL Test Systems with NC

To provide bounds for end-to-end delay and buffer size design of queuing systems, NC is a possible analytical framework to work with. NC was introduced by Cruz et al. in 1991 [6]. The analytical solutions for streaming devices have been derived by Le Boudec et al. in [14]. It facilitates the establishment of strict yet secure buffer and delay bounds in these kinds of streaming systems.

We adapted and applied the NC solutions by linear equations on measurement data from a HIL test system in [11]. But as the findings in [11] demonstrated, there is a need to refine the derived boundaries in order to make the system service curve estimate techniques practical The derived bounds with the WCET method are often growing to infinity.

To fill this gap, we started a review study on practical measurement methods for arrival and service curves and applied them to the HIL system, published in [9]. The methods for estimating service curves by measurement data proposed by Helm et al. [12] and Wandeler et al. [20] were applied by us in [9] on TL from a HIL test system. We used realistic industrial workload for streaming data to CAN and Automotive Ethernet interfaces for re-injection to the DUT. The study showed that the delay and backlog bounds, generated by these estimated service curves are not tight enough in all cases, to be used in practice. Compared to the maximum measured backlog or end-to-end delay, the calculated bounds were a factor of over 10k higher.

The subsequent chapter of this paper introduces our new approach TBASCEM, which enables monitoring of the streaming performance like latency, rate, and backlog of the HIL systems during operation. It estimates arrival and service curves by using measurements which provide tight bounds for delay and backlog.

3 TBASCEM APPROACH

Our proposed concept aims to measure and estimate arrival and service curves for non-hard real-time software services in a streaming chain of a HIL system. This methodology can be applied to other streaming systems, including video streaming services, to accurately measure worst-case latency, backlog, and estimate burst parameters. Furthermore, it can estimate arrival and service curves from this data for tight NC bound calculations.

The approach relies on an iterative online algorithm derived by reverse engineering of linear NC bound equations. It replaces the conventional TL used for evaluating the processing performance of software modules. By doing so, the amount of logging data is significantly reduced. Instead of saving two timestamps per message, our method only requires saving three variables in total for one software process, based on timestamps and the number of messages in the queue. The iterative calculation process is computationally efficient, making it suitable for real-time operation of a HIL test bench in streaming mode.

The estimated service curves provide an overview of the HIL system's performance during operation, offering insights into any system influences and changes. By logging system data and storing these curves in a database, it is possible to calculate the likelihood of these service curves and analyse system affects more easily. This information may then be utilized with the stochastic NC framework.

Basically, the service curves play a crucial role in determining delay and backlog bounds with NC. If the HIL system needs to process new input data with a different arrival curve behaviour, the buffer size and pre-buffer time parameter will require adjustments. Our concept helps suggest an appropriate pre-buffer time to prevent buffer underflow at the playback buffer, and can also save time if the recommended pre-buffer time is significantly lower than the initially designed value. Furthermore, it allows for predicting the

required buffer size between each software service, with a defined safety factor, represented by the desired tightness factor.

3.1 Concept Idea of TBASCEM

The following Figure shows the queue and server system that can be abstracted by our streaming SW in the HIL system and the available measurement points.

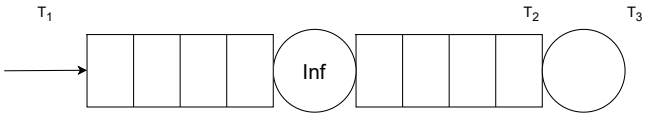


Figure 1: Queue and software service

During operation, our algorithm generates fundamental measurements by monitoring two key metrics: the maximum delay and maximum backlog between the input to a queue of a server (represented by T_1 in Figure 1) and the output of the server (represented by T_3 in Figure 1). To compute the burst parameter of the arrival curve, we utilize the given mean-rate from the original data and determine the vertical deviation from the mean-rate curve in each iteration, like illustrated in Figure 2a. We then store only the maximum positive and minimum negative values of this deviation.

The objective of the reverse engineering algorithm is to derive the rate-latency service curve parameters using the measured worst-case delay, backlog and burst parameter. The final reverse calculation from these three parameters to the service curve parameters are illustrated in Figure 2b. Furthermore, Figure 2c illustrates the reverse calculation in the negative burst domain.

The algorithm takes the following data as input for its calculations: for each packet i, the timestamp of the packet, t_i, and the number of bytes at each packet, b_i. During operation, we measure the maximum queue length, q_{max}, and the maximum delay, l_{max}. The algorithm utilizes the inter-arrival time Δt_i, which represents the time difference between two successive incoming packets at time T_i and time T_{i-1}.

$$\Delta t_i = T_i - T_{i-1} \tag{1}$$

The algorithm used for the arrival curve run-time measurement initially calculates the cumulative sum of inter-arrival time for each step i:

$$\sum_{i=0}^{i} \Delta t_i = \tilde{T}_i \tag{2}$$

Let $\tilde{T}_i$ represent the time passed until packet i, measured in seconds, and $\tilde{B}_i$ denote the total number of bits sent until packet i:

$$\sum_{i=0}^{i} b_i = \tilde{B}_i \tag{3}$$

The mean input rate, r_{in}, is calculated from the timestamp T_0 of the measurement data, which will be re-injected to the DUT.

Next, we iteratively compute the deviation from the ideal curve (see Figure 2a):

$$r_{in} \cdot \tilde{T}_i - \tilde{B}_i = \Delta b_i \tag{4}$$

Where Δb quantifies the deviation in terms of bits or messages.

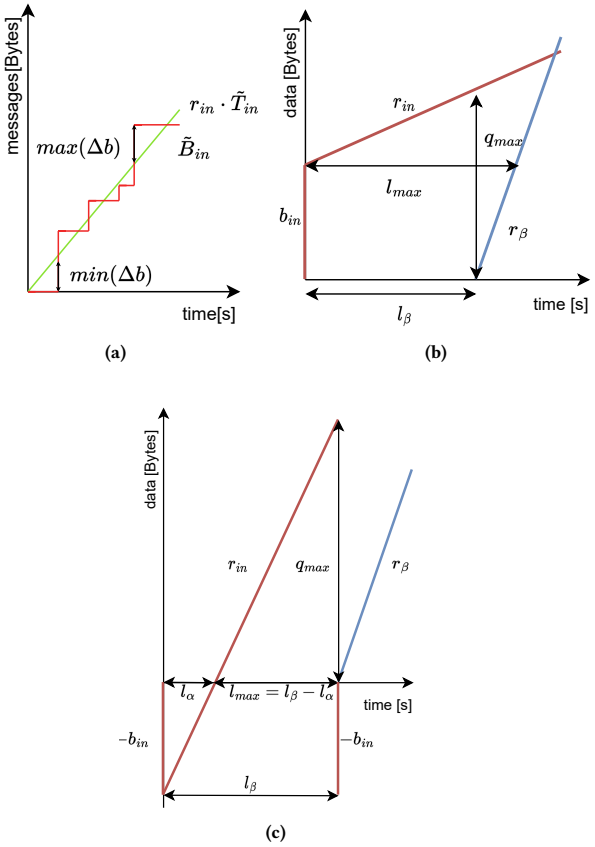


Figure 2: (a) Visualisation of burst parameter Δb estimation for the arrival curve; (b) l_β and r_β parameter estimation of the service curve ; (c) NC solution with negative burst

The algorithm determines the burst parameter in the arrival curve during operation as follows:

- Measure $min(\Delta b_i)$ and store it as b_{min} if the value is smaller than the previous saved value.
- Measure $max(\Delta b_i)$ and store it as b_{max} if the value is higher than the previous value.
- Calculate $b_{in_{pos}}$ as $(\Delta b_i) - b_{min}$ and update the maximum value of $b_{in_{pos}}$ if the calculated value is higher than the old value.
- Calculate $b_{in_{neg}}$ as $b_{max} - (\Delta b_i)$ and update the maximum value of $b_{in_{neg}}$ if the calculated value is higher than the old value.

The TBASCEM approach for estimating the service curve rate and latency parameters is based on the following measurement values:

- burst parameter b_{in} of the arrival curve at the service input, divided into a positive $b_{in_{pos}}$ and a negative $b_{in_{neg}}$ deviation from the mean rate

- burst parameter b_{out} of the arrival curve at the service output, divided into a positive $b_{out_{pos}}$ and a negative $b_{out_{neg}}$ deviation from the mean rate
- maximum latency l_{max} between service input and output
- maximum queue length q_{max} between service input and output, however, we determine the final maximum queue length as the maximum of q_{max} and $b_{out_{pos}}$. This approach is based on the NC theory, as the output bound is equal to the backlog bound.

3.2 Reverse Engineering Algorithm

The reverse engineering approach to calculate the service curve parameter rate r_β and latency l_β, involves the following equations, based on the basic geometric connections shown in Figure 2b.

For the calculation of the latency parameter l_β, the following equation can be used:

$$l_\beta = \frac{q_{max} - b_{in}}{r_{in}} \tag{5}$$

The parameter l_β is expressed in seconds.

Next, we obtain the rate parameter r_β using the following equation:

$$r_\beta = \frac{b_{in}}{l_{max} - l_\beta} \tag{6}$$

where the parameter r_β is given in units of $\left[\frac{Bytes}{s}\right]$ or $\left[\frac{bit}{s}\right]$ or $\left[\frac{msg}{s}\right]$.

However, this reverse engineering approach has two risks of incorrect parameter estimation: The first risk occurs when the numerator of Equation (5) becomes ≤ 0, resulting in $l_\beta \leq 0$. This can happen if the approximation of Δb from the input flow is overestimated. The second incorrect estimation can occur when the denominator of Equation (6) becomes smaller than 0, leading to r_β becoming negative. This can happen if l_β is higher than l_{max}, what can be caused by an underestimated b_{in}.

To handle these risks, we introduce the estimation parameter $\widetilde{b_{in}}$ and use it instead of the measured parameter b_{in}, to distinguish it from the measurements.

To determine a good value for $\widetilde{b_{in}}$, we furthermore analyse the tightness calculation of the two bounds, with d_{bound} as the delay bound and t_D as the delay bound tightness. Respectively, we analyse the backlog bound with q_{bound} and t_B as the backlog bound tightness. We establish the equations for calculating the tightness of the bounds, starting with the backlog bound:

$$q_{bound} = r_{in} \cdot \widetilde{l_\beta} + b_{in} \tag{7}$$

$$t_B = \frac{q_{bound}}{q_{max}} = \frac{r_{in} \cdot \frac{q_{max} - \widetilde{b_{in}}}{r_{in}} + b_{in}}{q_{max}} = \frac{q_{max} + b_{in} - \widetilde{b_{in}}}{q_{max}} \tag{8}$$

The tightness needs to be ≥ 1 to ensure that the bounds are not undershot. Therefore, an additional constraint is derived from Equation (8):

$$C3 : \widetilde{b_{in}} \leq b_{in} \tag{9}$$

The tightness formula for the delay bound is set up as follows:

$$d_{bound} = \frac{b_{in}}{\widetilde{r_\beta}} + \widetilde{l_\beta} \tag{10}$$

We calculate the tightness of the delay bound using the following equation:

$$t_D = \frac{d_{bound}}{l_{max}} = \frac{\frac{b_{in}}{\widetilde{b_{in}}} + \widetilde{l_\beta}}{l_{max}} = \frac{\frac{b_{in}}{\widetilde{b_{in}}} \cdot l_{max} + \frac{q_{max} - \widetilde{b_{in}}}{r_{in}} \cdot (1 - \frac{b_{in}}{\widetilde{b_{in}}})}{l_{max}} \tag{11}$$

To ensure that the delay bound is not undershot, the tightness needs to be ≥ 1, and from equation 11, we get the additional constraint:

$$t_D \geq 1 \Longrightarrow \frac{b_{in}}{\widetilde{b_{in}}} + \frac{q_{max} - \widetilde{b_{in}}}{r_{in} \cdot l_{max}} \cdot (1 - \frac{b_{in}}{\widetilde{b_{in}}}) \geq 1 \tag{12}$$

Tightness is important because loose bounds lead to inefficient HIL system design. Specifically, in the case of a backlog bound, it leads to a waste of memory resources, and in the case of a delay, it leads to a loss of time.

We derive the following two conditions from Equation (5) and (6) that must be handled separately to derive a service curve with valid bounds by the measurements.

3.2.1 Basic Conditions for Valid Bounds. The basic assumption is that there are no bottlenecks in the service. So, we set up the following inequality at first and derive the basic condition from it.

$$\begin{aligned} & & A1 : r_\beta \geq r_{in} \\ \Longrightarrow \frac{\widetilde{b_{in}}}{l_{max} - l_\beta} \geq r_{in} & \Longleftrightarrow \frac{\widetilde{b_{in}}}{lmax - \frac{q_{max} - \widetilde{b_{in}}}{r_{in}}} \geq r_{in} \\ & \Longleftrightarrow \widetilde{b_{in}} \geq r_{in} \cdot l_{max} + \widetilde{b_{in}} - q_{max} \\ & \Longrightarrow CD1 : \frac{q_{max}}{l_{max}} \geq r_{in} \end{aligned} \tag{13}$$

The assumption $A1$ and setting in Equation (6) leads to the condition $CD1$, what we prove at first. If the condition is not fulfilled, we assume another possible theoretical solution. If $l_\beta > l_{max}$ the inequality changes to the following term:

$$\begin{aligned} A1 : r_\beta \geq r_{in} \Longrightarrow \frac{\widetilde{b_{in}}}{l_{max} - l_\beta} \geq r_{in} \\ If : \ l_{max} - l_\beta \leq 0 \Longrightarrow \widetilde{b_{in}} \leq r_{in}(l_{max} - l_\beta) \\ \widetilde{b_{in}} \leq r_{in}(l_{max} - \frac{q_{max} - \widetilde{b_{in}}}{r_{in}}) \Leftrightarrow \widetilde{b_{in}} \leq rin \cdot l_{max} + \widetilde{b_{in}} - q_{max} \\ \Longrightarrow CD2 : \frac{q_{max}}{l_{max}} \leq r_{in} \end{aligned} \tag{14}$$

The visualisation of the theoretical concept for $CD2$ and the negative solution space of $\widetilde{b_{in}}$ is shown in Figure 2c.

We know that the arrival curve shown in Figure 2c is not a valid arrival curve according to the NC definition in the min-plus algebra, because it does not fulfil the subadditivity property. However, in the

max-plus algebra and in the Real-Time Calculus (RTC) framework, it is used. So, we also use it as an arithmetic solution to estimate the service curve parameters if the condition $CD1$ is not fulfilled and a possible solution can be found in the negative solution space. We later use the arrival curve with the positive burst parameter for the calculation of the bounds.

Also, the equation (14) for the service rate is not geometrically tractable, and cannot directly be applied to the solution. We can assume that there is no bottleneck and that the service rate corresponds to the input rate, and the input rate can be estimated by this formula. So, the inequality in Equation (14) is still valid.

However, estimating the $\widetilde{b}_{in}$ parameter may lead to overestimation, as the maximal and minimal deviation from the average rate, like $b_{in_{pos}}$ and $b_{in_{neg}}$ are measured, may not necessarily occur at subsequent arrivals, as shown in Figure 2a. The deviation could happen at entirely different epochs in the trace.

To find a suitable value for $\widetilde{b}_{in}$ that ensures a valid calculation of the service rate and latency parameter, it must be located within certain limits, not necessarily matching the measured value $b_{in_{pos}}$. We derive the constraints for these limits in the following sections.

3.2.2 Basic Constraints for Valid Bounds under Condition 1. To ensure a valid value for the latency l_β parameter, the following constraint needs to be met, based on Equation (5):

$$q_{max} - \widetilde{b}_{in} \geq 0 \Longrightarrow C1 : \widetilde{b}_{in} \leq q_{max} \tag{15}$$

To ensure a valid value for the rate parameter, the following constraint needs to be fulfilled, based on Equation (6):

$$l_{max} - l_\beta > 0 \; with (5) \Longrightarrow C2 : \widetilde{b}_{in} > q_{max} - l_{max} \cdot r_{in} \tag{16}$$

Since the burst parameter under Condition 1 should not be smaller than 0, a final constraint is added:

$$C4 : \widetilde{b}_{in} \geq 0 \tag{17}$$

3.2.3 Basic Constraints for Valid Bounds under Condition 2. Since the burst parameter under Condition 2 should be smaller than 0, a first constraint $C5$ is added:

$$C5 : \widetilde{b}_{in} \leq 0 \tag{18}$$

To ensure a valid value for the latency l_β parameter, the constraint needs to be met, based on Equation (5):

$$C6 : q_{max} - \widetilde{b}_{in} \geq 0 \Longrightarrow \widetilde{b}_{in} \leq q_{max} \tag{19}$$

To ensure a valid value for the rate parameter, the following constraint needs to be fulfilled, based on $CD2$ described in Equation (14):

$$l_{max} - l_\beta < 0 \; with \; (5) \implies C7 : \widetilde{b}_{in} < q_{max} - l_{max} \cdot r_{in} \tag{20}$$

The final constraint is the same as constraint 3 derived by Equation (8):

$$C8 : \widetilde{b}_{in} \leq b_{in} \tag{21}$$

3.2.4 The Optimization Function — Finding a Solution for the Arrival and Service Curve Parameter Estimation Problem. As a basis, we set up an optimization function which shall be minimized. It can be derived by finding the minimum of both summed tightness factors t_D and t_B:

$$\begin{aligned} & t_B + t_D \\ = \frac{q_{max} + b_{in} - \widetilde{b}_{in}}{q_{max}} + \frac{b_{in}}{\widetilde{b}_{in}} + \frac{q_{max} - \widetilde{b}_{in}}{r_{in} \cdot l_{max}} \cdot (1 - \frac{b_{in}}{\widetilde{b}_{in}}) & \\ = \underbrace{-(\frac{q_{max} + l_{max} r_{in}}{q_{max} l_{max} r_{in}}) \widetilde{b}_{in}}_{linear\ part} + \ldots & \\ \underbrace{(b_{in} - \frac{b_{in} q_{max}}{r_{in} l_{max}}) \frac{1}{\widetilde{b}_{in}}}_{hyperbolic\ part} + \ldots & \\ \underbrace{(1 + \frac{b_{in}}{q_{max}} + \frac{b_{in} + q_{max}}{r_{in} l_{max}})}_{constant\ part} & \end{aligned} \tag{22}$$

The optimization function (22) consists of a linear, a hyperbolic, and a constant part, using $\widetilde{b}_{in}$ as the variable. We use the built-in MATLAB function *solve*, to solve it as an optimization problem fulfilling either constraint 1-4 and inserting $b_{in_{pos}}$ as b_{in} for condition 1, or constraint 5-8 and inserting $b_{in_{neg}}$ as b_{in} for condition 2. However, the minimum could also be derived by graphical analysis of the optimization function (22) within the constraints.

In the first step of our algorithm, we verify, if condition $CD1$ holds true. Using the constraints $C1$ to $C4$ and $b_{in_{pos}}$ as the parameter b_{in} in the optimization function, we can determine a suitable parameter for $\widetilde{b}_{in}$ to find proper service curve parameters.

If $CD1$ it is not satisfied, $CD2$ must hold true. So, we derive the service curve based on our measurements for the negative burst parameter $b_{in_{neg}}$. We insert $b_{in_{neg}}$ as the measured parameter b_{in} into the formula of the optimization function (22) and solve it. We always use the $b_{in_{pos}}$ for the burst parameter of the arrival curve, to derive a valid arrival curve fulfilling the subadditivity property.

4 EVALUATION AND RESULTS

In this chapter, we state three research questions (RQ1-RQ3), explain why the question is important to us, we explain the methodology we use to answer the research question, we state a hypothesis, explain our experimental setup, and discuss the results.

RQ1: How tight are the end-to-end delay and backlog bounds by the estimated arrival and service-curves derived by different estimation methods based on TL from real industrial workload?

There are various methods which can be found in literature for estimating service curves from TL. Our aim is to decrease the TL and to enable a RTM technique during operation. To this end, we have developed the TBASCEM method. However, prior to implementing the RTM in software, it is worth evaluating the tightness of the bounds we can generate with the TL already collected from the HIL system.

Methodology: To evaluate the tightness, we implement the method in MATLAB and apply it offline on HIL TL. We furthermore compare it to other methods from literature, applied on the same dataset.

Hypotheses: Our Hypothesis is that TBASCEM derives the tightest bounds compared to all other applied methods.

Experimental Setup: To assess the performance of various arrival and service curve algorithms, we employ a dataset comprising 81 instances of TL with each, 61116 data points divided into 9 meaningful subserver configurations. This leads to a set of 44 million data points in total, collected from our prototype HIL system. We used realistic industrial workload; emulating radar data send as Ethernet packets over a 100 Mbps channel to the DUT with an average rate of 7.222 Mbps and a maximum rate of 100 Mbps while using Ethernet packets of 1538 byte length. TBASCEM is implemented in MATLAB and is subsequently applied to the recorded timestamps originating from the software processes. We extract both, latency and backlog bound from the arrival curve yielded by a single RTM algorithm, as well as from the service curve generated by four other state-of-the art algorithms, namely Best-Case Execution Time (BCET), Mean-Case Execution Time (MCET), WCET and the approach by Alcuri et al.. Additionally, we conduct a comparison of the bounds' tightness by dividing the NC bound by the maximum measured values of queue length and end-to-end delay. We used software timestamps as a baseline for our performance evaluation of the HIL streaming system. Having in mind that these Software (SW) timestamps are not as precise at Hardware (HW) timestamps. However, as the system processing time works in the ms range and their precision lays in the μs range, they can be presumed as precise enough.

Results:

We evaluate the tightness of the TBASCEM and the other algorithms offline using TL. Subsequently, we compare the computed bounds of all methods as tightness factors by dividing them with the maximum measured delay or backlog. These comparisons are presented in Figure 4 as box plots, which displays the following information: the median, the lower and upper quartiles, any outliers (computed using the interquartile range), and the minimum and maximum values that are not outliers. The box plot or box chart is described by MathWorks™according to [16] as follows. The sample median is the line inside each box. The top and bottom edges of each box are the upper and lower quartiles, respectively, where the upper quartile corresponds to the 0.75 quantile and the lower quartile corresponds to the 0.25 quantile. The distance between the top and bottom edges is the Interquartile Range (IQR). Outliers are values that are more than 1.5 IQR away from the top or bottom of the box. The whiskers are lines that extend above and below each box. A whisker connects the upper quartile to the non outlier maximum and to the non outlier minimum respectively. A tightness factor of smaller than 1 is an undershot and hence rated as a violation of the bound by the maximum measured value. We refer the interested reader to the full description of MathWorks™defined box charts to this source [16]. A tightness factor between 1 and 10 is rated as high tightness. A tightness factor between 10 and 10^3 is rated as medium tightness. And a tightness factor over 10^3 we rate as low tightness. However, it depends on the absolute value at the end if the tightness factor can be used as a safety factor for the playback-buffer size or pre-buffer time. If the measured service delay is in the μs range and the maximum measured backlog is in the byte range, a safety factor of 10k or even 1000k would be still in practice feasible. However, it would lead to inefficient use of computing resources. It is a trade-off between system robustness and resource capabilities. Remarkably, the delay tightness of the TBASCEM algorithm is mostly 1 in our datasets with outliers at 1.1 what was our aim, to have a very tigtht delay bound. Compared to the boxplots of the other algorithms, the TBASCEM bounds are significantly tighter. It is followed by the Alcuri and MCET method. They can't be significantly distinguished, as their boxes overlap. However, the median is significantly different, with 25 for Alcuri and around 60 for the MCET method. The BCET follows on the third place and the WCET method derives the untightest results with 60k in median but seems to have almost no variation, however this mainly depends on the logarithmic scale of Figure 4. While diving into the numbers in Table 1, we see that its whiskers and its outliers range between 53k and 70k and its IQR is around 562. For the backlog tightness, the results look a bit different. The Medium tightness of TBASCEM is with 2 the lowest of all algorithms. However, as the boxes overlay with the Alcuri and MCET method, they are not significantly distinguishable. The median value of the Alcuri algorithm is 10. The Median of the MCET method lays significantly higher at 90. The BCET method lays with its median of 300 significantly higher as the three before mentioned methods, its boxes lay between 100 and 11k also significantly higher and its whiskers range between 11 and 30k what is also significantly higher. Its outliers are near to the upper whisker also at around 28k. The WCET reaches again the significantly untightest median bound with 25k, while its box lays between 14k and 31k and its whiskers range between 500 and 58k without any outliers. However, it highly depends on the system parameters and the absolute value if the bounds derived by the arrival and service curve values can be used in practice. If they cannot be used, the maximum measured end-to-end delay and backlog can be used instead. However, as the TBASCEM algorithm produces tight bounds in our datasets for the delay, we can use the results to design the pre-buffer delay of our system. Even with a backlog tightness up to 10k we also can use the approach to design our buffers, as the Random Access Memory (RAM) resources are in the Gigabyte range while the calculated absolute bounds would be still in the Megabyte range. However, the design in total would be still just medium efficient, but is comparable to the other state-of-the-art methods.

RQ2: How high is the impact on the CPU performance of the eased runtime measurement method for TBASCEM compared to TL?

Explanation: To monitor our HIL streaming system's performance while it is in use, the RTM must reduce the saved variables while utilizing a relatively small amount of additional computational power. Producing less logging data would mean to reduce the measurements and implement an online algorithm for the measurements based on the TBASCEM approach. The logging data would be highly reduced, however, what would that mean for the CPU performance, if that algorithm runs in parallel on the computing machine?

Methodology: Implementation of the runtime measurement algorithm for TBASCEM and run it without logging, with TL and with TBASCEM logging and log the CPU load, compare the measured CPU load in a box plot. If the boxes cover each other and

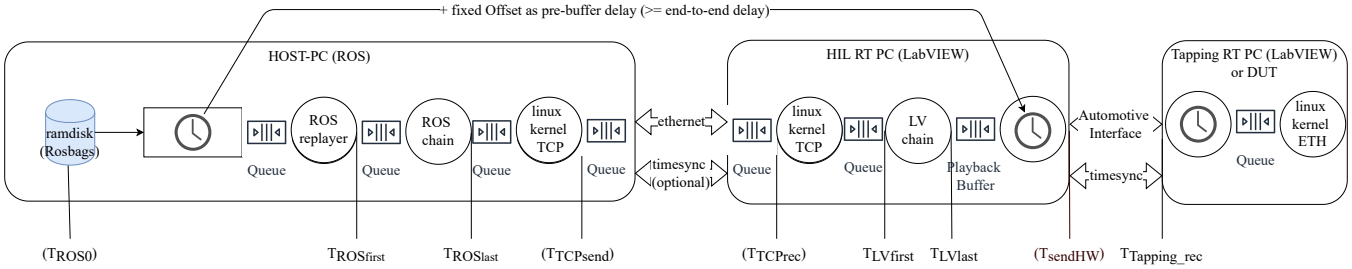


Figure 3: HIL streaming and queuing system with SW instrumentation to collect timestamps

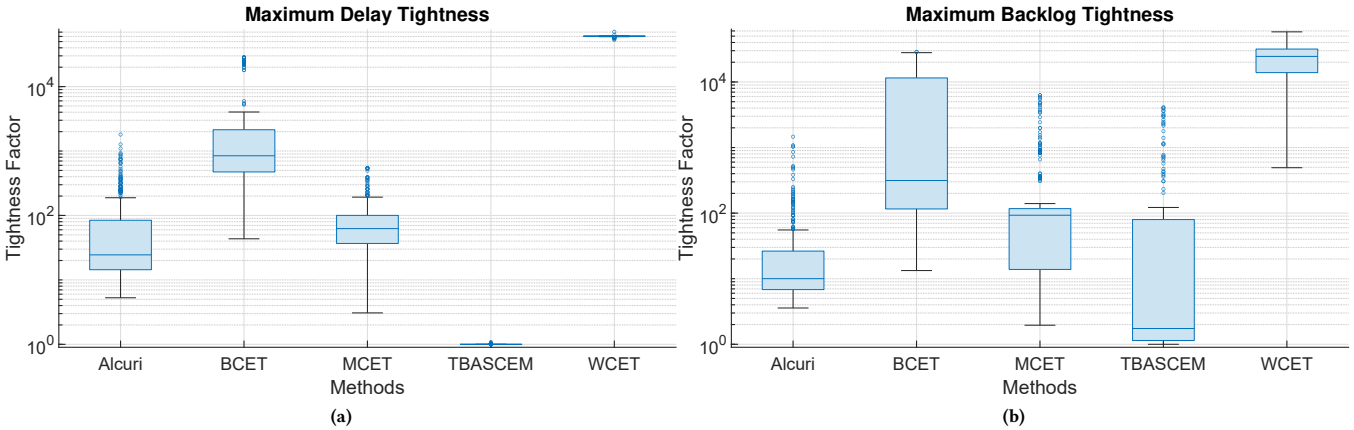


Figure 4: Performance evaluation: Maximum tightness of end-to-end delay bound (a) and backlog bound (b) over different servers and derived with different methods

Table 1: Summary Statistics for Delay Tightness

Method	Alcuri	BCET	MCET	TBASCEM	WCET
Upper Outlier	1802.2	28731.1	547.6	1.1	70817.8
Upper Whisker	188.3	4033.3	192.4	1.0	61265.0
0.75 Quantile	84.3	2139.2	100.3	1.0	60812.2
Median	24.5	844.0	62.5	1.0	60574.6
IQR	69.9	1665.7	63.5	0.0	562.2
0.25 Quantile	14.4	473.5	36.8	1.0	60250.0
Lower Whisker	5.3	43.4	3.1	1.0	59452.5
Lower Outlier	NaN	NaN	NaN	NaN	53263.4

Table 2: Summary Statistics for Backlog Tightness

Method	Alcuri	BCET	MCET	TBASCEM	WCET
Upper Outlier	1464.5	28706.2	6271.6	4112.0	NaN
Upper Whisker	55.4	27984.2	139.8	122.1	58111.9
0.75 Quantile	26.4	11524.5	117.1	79.8	31762.1
Median	10.0	314.3	92.9	1.7	24543.5
IQR	19.6	11408.6	103.4	78.7	17879.9
0.25 Quantile	6.8	115.9	13.8	1.1	13882.2
Lower Whisker	3.6	13.3	2.0	1.0	492.5
Lower Outlier	NaN	NaN	NaN	NaN	NaN

there is no gap between them, they are assumed as not significantly distinguishable.

Hypotheses: We postulated that the CPU performance demand for the run-time algorithm would be significantly lower than that of TL, given our observation that file writing places a substantial burden on CPU resources.

Experimental Setup: In order to validate this hypothesis, we executed a proof-of-concept implementation of the TBASCEM run-time measurement algorithm using LabVIEW. Subsequently, we carried out a comparative analysis, evaluating the LabVIEW-based TBASCEM implementation against TL, as well as a scenario without any logging. The assessment focused on parameters such as CPU utilization. CPU utilization measurements were taken every second using the Linux *top* command. This was chosen to provide granularity while avoiding overloading the system with monitoring measurements. We captured a snapshot of the CPU utilization of all cores as a percentage. We stream Ethernet data via 1,4,8,10,16 streams in parallel while logging via TBASCEM, TL and no logging while logging the CPU load every second via *top* command.

Results: The results of the measurements with different logging configurations, while scaling up the number of streams is shown in Figure 5. We see in the results, that the CPU utilization of TBASCEM, TL, and without logging differ not much per stream. They lay all in a comparable range, and the boxes overlap, so they are not

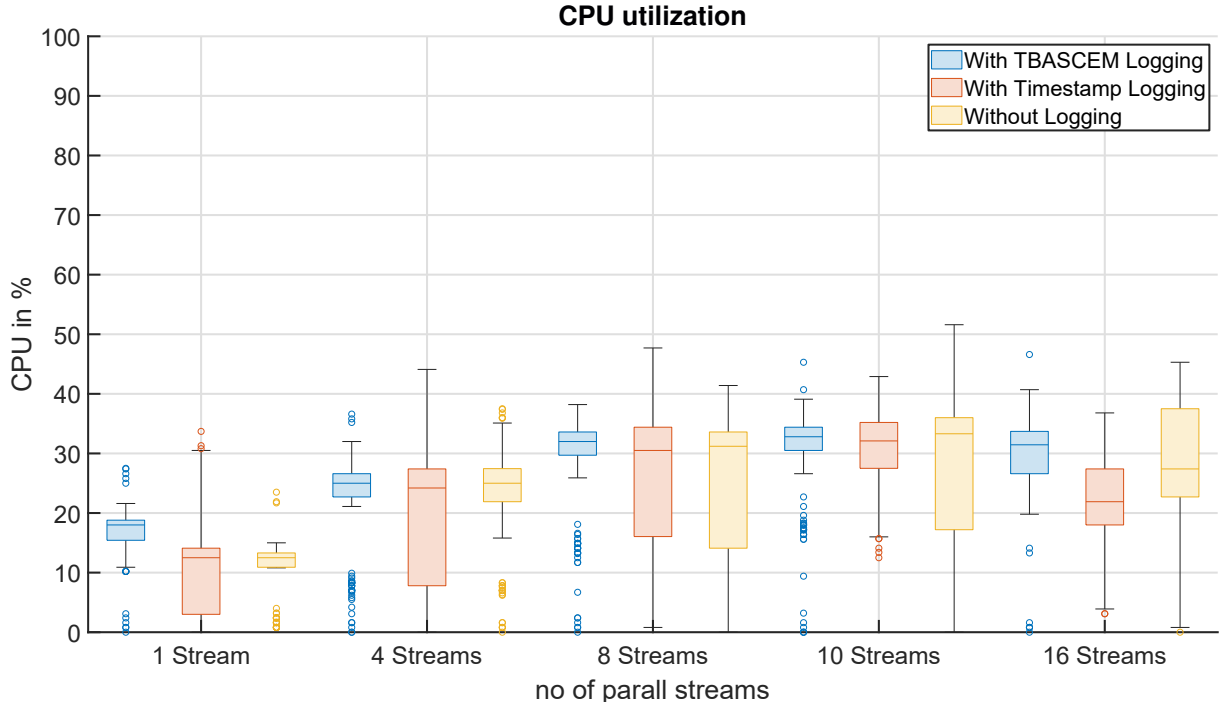


Figure 5: Measurement of CPU utilization of different logging methods

distinguishable. The measurements of 1, 4, 8, and 10 streams show consistent results for the higher peaks in CPU utilization of TL compared to TBASCEM logs. For 16 streams, the TBASCEM logging peak is higher. So, the TL and writing to a file, could indeed need more often high CPU resources. We don't see this always, like in the example of 16 streams, as the granularity of the CPU logs are too low. We could increase that with a higher frequency of *top* logs, but this would also increase the CPU load itself. However, the difference of the CPU load for 1 and 16 streams between the methods with TL and TBASCEM logging are just about 5%. So, the TBASCEM RTM approach may not need less performance, however it is comparable to the TL method and does not need much more CPU performance. Furthermore, the data size would be reduced, since the algorithm merely stores four values instead of the two TL per sample, a difference that becomes pronounced with a high number of samples and testing time. Table 3 displays the file sizes for both a TL at a single point and the TBASCEM log written to a CSV file.

Table 3: File sizes of streams with different duration with different logging methods.

	timestamp log in [kB]	**TBASCEM log in [kB]**
90s stream	1970	1
180s stream	3940	1
360s stream	7880	1
5000h stream	394.000.000	1

Considering the parallel streaming of up to 40 streams and the requirement for at least two points with timestamps to calculate delay and backlog, the overall file size can quickly increase up to several TB for a stream duration of approximately 5000h just for TL. Transferring the data from the HIL system to a different server for post-processing may be necessary, even though memory may not be an issue. This needs additional time, what could be used for the normal operation of the HIL system instead.

RQ3: How performant are the arrival- and service-curve estimation methods implemented as online algorithms compared to state-of-the-art TL?

Explanation: We want to compare the performance needs of the different methods for the memory and CPU, to rate their scalability and their tightness. These are the most relevant requirements for a run-time monitoring method.

Methodology: We assess the computational effort, memory usage, and scalability qualitatively by considering how the method operates and must be implemented as a runtime RTM. We describe the effort and provide a qualitative rating. The tightness of the bounds can be evaluated quantitatively by examining the results of the initial evaluation and comparison. The results have been categorised into qualitative values, with values under 10 considered high, those between 10 and 1000 considered medium, and values above 1000 considered low. This approach ensures consistency with other qualitatively rated factors. Additionally, we outline our requirements for the run-time monitoring approach.

Results: Various algorithms for estimating arrival and service curves using NC have been extensively discussed in the literature in recent years. The subsection offers valuable insights into the different approaches proposed by researchers for estimating arrival

curves and service curves, helping readers understand the diversity and nuances of the existing methods.

Service Curve Performance Estimation: When evaluating the suitability of the measurement method for normal run-time operation of a computer system, our primary focus is on achieving high performance, characterized by low computational effort and minimal memory usage, to ensure excellent scalability. A high level of tightness for the delay and a medium level of tightness for the backlog bound is considered sufficient for our specific objectives. Emphasizing efficiency and the ability to handle larger workloads is essential in practical applications, guiding our approach to ensure the method's practicality and effectiveness in real-world applications. The approach proposed by Alcuri et al. [1] follows an iterative method that analyses backlogged periods first for mean-rate estimation and then for maximum latency estimation. However, due to its high computational effort and memory usage, its scalability is rated as low, making it unsuitable as a run-time measurement algorithm. The WCET algorithm, proposed by Helm et al. [12] and applied to a HIL test system in Funda et al. [9] exhibits good performance but suffers from low tightness, particularly in non-hard real-time streaming systems where the mean service rate is close to the mean input rate. Any short-term WCET that results in a service rate lower than the input rate will create infinite bounds, limiting its practical application. The WCET method displayed in Figure 4 often yields no results due to an infinite bound, which is not illustrated.

The MCET algorithm, also proposed by Helm et al. [12] and detailed in Funda et al. [9] shows medium suitability from a performance perspective, with medium to low tightness. However, it requires saving a significant amount of data as all processing times are accumulated to derive rate and latency parameters.

The BCET algorithm, inspired by Wandeler et al. [20] and applied to a HIL test system by Funda et al. in [9] is unsuitable in terms of performance since it requires storing and sorting all measured processing latencies. Moreover, its tightness is insufficient, as it achieves medium to low tightness of the measured values as displayed in Figure 4. In contrast, the TBASCEM algorithm proposed in this paper has been evaluated to have medium computational requirements and low memory usage and high tightness for the delay and high to medium tightness for the backlog. The performance evaluation and tightness assessment of the TBASCEM algorithm are the primary focus of this paper and have been presented in the preceding chapters.

Arrival Curve Performance Estimation: The algorithm developed by Bouillard [5] is structured in multiple layers to detect more refined periodical behaviours. Initial data points that fall outside the specified bounds are identified as a sub-flow and are used as input for subsequent stages. From its description, we assess its computational effort to be of medium magnitude and its memory usage to be high, resulting in a medium level of scalability. However, this algorithm is not applicable in our context since we do not employ a traffic shaper like a token-bucket, which is a prerequisite for this algorithm. As a result, we cannot provide the required maximum burst parameter in advance; instead, we need to measure it during run-time.

On a different note, the iterative algorithm introduced in [9] lays a strong foundation for run-time implementation. With a foundational understanding of NC and basic measurements, this algorithm can be extended to estimate the minimum service curve.

The direct iterative approach discussed by Funda et al. in [9] estimates both the burst and mean-rate parameters with reduced computational effort. However, it necessitates the continuous storage of data over time, leading to a gradual increase in memory usage, which we rate as excessive. Nevertheless, it attains the highest level of tightness since it aligns with the definition of arrival curves. This algorithm serves as a valuable benchmark for comparison purposes.

In this paper, we operate under the assumption of low-performance requirements and an at least medium level of tightness for the run-time measurement demands. To validate these assumptions, we conducted a comprehensive experimental performance evaluation of the RTM for the reverse engineering TBASCEM algorithm. The TBASCEM algorithm proposed in this paper has been evaluated to have medium computational requirements and low memory usage and medium to high tightness.

5 RELATED WORK

Algorithms for estimating service curves using NC have been a common topic in the literature for the past years [1, 3–5, 8, 9, 12, 15, 20]. In this section, we present some examples of them, which we have chosen due to their applicability for our use-case of a FIFO queuing server system.

The work by Alcuri et al. [1] introduces a method to estimate service curves for various types of systems, including non-First-In-First-Out ones. The algorithm segments input and output traffic measurements into backlogged periods (periods when the buffer is not empty) and iteratively determines the start time, end time, and the amount of output traffic for each backlogged period. Throughput r of each period is computed as the bits leaving the system divided by the duration of the period. A maximum estimation technique finds the maximum throughput r_{max} among all backlogged periods. By tracing a line with a slope of r_{max} at all points of the departure process and projecting it onto the horizontal axis, the delay T is computed. Maximum delay T_{max} of each backlogged period is then obtained. The service curve is represented as a rate-latency curve with rate r_{max} and latency T_{max}[1]. This approach aligns with the definition of strict service curves as mentioned by Le Boudec et al. in [14].

The first method, proposed by Helm et al. [12], utilizes WCET to estimate the service curve. It takes the maximum measured processing latency and the minimum measured rate as the latency and rate for the service curve, respectively. While this approach performs well for hard real-time requirements, it requires that the minimum system service rate to be higher than the mean input rate. For systems involving streams of limited length and playback buffers, uninterrupted streaming is possible even if the mean system rate is not higher than the mean input rate.

To address this, the second method is based on the MCET and estimates the service using the measured mean rate. It computes the system latency by considering the maximum and minimum deviation between the input flow and the mean flow, effectively

accounting for latency outliers. This approach is inspired by Helm et al. [12].

The third method, based on BCET, orders the measured data with cumulative latencies in descending order to derive a service curve. A tangent at the rate-point of interest is traced, and the intersection point with the time-axis determines the system latency. However, this method may lead to highly overestimated bounds, particularly when patterns in the service flow exist. Additionally, it can be employed to estimate the arrival curve by sorting measured data in ascending order by the inter-arrival time. The intersection between the tangent and the y-axis (bytes) determines the burst parameter. This approach is inspired by the sliding window approach mentioned in the real-time calculus (RTC) framework by Wandeler et al. [20]. This method is particularly useful for assessing the feasibility of streaming a time-limited stream.

There are other measurement-based estimation methods for service curves mentioned in a literature review and survey study, that was conducted by Fidler et al. on various service curve models in the NC framework [8]. Especially in the section about "measurement-based service curve estimation" [8], are different methods mentioned, like the one from Alcuri et al. [1]. We have not re-implement all the methods and compared to our method. Especially the method by Undheim et al. [19] seek to estimate the latency and rate parameter during a burst and backlogged period like Alcuri et al. [1]. The main difference is that Undheim et al. use the max-plus algebra instead of the min-plus algebra. It could be, that the max-plus algebra-based estimation derives tighter bounds than the min-plus algebra-based solution used in [1]. This was already observed ones by Xie et al. in [21], while applying NC with min-plus algebra and max-plus algebra on priority scheduling for deriving delay bounds.

6 CONCLUSIONS

This paper introduces the TBASCEM methodology. The method delivers arrival and service curves which produce tight delay bounds and involves an efficient measurement process. It combines a RTM technique and a reverse engineering algorithm to estimate linear arrival and service curve parameters from measured data. However, it can also be applied to state-of-the-art TL from the input flow and output flow of any FIFO server queuing system. We applied the alternative methods on realistic data from the HIL system and received bounds with a tightness factor from several hundred up to several thousands. So, we found a gap for service curve estimation from measurement data, which produces tight bounds.

According to our empirical investigation, the reverse engineering algorithm TBASCEM provides tighter bounds compared to other methods in the literature. It can be applied on either TL of input and output flows of a server queuing system, or on reduced measurements we proposed here in this paper. Furthermore, we found out that the TBASCEM RTM adequately uses similar CPU performance like state-of-the-art TL, while it reduces limitations by data explosion associated with conventional TL, as confirmed by empirical evaluation of CPU utilisation and exemplary calculation of logging data sizes.

The TBASCEM RTM can be used in any streaming system to generate arrival and service curves without significantly impacting performance. It can be employed to evaluate performance, design buffer sizes and pre-buffer time, and continuously monitor performance during operation, to make service outliers visible.

Future work could involve comparing other measurement-based estimation methods for service curves mentioned in [8], that we did not implement and compare to our method. Additionally, our concept estimates the burst parameter. One could argue that it would be better to measure this parameter in a more appropriate manner. It is also critical to note that we assume all maximum parameters occur simultaneously, which is highly probable, but cannot be guaranteed. It may be more beneficial to measure the arrival curve burst parameter and the service curve rate and latency parameter directly online during operation from current values, saving only the maximum latency and minimum rate parameter for future reference. The TBASCEM optimization algorithm could also be adapted to find a solution for a given tightness factor for either the latency or the backlog within the solution space of the functional share of arrival and service curves. Future research could also include to extend the approach to the stochastic NC framework for formal validation of soft real-time streaming systems.

ACKNOWLEDGMENT

We gratefully acknowledge the invaluable contributions of: Lisa Maile (Friedrich-Alexander Universität Erlangen-Nürnberg), Dušan Okanović and Dhruv Jagga for insightful proofreading and feedback. Prof. Christian Berger (University of Gothenburg) for thought-provoking questions. Hampel Software Engineering GmbH for implementing the TBASCEM RTM in LabVIEW. Frank Keck (ZF Mobility Solutions GmbH) for unwavering support. This research was supported by ZF AG.

REFERENCES

[1] Luigi Alcuri, Giuseppe Barbera, and Giuseppe D'Acquisto. 2005. Service Curve Estimation by Measurement: An Input Output Analysis of a Softswitch Model. In *Quality of Service in Multiservice IP Networks*, Marco Ajmone Marsan, Giuseppe Bianchi, Marco Listanti, and Michela Meo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–60.

[2] Bulgaria Aleksandrov, Chavdar Acad, Bulgaria Rumenin, Christian Magele, Stoyanov, Bulgaria Sotirova, Ritchie, Toepfer, Hartmut Brauer, Marin Hristov, Repetto, Bulgaria Antchev, Bulgaria Mihailov, Bulgaria Romansky, Bulgaria Vasilev, Japan Tanaka, Ventsislav Valchev, Vladimir Shelyagin, Ukraine Acad, and Anna Stoynova. 2019. Review of hardware-in-the-loop -a hundred years progress in the pseudo-real testing. 54 (12 2019), 70–84.

[3] A.A. Baybulatov and V.G. Promyslov. 2019. Control System Availability Assessment Via Maximum Delay Calculation. In *2019 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)*. 1–6. https://doi.org/10.1109/ICIEAM.2019.8743012

[4] A.A. Baybulatov and V. G. Promyslov. 2017. A Technique for Envelope Regression in Network Calculus. In *2017 IEEE 11th International Conference on Application of Information and Communication Technologies (AICT)*. 1–4. https://doi.org/10.1109/ICAICT.2017.8687034

[5] Anne Bouillard, Laurent Jouhet, and Eric Thierry. 2009. Service curves in Network Calculus: dos and don'ts.

[6] R.L. Cruz. 1991. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory* 37, 1 (1991), 114–131. https://doi.org/10.1109/18.61109

[7] dSPACE GmbH. 2017. FAQ 242 - Handling Overrun Situations. https://www.dspace.com/shared/support/faqpdf/faq242.pdf

[8] Markus Fidler. 2010. Survey of deterministic and stochastic service curve models in the network calculus. *IEEE Communications Surveys & Tutorials* 12, 1 (2010), 59–86. https://doi.org/10.1109/SURV.2010.020110.00019

[9] Christoph Funda, Pablo Marín García, Reinhard German, and Kai-Steffen Hielscher. 2023. Arrival and Service Curve Measurement-Based Estimation Methods to Analyze and Design Soft Real-Time Streaming Systems with Network Calculus. In *2023 3rd International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. 1–8. https://doi.org/10.1109/ICECCME57830.2023.10253001

[10] Christoph Funda, Kai-Steffen Jens Hielscher, and Reinhard German. 2021. Discrete event simulation for the purpose of real-time performance evaluation of distributed hardware-in-the-loop simulators for autonomous driving vehicle validation. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 80 (2021).

[11] Christoph Funda, Tobias Konheiser, Thomas Herpel, Reinhard German, and Kai-Steffen Hielscher. 2022. An industrial case study for performance evaluation of hardware-in-the-loop simulators with a combination of network calculus and discrete-event simulation. In *2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. 1–7. https://doi.org/10.1109/ICECCME55909.2022.9988051

[12] Max Helm, Henning Stubbe, Dominik Scholz, Benedikt Jaeger, Sebastian Gallenmüller, Nemanja Deric, Endri Goshi, Hasanin Harkous, Zikai Zhou, Wolfgang Kellerer, and Georg Carle. 2021. Application of Network Calculus Models on Programmable Device Behavior. In *2021 33rd International Teletraffic Congress (ITC-33)*. Avignon, France, 1–9. https://gitlab2.informatik.uni-wuerzburg.de/itc-conference/itc-conference-public/-/raw/master/itc33/hel21ITC33.pdf?inline=true

[13] Wolfgang Kellerer and Amaury Van Bemten. 2016. *Network Calculus: A Comprehensive Guide*. Technical Report 201603. Technische Universität München Lehrstuhl für Kommunikationsnetze, Arcisstr. 21, 80333 München, German.

[14] Jean-Yves Le Boudec and Patrick Thiran (Eds.). 2001. *Network Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–81. https://doi.org/10.1007/3-540-45318-0_1

[15] Ralf Lübben and Markus Fidler. 2017. Service Curve Estimation-Based Characterization and Evaluation of Closed-Loop Flow Control. *IEEE Transactions on Network and Service Management* 14, 1 (2017), 161–175. https://doi.org/10.1109/TNSM.2016.2638471

[16] MATHWORKS. [n. d.]. Box chart (box plot) - MATLAB boxchart - MathWorks Deutschland — de.mathworks.com. https://de.mathworks.com/help/matlab/ref/boxchart.html. [Accessed 22-02-2024].

[17] Franc Mihalič, Mitja Truntič, and Alenka Hren. 2022. Hardware-in-the-loop simulations: A historical overview of engineering challenges. *Electronics* 11, 15 (2022), 2462.

[18] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

[19] Astrid Undheim, Yuming Jiang, and Peder J. Emstad. 2007. Network Calculus Approach to Router Modeling with External Measurements. *2007 Second International Conference on Communications and Networking in China* (2007), 276–280. https://api.semanticscholar.org/CorpusID:6662110

[20] Ernesto Wandeler. 2006. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. Ph. D. Dissertation. ETH Zurich.

[21] Jing Xie and Min Xie. 2013. Delay bound analysis in real-time networks with priority scheduling using network calculus. In *2013 IEEE International Conference on Communications (ICC)*. IEEE, 2469–2474.

A APPENDIX

A.1 Technical Information about the HIL Systems Under Study

The HIL cluster consists of a HOST PC and a HIL RT-PXI.

A.1.1 HOST PC. The HOST PC is equipped with a 2012 Intel Xeon E3-1230 V2 3.3 GHz processor (four physical CPU cores) and 16 GB system memory. All worker nodes are connected via 40 Gbit Ethernet in a single-switch star topology. Each node runs Gentoo Linux (kernel version 3.6.11) and Java 1.7.0.13.

A.1.2 HIL PC. The HIL RT-PXIe-8880 is equipped with an Intel(R) Xeon(R) CPU E5-2618L v3 @ 2.30GHz (8 physical CPU cores) and 24 GB system memory. All worker nodes are connected via 40 Gbps Ethernet in a single-switch star topology. Each node runs NI Linux Real-Time x64 4.14.146-rt67 and other NI LabVIEW Runtime and NI Drivers.

GLOSSARY

BCET Best-Case Execution Time (BCET) is the lowest observed execution time of a software process running on a dedicated computing machine.. 7, 10, 11

DUT Device Under Test (DUT) is the technical device, what is integrated into the hardware-in-the-loop simulator and stimulated by measurement data from the real-world device or by simulation.. 1–4, 7

FIFO First-In, First-Out (FIFO) refers to a principle where the first item to enter a system or queue is the first to be processed or served.. 1, 10, 11

HIL Hardware-in-the-Loop (HIL) test system or simulator or test bench is a methodology and a technical system for testing and validation of a technical product. See [2, 17] for details.. 1–12

HW Hardware (HW) are machines, wiring, and other physical components of a computer or other electronic system.. 7

IQR interquartile range (IQR) refers to the distance between the top and bottom edges of a boxplot, corresponding to the upper quartile or 0.75 quantile and the lower quartile or 0.25 quantile, respectively. 7

MCET The Mean-Case Execution Time (MCET) is the mean or average observed execution time of a software process running on a dedicated computing machine.. 7, 10

NC Network Calculus (NC) is a system theoretical approach for calculating delay and backlog bounds by min-plus algebra.. 1–5, 7, 9–11

RAM Random Access Memory (RAM) is a type of computer memory that stores data that can be searched by programs.. 7

RTM Run-Time Measurement (RTM) approach are measurement methods for performance evaluation of software during system operation. In this paper mainly used to measure the maximum end-to-end delay and backlog between the input and output flow of a message stream in a software service process with queues in between. Additionally, the maximum burst parameter of the input and output flow is estimated.. 2, 6, 7, 9–11

SW Software (SW) are programs and other operating information used by a computer.. 7

TL Timestamp Logging (TL) is a state-of-the-art method for performance evaluation of software timing. Timestamp Logs (TL) are the respective data-files generated by the logging. In this paper mainly used to measure the input and output flow of a message stream in a software service process with queuing.. 1–3, 6–9, 11

WCET Worst-Case Execution Time (WCET) is the highest observed execution time of a software process running on a dedicated computing machine.. 3, 7, 10

Received 17 November 2023; Accepted 29 December 2023; Revised 11 March 2024

A Learning-Based Caching Mechanism for Edge Content Delivery

Hoda Torabi
York University
Department of Electrical Engineering & Computer Science
Toronto, Canada
hodat@yorku.ca

Hamzeh Khazaei
York University
Department of Electrical Engineering & Computer Science
Toronto, Canada
hkh@yorku.ca

Marin Litoiu
York University
Department of Electrical Engineering & Computer Science
Toronto, Canada
mlitoiu@yorku.ca

ABSTRACT

With the advent of 5G networks and the rise of the Internet of Things (IoT), Content Delivery Networks (CDNs) are increasingly extending into the network edge. This shift introduces unique challenges, particularly due to the limited cache storage and the diverse request patterns at the edge. These edge environments can host traffic classes characterized by varied object-size distributions and object-access patterns. Such complexity makes it difficult for traditional caching strategies, which often rely on metrics like request frequency or time intervals, to be effective. Despite these complexities, the optimization of edge caching is crucial. Improved byte hit rates at the edge not only alleviate the load on the network backbone but also minimize operational costs and expedite content delivery to end-users. In this paper, we introduce HR-Cache, a comprehensive learning-based caching framework grounded in the principles of Hazard Rate (HR) ordering, a rule originally formulated to compute an upper bound on cache performance. HR-Cache leverages this rule to guide future object eviction decisions. It employs a lightweight machine learning model to learn from caching decisions made based on HR ordering, subsequently predicting the "cache-friendliness" of incoming requests. Objects deemed "cache-averse" are placed into cache as priority candidates for eviction. Through extensive experimentation, we demonstrate that HR-Cache not only consistently enhances byte hit rates compared to existing state-of-the-art methods but also achieves this with minimal prediction overhead. Our experimental results, using three real-world traces and one synthetic trace, indicate that HR-Cache consistently achieves 2.2–14.6% greater WAN traffic savings than LRU. It outperforms not only heuristic caching strategies but also the state-of-the-art learning-based algorithm.

CCS CONCEPTS

• **Theory of computation** → **Caching and paging algorithms**; • **Computing methodologies** → **Machine learning**; • **Networks** → **Network services**; • **General and reference** → *Performance*.

KEYWORDS

caching, content delivery network, hazard rate, hit probability

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645037

ACM Reference Format:
Hoda Torabi, Hamzeh Khazaei, and Marin Litoiu. 2024. A Learning-Based Caching Mechanism for Edge Content Delivery. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645037

1 INTRODUCTION

The increasing use of multimedia content services and the expansive deployment of IoT systems necessitate fast and dependable content delivery. Essential online services, such as web hosting and video streaming, rely heavily on the efficiency of these deliveries. With the advancement of 5G networks, edge caching has become a pivotal technology for addressing these rising demands and improving overall performance. By storing content closer to users at the network edge, edge caching plays an essential role in enhancing user experience and reducing the bandwidth required across the wide-area network between edge nodes and the original content servers.

This surge in demand for content delivery directly leads to a substantial and growing volume of traffic. This increase has a significant financial impact on network service providers, particularly if traffic is not managed effectively at the edge. While low latency is critical for delivering small-size, latency-sensitive content, the primary objectives for distributing larger files, such as video streams and extensive downloads, are to minimize traffic handling costs and prevent overload at network bottlenecks [20]. Therefore, a key goal of caching at the edge of network is to maximize the fraction of bytes served locally from the cache [11], also known as the byte hit ratio (BHR).

The caching algorithm, which decides which objects are cached, is integral to achieving a low byte miss ratio. As such, this problem has been extensively studied since the advent of the internet. Caching strategies have evolved from basic heuristic methods like Least Recently Used (LRU), which evicts the oldest data first, to intricate algorithms that combine frequency and recency (e.g. Hyperbolic) and others that use a composition of frequency and object size (e.g. GDSF). Despite extensive research, most production systems—such as those employed by Akamai [23], Memcached, and NGINX—commonly implement LRU variants as their standard caching algorithm. Yet, these may not be ideally suited to the particular demands of edge caching, which contends with limited cache sizes and the unpredictable nature of user requests [17]. The challenge in designing effective caching algorithms is that workload characteristics, like object access patterns or request processes, are not constant and often change over time. Thus, a heuristic that performs well for one workload scenario may falter in another, or fail

to adapt when access patterns evolve, underscoring the necessity for flexible caching strategies that can overcome these challenges.

Recent advancements in machine learning (ML) have opened new avenues for enhancing cache algorithms, particularly in the face of the aforementioned challenges. One increasingly popular method is to employ ML techniques to forecast the popularity of objects for proactive caching [32]. Yet, this method can lead to suboptimal cache performance [8] under some workloads and proactive downloads consume additional bandwidth which is counter-intuitive to our optimization goal. Another popular approach employs model-free reinforcement learning (RL), where the system starts with no preconceived notions about the traffic patterns and iteratively works its way toward the optimal caching strategy [16, 30, 40, 42]. In this system, decision-making is honed through direct interaction with the environment, utilizing a reward function to reinforce actions that yield positive outcomes. Due to the large state-action space, RL methods tend to be more complex and require greater computational resources. Additionally, they can be sensitive to hyper-parameters, making it challenging to fine-tune their performance.

A promising approach for designing cache algorithms has been to leverage oracle policies, such as the offline Belady algorithm [3], the practical flow-based offline optimal (PFOO) algorithm [5], and the more recent Hazard Rate (HR) based upper bound [26], which compute the theoretically optimal cache decisions. This is achieved by either learning to "imitate" the optimal decisions [4, 18] or by directly learning and predicting the objects' next request arrival to inform the optimal caching choice as explored by [34]. In this paper, we introduce HR-Cache framework, a new learning-based caching framework grounded in the principles of Hazard Rate Ordering (HRO) rule introduced in [1]. HR-Cache is based on several original contributions. The framework is divided into two main components: The first component calculates the caching decisions for a window of past requests based on the HRO rule. The second part then trains an ML model that maps a set of features to HRO cache decisions. This model is then applied to in the next window to predict the "cache-friendliness" of objects at the time they are requested. In the event of a cache miss, where eviction is necessary to make space, our framework preferentially evicts items in the cache that were previously identified as "cache-averse."

Our application of the HRO rule presents an intricate challenge: accurately determining the hazard rate function for object inter-request times to reconstruct the HRO, without making simplifying assumptions about the nature of the request distribution. We address this by employing a Kernel Hazard Estimator, which estimates the hazard function directly from the data without assuming a specific parametric form for the distribution. This consideration can be particularly important, as our use of ML methods is intended to address the shortcomings of heuristic-based algorithms, which usually excel with specific access patterns but not others. Therefore, making assumptions about the workload might negate the advantages that machine learning brings to our caching decision process. Putting all this together in a practical system, however, requires us to address other challenges including controlling the computational overhead for ML training and prediction. Our evaluation results using production and synthetic traces show that our learning-based policy consistently performs better than state-of-the-art methods and reduces WAN traffic by 4–25% compared to the LRU replacement policy and reduces the prediction overhead by a factor of 19.2x compared to the state-of-the-art learning-based cache policy.

Roadmap: We organize the rest of this paper as follows. Section 2 provides background and discusses related work. Section 3 presents the HR-Cache algorithm. Section 4 empirically evaluates the proposed scheme. Section 5 concludes the paper with a summary of its contributions.

2 BACKGROUND & RELATED WORK

Most earlier designs of caching rely on heuristic-based methods including the least recently used (LRU), least frequently used (LFU), and first in first out (FIFO), along with their variants. While these classical methods offer straightforward solutions for managing cache resources, they often fall short in adapting to the dynamic and complex nature of request patterns. Moreover, different features may have varying levels of importance across diverse workloads, a nuance that heuristic methods struggle to accommodate. This limitation is particularly evident in edge networking environments, where traditional traffic assumptions may no longer be valid [27, 35]. In addition, prior work [4, 34] have pointed out a considerable discrepancy between current state-of-the-art caching designs and theoretical upper limits on cache performance, as established by algorithms like Belady's algorithm [3], flow-based offline optimal [5], and the hazard rate upper bound [26]. This significant gap, along with the supporting evidence from recent measurement studies in edge caching systems [10, 13, 33], signals a clear opportunity for enhancements in cache performance and addressing the limitations of existing caching strategies. In light of these developments, there has been a growing emphasis in recent research on developing learning techniques that can intelligently manage cache resources. In the following part, we will focus on a discussion of learning-based cache policies, providing the necessary background and rationale for the development of the proposed HR-Cache framework. Simultaneously, this portion will serve as a review of related work in this area. We'll finish this section with a brief overview of use of ML methods to improve performance, bringing together the key aspects of our research discussion.

2.1 Learning-based Caching

Approaches to learning-based caching can be roughly grouped into three categories, with the first category encompassing recent research efforts focused on leveraging theoretically optimal caching policies to develop learning-based methods. A significant point of reference here is the Belady optimal policy [3]. This algorithm operates on the principle of evicting the object that will be used furthest in the future, thereby minimizing miss rate. While Belady's algorithm provides an ideal strategy for cache replacement, its real-world application has been limited because it requires foreknowledge of future cache access patterns, which is generally not feasible. Nevertheless, this algorithm forms a basis for designing practical caching policies.

Hawkeye [14] was the first to introduce learning from the Belady's algorithm. Hawkeye employs a binary classification model to determine whether a cache line is likely to be reused (deemed "cache-friendly") or not ("cache-averse"). Their policy prioritizes

the eviction of cache-averse lines over cache-friendly ones. By using oracle labels for previous access patterns, Hawkeye effectively transforms cache replacement into a supervised learning challenge. Building upon Hawkeye's foundation, Glider [31] enhances this approach by integrating deep learning techniques to develop a more accurate predictor than its predecessor. However, it's important to note that both Hawkeye and Glider focus on hardware caches and are not directly applicable to software cache systems, particularly those handling variable-sized objects. Another work, "Parrot" as described in [18] adopts an imitation learning approach to automatically learn cache access patterns by leveraging Belady's. Although effective, its computational demands can be significantly high.

Diverging from Parrot's methodology, LRB, as outlined in [34], employs a different strategy by predicting the next arrival times of object requests. This enables LRB to approximate Belady's algorithm through a supervised learning method. By learning the next access time for each object based on a multitude of features, LRB identifies and evicts objects predicted to have the furthest request time. This strategy has demonstrated enhanced performance over state-of-the-art caching algorithms in terms of byte hit ratios. However, LRB is not without its limitations. To closely emulate the optimal offline oracle, a system like LRB is required to predict the next access times for all objects in the cache, selecting for eviction the one with the most distant future request. This prediction process can be extremely resource-intensive for large caches. LRB mitigates this by limiting the inference to a sample of 64 objects for each eviction. Despite this optimization, the prediction overhead remains a significant computational burden. LRB's use of dynamic features means that prediction results are not reusable over time, necessitating fresh sampling and inference for every eviction. Reflecting this overhead, LRB's simulation shows that on a single CPU core, each eviction in LRB consumes 227.19 µs.[1] Consequently, this caps the eviction rate at a maximum of approximately 4,500 objects per second per core, rendering it less efficient for high-demand production environments.

LFO [4], another work employing supervised learning, first calculates the sequence of optimal caching decisions (OPT) for recent history using a min-cost flow model from [5], designed for optimal caching of variable-sized objects. Following this calculation, LFO applies manually-designed features and a gradient boosting decision tree to train a binary classifier for caching decisions. The classifier's prediction is then used to imitate the admission policy of OPT and serve as a ranking metric for the eviction policy. However, the process of deriving optimal decisions based on the min-cost flow model is complex and computationally intensive, hindering LFO's ability to swiftly adapt to workload changes. Additionally, its design necessitates executing a prediction for every incoming request, further impacting its practical efficiency.

Inspired by similar principles to our work, LHR in [41] draws on the concept of the Hazard Rate bound from [26] to develop a learning-based caching policy. Unlike a direct adoption, LHR modifies this approach by constructing an online upper bound, which approximates the request process through a Poisson process. Under this assumption, the hazard rate remains constant and is equivalent to the request rate for each object. While this approach simplifies their model, it considerably narrows the applicability of LHR [12]; particularly in light of [26]'s demonstration that the HRO upper bound is effective for any stationary arrival process. Thus, LHR's reliance on the Poisson assumption potentially restricts the full exploitation of HRO's capabilities.

Considering the insights gained from the review of existing works, our framework's design will be informed around these pivotal lessons:

(1) **Utilization of HRO Bound:** Taking into account the limitations of LHR's Poisson assumption, our approach will seek to fully leverage the HRO bound's potential, avoiding oversimplified assumptions that could undermine the practicality and justification of using machine learning.
(2) **Minimizing Prediction Overhead:** Addressing the challenge of high computational demands seen in methods like LRB, our framework will prioritize efficient prediction mechanisms to enhance scalability and performance.
(3) **Decision-Making Process:** Considering the complexity of the LFO approach, we aim to create an efficient method for making caching decisions. This is important in fast-paced environments where models need regular updates and training. Our approach is designed for quick adjustments to stay up-to-date with frequent changes.

2.2 Use of Machine Learning to Improve Performance

Our work is part of the growing effort to use machine learning to improve system performance, especially in caching strategies. This effort extends across various fields, as evidenced by related literature showcasing the application of machine learning (ML) and deep learning (DL) in enhancing system performance and efficiency in different domains. For instance, the work in [36] employs deep neural networks (DNNs) for optimizing resource management in edge computing environments, enabling dynamic scheduling in distributed fog systems by estimating key Quality of Service (QoS) metrics. The work of [9] utilizes recurrent neural networks to develop performance models for queuing networks, aiming to improve resource utilization based on it. In the realm of database efficiency, the work in [19] leverages tree convolutional neural networks and reinforcement learning to optimize queries , while the work in [22] applies machine learning techniques for effective database indexing. Collectively, these studies underscore the potential of ML and DL as powerful tools for system performance optimization.

3 HR-CACHE

In this section, we begin by discussing the hazard rate upper bound introduced in [26], which forms the cornerstone of our approach. Building upon this foundation, we then introduce our learning-based caching policy, HR-Cache. The primary goal of HR-Cache is to assess whether a requested object is cache-friendly or cache-averse. Upon a cache miss, the requested object is inserted into the cache; however, objects identified as cache-averse are placed in a candidate queue for potential future eviction. HR-Cache gives priority to evicting objects from this candidate queue, resorting to the main queue only when the candidate queue becomes empty.

[1] For 64 GB cache size, Wikipedia 2019 workload

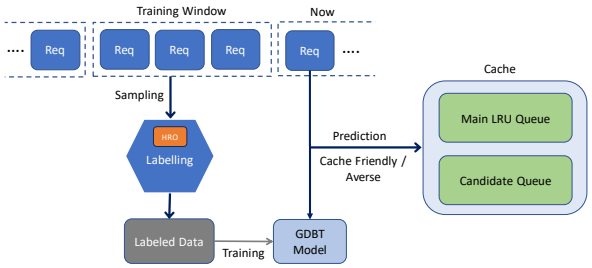


Figure 1: Architecture Overview of HR-Cache

To learn how to make this decision, our method reconstructs the hazard rate ordering solution from [25] for past requests within a window to provide training targets for our model. The output from the model is then used in our caching policy.

Figure 1 provides a high-level overview of the HR-Cache architecture.

3.1 Hazard Rate-Based Upper Bound

The framework in [26] considers a caching system serving n distinct objects, possibly of different sizes. The system is defined with a cache capacity of B bytes, where B is a predefined parameter representing the total storage capacity of the cache. In the basic case, the model assumes a cache of size B bytes addressing requests for n distinct objects of equal size. In this context, [26] introduces the hazard rate based rule, termed as HR-E, which operates as follows:

- At any given time t, HR-E first determines the hazard rate function for each object.
- Then it places in the cache the B objects which have the largest hazard rates (ties between equal rates are broken randomly) .
- A request at time t is considered a "hit" if the requested object is among those cached based on the aforementioned criteria.

They use this rule as a way to upper-bound various cache performance metrics including object hit and byte hit ratio.

They further extend this rule to obtain an upper bound on the byte hit probability for variable size objects. In this case, the authors adapt the hazard rate-based rule denoted as HR-FC to accommodate fractional caching, a strategy that permits the storage of a fraction of an object. Specifically, the rule at any time caches objects with the highest hazard rates until an object cannot fit. For the object that cannot be fully fit due to limited remaining cache capacity, only a sufficient number of bytes required to reach the cache limit are stored. In the case for equal-sized objects, the HR-E rule serves as an upper bound on the cache hit probability for non-anticipative caching policies, while HR-FC serves as an upper bound on the cache byte hit probability, which is the metric we are interested in. Throughout this work, we will collectively refer to these rules as the "HRO" rule (hazard rate ordering rule) for consistency and ease of reference.

3.2 Hazard Rate Function

Let us consider the sequential times at which object i is accessed as $\{\tau_{ik} \mid k \in \mathbb{Z}\}$. The time interval between consecutive requests—namely, the kth and $(k-1)$th requests—for the same object i is termed X_{ik} and computed as $\tau_{ik} - \tau_{i(k-1)}$, for $k \geq 1$. By default, τ_{i0} is set to zero. The sequence $\{X_{ik}\}_{k\geq 1}$ is assumed to form a stationary point process, with the cumulative distribution function (c.d.f) for the inter-arrival time given as $F_i(t) = \mathrm{P}(X_{ik} \leq t)$, and its corresponding density function is represented as $f_i(t)$.

The hazard rate function, denoted as $\lambda_i(t)$, associated with $F_i(t)$ is defined as follows:

$$\lambda_i(t) = \frac{f_i(t)}{1 - F_i(t)}, \quad t \in [0, F_i^{-1}(1)], \tag{1}$$

Here, the hazard rate function is the conditional density of the occurrence of an object request, given the realization of the request process over $[0, t)$ [7]. It is noteworthy that the hazard rate function's meaning can vary based on its application context. For example, in survival analysis, the hazard rate quantifies the conditional probability of an item's failure, given that it has remained functional up to a specific time point. In caching terminology, failure of an object can be treated as the object being requested.

3.3 Calculating Hazard Rate

To effectively implement hazard rate-based rule in our framework, we must first accurately determine the hazard rate function for each object. While this is relatively straightforward for synthetic data sets, it poses a significant challenge in real-world production settings. One approach to this challenge is approximating the inter-request times of objects using well-defined distributions, such as Poisson [41] or Generalized Pareto [26]. However, relying solely on these approximations could potentially diminish the benefits of leveraging machine learning in cache decision-making since these approximations may not be universally applicable across varying workloads and use-cases. Therefore, to calculate hazard rates that are adaptable to various workload trace distributions, we use the kernel hazard estimator proposed by [21]. We obtain this estimator by applying smoothing to the increments of the Nelson-Aalen estimator.

The Nelson-Aalen estimator is a non-parametric method used to estimate the cumulative hazard function in survival analysis. Unlike parametric methods, which make specific assumptions about the underlying hazard rate distribution, the Nelson-Aalen estimator does not require any such assumptions. We denote $H(t)$ as the cumulative hazard function at time t. The estimator is given by:

$$H(t) = \sum_{j:t_j \leq t} \frac{d_j}{n_j}$$

where t_j are the observed event times, d_j is the number of events at time t_j, and n_j is the number of subjects at risk just before time t_j. However, the Nelson-Aalen estimator results in a step function, which is not differentiable. Instead, kernel smoothing techniques are utilized to smooth the increments of the cumulative function estimate obtained by the Nelson-Aalen estimator [39]. The kernel hazard estimator we use takes the form:

$$\lambda(t) = \frac{1}{h}\sum_{i=1}^{n} K\left(\frac{t-t_i}{h}\right)\Delta H(t_i)$$

where $K(\cdot)$ is a kernel function (e.g., Epanechnikov kernel), h is the bandwidth, determining the width of the smoothing window, and $\Delta H(t_i)$ is the increment in the Nelson-Aalen estimate at time t_i, which is $\frac{d_i}{n_i}$.

Trace length	3.7 million
Unique objects	5638

Table 1: IBM Web Access Trace Collected from a Gateway Router

We use a real-world IBM trace from [26], to test the validity of the non-parametric hazard estimator. Details about the trace are provided in Table 1. For our experiment, we derive the upper bound on hit probability using the HR-E ordering rule with three different estimators. The first method employs the non-parametric estimator introduced earlier in this section. The study by [26] effectively estimated the hazard rate for each object in the IBM trace, assuming a Generalized Pareto distribution for inter-request times. We include the HR-E upper bound calculated under their estimator for validation. Additionally, we explore the HRE upper bound assuming request processes follow a Poisson process. For further comparison, we also present the object hit probabilities attained by the LRU and Belady's algorithms. The results of these comparisons, run under 3 different cache sizes, are illustrated in Figure 2.

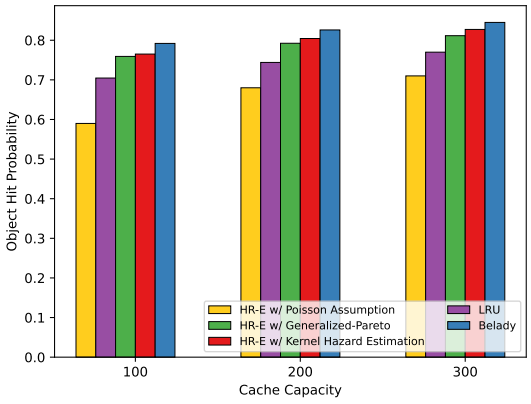


Figure 2: HR-E Upper Bound Comparisons and Hit Probabilities for LRU and Belady's Algorithms Across Three Cache Sizes.

As can be seen, the kernel hazard estimation method we use gives us an upper bound that aligns with the expected bound derived using the "good" parametric estimator of the Generalized Pareto distribution. This confirms that kernel hazard estimation is indeed suitable for our use case. As anticipated, the simplistic nature of the Poisson assumption results in LRU outperforming it. Moreover, our results reaffirm the tighter upper bound achieved by the HR-E rule compared to the Belady algorithm, consistent with the findings of the work in [26].

3.4 Learning From HRO

Before diving into the learning process, it is crucial to make a key observation. We argue that since the HRO bound (Section 3.1) is derived in a pre-fetching manner, it does not directly correspond to cache decision at the time of request to an object. Specifically, the HRO rule assumes that at any time t, the objects with the highest hazard rates among all available objects, have already been pre-fetched and are present in the cache. Thus, the requested object at time t is considered a hit if it is among objects in the cache. Based on this, for object i to be considered as cached in the system, the previous request to object i must admit it to the cache. Or in other words, when the request at time t arrives, it can only be considered a hit if object i was already cached due to a prior request. We classify these earlier requests as cache-friendly, as they are the ones leading to hits. With this and the HRO rule as a backdrop, we are set to develop a learning-based caching strategy. Our approach employs a sliding window of past requests $W[k]$. Using the gathered requests in $W[k]$, we do three things:

(1) First, using the inter-request times of objects in the window we calculate the increments of hazard rates for each object according to the Nelson-Aalen estimator, to be later smoothed by the kernel hazard estimator.
(2) Second, we go over the requests in the window and mark them as hit/miss based on the HRO rule. Meaning for each request at time t, we compute the hazard rate at time t for every object within the window using the kernel estimator, and consider objects with the highest hazard rates in cache until one doesn't fit. If the object requested at time t is among the cached objects, it is considered as a hit; otherwise, it is considered as a miss.
(3) Next, we examine the requests in the window once more: For each request i that was marked as a hit in the first step, we mark the previous request to i as cache-friendly. This provides us with a vector of cache decisions for requests in the window, which serves as the label data for our machine learning model training.

HR-Cache then trains a model that maps features to the decision derived in step 3. The trained model is subsequently used over the next window, $W[k+1]$, to inform cache decisions during which HR-Cache again records the requests to use for the next window and so on.

3.5 Training Data

An important design issue involves determining the optimal amount of past information to utilize. We adopt a sliding window approach, using the data within this window for hazard estimation, deriving the HRO cache decision, and model training. The choice of window size significantly impacts the system's effectiveness. A small window might result in few data for training or hazard rate estimations, while a window that is too large could lead to increased memory usage, as well as longer processing and training times. While some studies arbitrarily define their window sizes (e.g., [4] opts for a window of 1 million, [16] for the initial 10 million requests), [14] considers window size as a factor of cache capacity, 1× represents a window that consists of accesses to k cache lines, where k is the capacity of the cache. We choose a 3× window, meaning the unique

bytes of object requests in the window is three times the cache size as we find that this works well across all our experiments, however there is room for investigating how to set an optimal window size. In practice, the sliding window can encompass millions of objects, which presents significant challenges for the labeling process, particularly when reconstructing the HRO-Rule. To address this issue, HR-Cache employs a strategy of randomly sampling objects within the window to generate training samples. The sampling rate is automatically calibrated to ensure that the total number of operations stays within a manageable range, thus preventing the computational overhead from becoming prohibitive. In our C++ implementation, this adaptive approach has proven to be effective, yielding favorable results while keeping the computational demands at a reasonable level.

3.6 Features

When designing a machine learning model for cache decision, it is essential to choose relevant features that can help predict the optimal decision. Our chosen features encompass both the insights from past heuristics and the insight of recent learning-based caching policies. Traditional caching heuristics focus on individual metrics, such as object recency (as seen in LRU), its frequency (as in LFU), or object size. This is while learning-based methods allow us to incorporates a range of them. We consider the following features which can be derived in an online and robust manner.

(1) Delta series: The time differences between consecutive requests for an object. Δ_1 indicates the amount of time since an object was last requested. Δ_2 indicates the time in between an object's previous two requests and so on, i.e., Δ_n is the amount of time between an object's n^{th} and $(n-1)^{\text{th}}$ previous requests. This can provide insights into the object's access pattern, which can help predict future requests. We use 32 deltas as our features.
(2) Decayed frequency: Unlike simple frequency, decayed frequency accounts for the recency of requests by giving more weight to recent accesses. It calculates the fraction of requests for an object among all requests so far, but with a diminishing emphasis on older requests. This approach helps in capturing not just how often an object is requested, but also how its popularity or relevance changes over time.
(3) Static features. These include unchanging characteristics of an object, such as its size and type. Static features can be useful due to their inherent correlation with different access patterns. For our implementation we only consider size among static features due to the availability of data in our traces.

3.7 Training HR-Cache

The goal of HR-Cache is to map its features to a decision of whether an incoming line is cache-friendly or cache-averse according to the HRO rule. For this task, we employ gradient boosting decision tree (GBDT) model. GBDTs are known for their strong performance across various datasets, particularly useful in tasks involving classification and regression with structured data. They are also convenient because they don't need feature normalization. Additionally, their effectiveness in caching-specific tasks is supported by studies like [34] and [4].

Algorithm 1 HR-Cache Policy

```
procedure UpdateCache(object, lookupTable)
    Perform lookup for object in lookupTable
    if object is in cache (Hit) then
        if object is in Candidate Queue then
            if predicted as Cache-friendly then
                Change mode to Main Queue
                Move object from Candidate to Main Queue
            end if
        else if object is in Main Queue then
            if predicted as Cache-friendly then
                Promote object to MRU in Main Queue
            else if predicted as Cache-averse then
                Change mode to Candidate Queue
                Move object to Candidate Queue
            end if
        end if
    else                                  ▷ Request not in cache (Miss)
        if predicted as Cache-friendly then
            Add object to Main Queue
        else
            Add object to Candidate Queue
        end if
    end if
end procedure
```

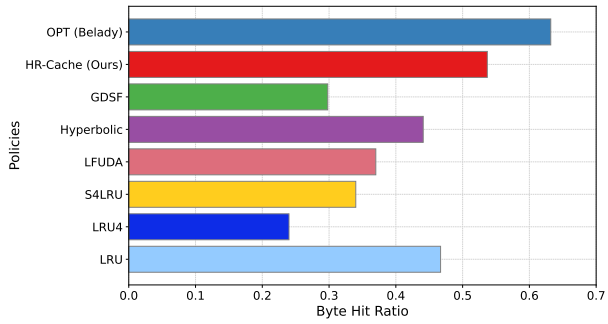


Figure 3: Comparison of HR-Cache to State-of-the-Art Heuristic Caching Systems for the IBM trace.

3.8 The HR-Cache Policy

Putting it all together, we design a caching policy guided by our learned model. For every object request, our HR-Cache predictor outputs a decision indicating whether the object is cache-friendly or cache-averse. This decision guides how we update the cache as detailed in Algorithm 1. The goal is to manage objects so that cache-averse items end up in the candidate queue, while cache-friendly ones are placed in the main queue. The candidate queue consists of objects that the HR-Cache identifies as unlikely to lead to hits, hence prioritized for eviction. Meanwhile, the main queue employs

Table 2: Summary of the traces used in our evaluation.

		Wikipedia 2018	Wikipedia 2019	CloudPhysics	EU Synthetic
Total Requests		84 million	90 million	27 million	100 million
Unique Objects Requested		7 million	11 million	8 million	41 million
Total Bytes Requested		2.6 TB	3.4 TB	360 GB	100 TB
Unique Bytes Requested		0.75 TB	1 TB	86 GB	38 TB
Request Obj Size	**Mean**	34 KB	41 KB	14 KB	1 MB
	Max	674 MB	558 MB	1 MB	7 MB

an LRU strategy, ensuring that, should it become necessary to evict items from the main queue (once the candidate queue is depleted), the items least recently used are evicted first.

For a preliminary evaluation, we use the IBM request trace from Section 3.3 to assess the effectiveness of our learning framework. Given the trace's limited length, we use the initial one million requests to derive HRO decisions as outlined in Section 3.4. Subsequently, we train a model based on these decisions and apply the HR-Cache policy to evaluate the byte hit ratio on the remainder of the trace. As depicted in Figure 3, HR-Cache demonstrates its effectiveness by achieving a Byte Hit Ratio that surpasses the state-of-the-art heuristic policies, even within the limited range of this relatively short trace.

4 EXPERIMENTAL EVALUATION

We developed our framework in C++ as part of a trace-driven simulator designed to accurately assess our framework's miss ratios by replaying cache requests from traces. For the implementation of the Gradient Boosted Decision Trees (GBDT) model, we utilized the LightGBM framework [15]. The code is publicly available on our GitHub repository[2] to facilitate the reproducibility of our proposed research in this paper. Additionally, we introduce an optimization in our implementation in the following section and examine its impact in Section 4.4.

4.1 Batched Predictions

The basic HR-Cache needs to predict cache-friendliness of objects as each request arrives. To take advantage of the architectural strengths of multi-core processors in contemporary CDN and edge servers, we implement data parallelism in our cache decision-making. This modification permits parallel predictions for N requests simultaneously. The chosen batch size, N, plays a critical role in balancing parallelism and miss ratio. A small N fails to fully utilize the potential of parallelism, while an excessively large N can lead to delayed predictions and negatively affect the miss ratio. We selected a batch size of $N = 128$, finding it optimal for harnessing parallelism without affecting our miss ratio. For instance, in our simulations, which do not account for object retrieval overhead, a batch size of $N = 128$ enabled an increase in throughput from handling 11,828 requests per second to 98,404 requests per second, while maintaining cache performance efficiency on the Wiki 2019 trace.

We conduct trace-driven simulations to evaluate the performance of HR-Cache against a broad spectrum of state-of-the-art caching algorithms. Our analysis primarily focuses on two key questions: First, we examine how the byte miss ratio of HR-Cache compares with that of other state-of-the-art research systems across a variety of traces and cache sizes. Second, we assess how HR-Cache performs in relation to the state-of-the-art (SOA) learning-based cache mechanisms, particularly in terms of prediction overhead.

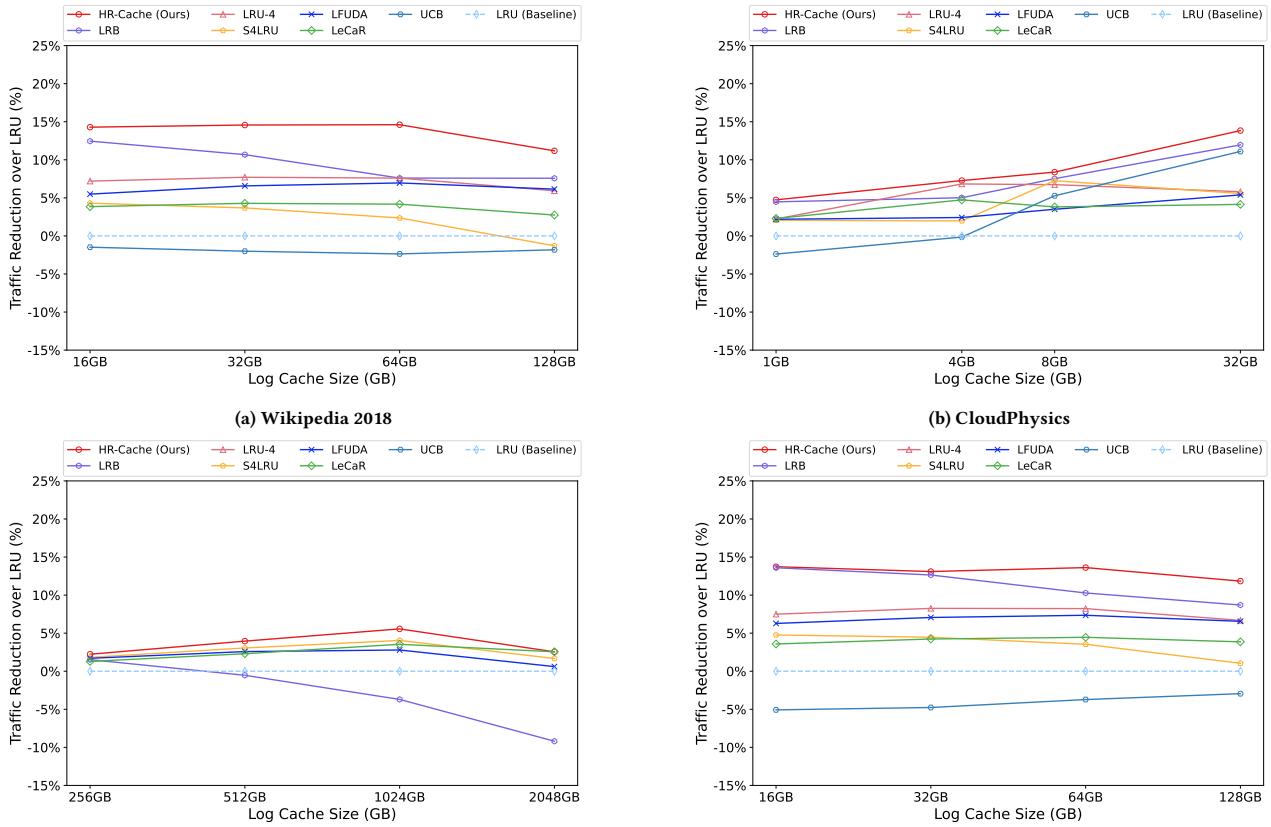
4.2 Experimental Methodology

This subsection describes traces, the experiment setup of our simulation, the competing algorithms, and the parameter settings of HR-Cache. Unless otherwise noted, the reported results for HR-Cache are based on its default operation settings, which include batch-mode inference with a batch size of 128.

Workloads. Our evaluation uses a set of four distinct traces to create a diverse testing environment for the HR-Cache system. This includes two public CDN production traces from Wikipedia for the years 2018 and 2019 [34], a public trace from CloudPhysics [38], and a synthetic trace generated using the JEDI tool in [28, 29]. The selection of these traces aims to represent the performance of HR-Cache across a wide spectrum of real-world and synthetic workloads. Detailed descriptions of each trace source are as follows:

(1) **Wikipedia Traces (2018 and 2019)**: These traces are sourced from Content Delivery Network (CDN) nodes in a metropolitan area in 2018 and 2019, respectively. They mainly consist of web and multimedia content, including images and videos, catering to Wikipedia pages. To reflect the typical environment of edge caches, our evaluations on these traces are conducted with cache sizes of 16 GB, 32 GB, 64 GB, and 128 GB, aligning with the characteristics of smaller cache sizes often found in edge caches [17].
(2) **CloudPhysics Trace**: A Block I/O trace from [38], capturing the activity of VMware virtual disks. This trace introduces a more diverse workload for our analysis, extending beyond the typical CDN scenarios to demonstrate the generalizability of our method across different computing environments. In our analysis of this trace, we chose cache sizes of 1 GB, 4 GB, 8 GB, and 16 GB, reflecting common configurations in virtual machine environments.
(3) **EU Synthetic Trace**: This trace is generated using the JEDI tool [29] which produces traces that have similar caching properties and object-level properties as original production traces. We use the "eu" traffic class which is tailored to replicate the traffic patterns observed in an Akamai's production CDN, specifically those serving content related to social media. For this trace, we use cache sizes of 256 GB, 512 GB, 1 TB,

[2]https://github.com/pacslab/HR-Cache



(a) Wikipedia 2018

(b) CloudPhysics

(c) EU Synthetic

(d) Wikipedia 2019

Figure 4: WAN traffic reduction compared to LRU across various cache sizes for HR-Cache and seven leading algorithms. HR-Cache consistently achieves 2.2–14.6% greater WAN traffic savings than LRU, outperforming the SOA alternatives.

and 2 TB. This decision is based on the trace's large working set size, where smaller cache sizes would not be effective or meaningful for performance analysis.

Table 2 summarizes key properties of the four traces.

State-of-the-art algorithms. In our evaluation, HR-Cache is compared with twelve state-of-the-art caching algorithms: LRB, LRU, LRU-4, S4LRU, GDSF, LFUDA, AdaptSize, Hyperbolic, LHD, LeCaR, and UCB. To enhance readability, we present only the six best-performing algorithms compared to LRU. These are divided into two categories: 1) learning-based algorithms, which include LRB [34], LeCaR [37], and UCB [6]; and 2) heuristics-based algorithms, comprising LRU-4 [24], LFUDA [2], and S4LRU[13].

Parameter	Value
Learning Rate	0.1
Max Depth	50
Number of Trees	100
Max Number of Bins	255
Objective	logistic regression

Table 3: Parameters of the GBDT Model

Experimental Setup. All simulation experiments were run on a Google Cloud server with 24 E2-v CPUs (12 shared physical cores) and 64 GB of RAM. Unless specified otherwise, the reported results for HR-Cache are based on the settings that HR-Cache operates in batch-mode inference with a batch size of 128. We also set the frequency decay factor to 0.9 for the decayed frequency feature. Throughout our evaluation, we utilized the parameters listed in Table 3 for the GBDT model in LightGBM.

We note that the LRB algorithm was run using its default window parameter. The longer duration of this default memory window, in comparison to the lengths of our traces and the sizes of our caches, might have a bearing on its performance. However, any such influence is expected to be advantageous, which contributes to a balanced comparison in our study.

In all our experiments, the initial training window, during which HR-Cache reverts to LRU, is considered a warm-up phase. We report the metrics for HR-Cache and other algorithms after this period. Notably, LRB starts its training ahead of our framework, and thus, this warm-up phase provides enough time for its training phase to start.

4.3 Main Results

We compare HR-Cache with the caching algorithms detailed in Section 4.2, utilizing simulations across various cache sizes. Figure 4 illustrates the reduction in wide-area network (WAN) traffic for each algorithm relative to LRU, across different cache sizes and the six traces.

HR-Cache consistently outperforms existing state-of-the-art algorithms, securing the lowest byte miss ratios across various combinations of traces and cache sizes. The sole exception is observed with the EU Synthetic, size 2048, where HR-Cache achieves performance equivalent to that of LeCar. On average, HR-Cache reduces WAN traffic by over 9.7% compared to LRU, with reductions ranging from 2.2–14.6%. Its robust performance is evident across all traces, unlike other algorithms that lack consistent improvements across varying traces and cache sizes.

For instance, LRU-4 improves performance over LRU in 3 of the workloads, but completely underperforms in the EU traces, resulting in a significant 16-24% increase in traffic over LRU (not depicted in the plot due to being below the y-axis). On the other hand, UCB generally underperforms compared to the other algorithms, with a notable exception in CloudPhysics at 16 GB, where it closely rivals HR-Cache and LRB. Shifting focus to LeCaR and LFUDA, these algorithms consistently outperform LRU, yet they do not manage to surpass the effectiveness of other top-performing policies. LRB, on the other hand, exhibits strong results on the Wiki traces, however, it performs the same or falls short in comparison to HR-Cache even where it performs best. Moreover, LRB is outperformed by heuristic algorithms in an instance of CloudPhysics and EU Synthetic and undergoes a significant decrease in effectiveness in the EU Synthetic trace, particularly as cache sizes increase.

Furthermore, it is important to note that the pattern of WAN traffic reduction achieved by HR-Cache does not consistently correlate with cache capacity. For instance, in the EU Synthetic trace, we observe that the traffic reduction effectively doubles when moving from 256 GB to 1 TB. Conversely, in CloudPhysics, HR-Cache's reduction over LRU generally shows an increasing trend, yet there are instances where the improvement trend inversely declines. This variability suggests that the traces used in our study encompass a diverse array of request patterns, influencing the performance dynamics of HR-Cache differently across scenarios.

Overall, these results suggest that heuristic-based algorithms excel with specific patterns but falter with others. A similar trend is observed among the learning-based algorithms we evaluated. UCB generally underperforms across the board, and LeCaR struggles to match the performance of state-of-the-art alternatives. LRB, although demonstrating strengths in certain scenarios, does not consistently show improvement, underscoring the variability in its efficacy.

4.4 Prediction Overhead Optimization

In this section, we analyze the additional prediction overhead introduced by HR-Cache in comparison with the state-of-the-art LRB algorithm. To understand the source of this overhead in both LRB and HR-Cache, we examine this overhead for the Wiki 2018 trace.

LRB incurs prediction overhead by running predictions on 64 samples for each eviction event. In contrast, HR-Cache requires a prediction for each incoming request to determine cache-friendliness. However, HR-Cache's batch mode significantly reduces this requirement by enabling inference on every 128 requests, rather than on each individual request. As both frameworks utilize the GBDT model, we measure the inference time for batches of 64 (LRB's eviction candidate count) and 128 (HR-Cache's inference batch size) inputs, respectively. The results of these measurements is found in Table 4.

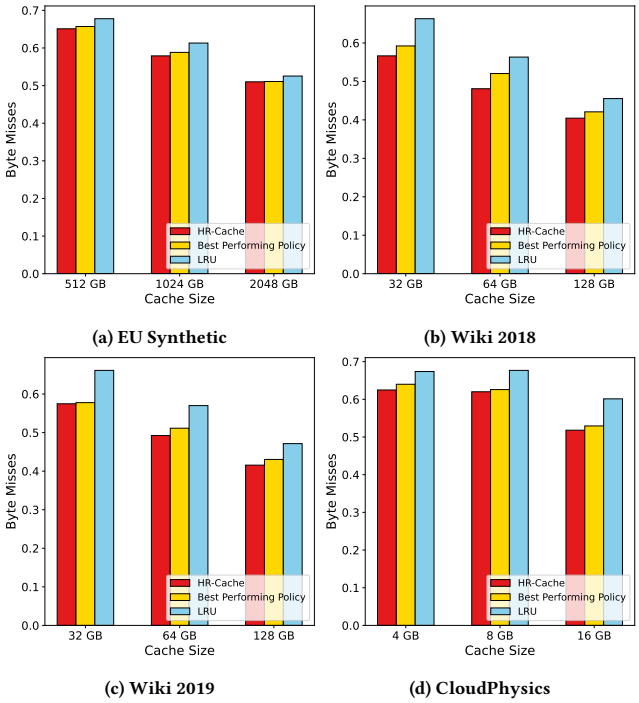


Figure 5: Comparison of Byte Miss Ratios for HR-Cache, the Best Performing Policy, and LRU

	Prediction batch	Prediction time (μs)
LRB	64	183
HR-Cache	128	220

Table 4: Comparison of Prediction Batch Sizes and Prediction Times for LRB and HR-Cache Algorithms

Given that LRB is required to run predictions with every eviction event, its prediction overhead is directly tied to the object miss ratio. For our analysis, we assume the best-case scenario for LRB, where only one object needs to be evicted per cache miss.

Under the Wikipedia 2018 workload for cache sizes of 64 GB and 128 GB, LRB is required to run predictions for 18% and 13% of requests, respectively. In contrast, HR-Cache with a batch size of 128, effectively runs predictions for only 1/128 of requests. Taking this and the measured inference times into account, this translates to a prediction overhead reduction by factors of 19.2x and 13.8x for cache sizes of 64 GB and 128 GB, respectively, when compared to LRB.

Table 5: Prediction Overhead Reduction For Wiki 2018

Alg.	Miss Ratio		Pred Time ($\mu s/req$)		Reduction Factor	
	64 GB	128 GB	64 GB	128 GB	64 GB	128 GB
LRB	0.18	0.13	32.94	23.8	-	-
HR-Cache	-	-	1.72	1.72	19.2x	13.8x

Another aspect of overhead comes from the process of feature building. HR-Cache constructs one feature per request, while LRB, in contrast, needs to build 64 features on each object miss. This difference results in a significant reduction of overhead for HR-Cache. Specifically, under the Wiki 2018 workload for cache sizes of 64 GB and 128 GB, HR-Cache achieves a reduction in feature-building overhead by factors of 11.5x and 8.3x, respectively, compared to LRB.

To illustrate HR-Cache's computational burden, consider the Wiki 2018 trace with a cache size of 64: replaying 84 million requests, conducting frequent training and inference, and updating our cache based on these predictions, takes approximately 13 minutes, which is more than acceptable given the inter-arrival request rates for objects.

4.5 Ablation

In Section 3.4, we discussed how hit or miss outcomes determined by hazard ordering may not directly correspond to cache decisions. This is because Hazard Rate Ordering (HRO) assumes objects with the highest hazard rates are always pre-fetched and available in the cache whenever a request occurs at time t. Therefore, if a request for an object at time t is a hit, we previously classify it as cache-friendly in its last request. We also noted that modeling the request process as a Poisson process is a simplification, even though it offers a less complex method for calculating hazard rates. Under this process, the hazard rate remains constant. In our study, we conduct an ablation analysis on three of the traces, where we remove the look-back option in one scenario. In another, we operate HR-Cache under the Poisson assumption, as opposed to using kernel hazard estimation. Figure 6 shows the relative gains of our assumptions. On the Wiki 2018 trace, the look-back option assumption significantly influence performance, a trend also observed in the CloudPhysics and EU traces. Under the Poisson assumption, the performance on the Wiki 2018 trace is markedly diminished, whereas this assumption has a minimal impact on the other two traces. This disparity also confirms our hypothesis that the Poisson assumption may not be universally applicable to all real-world traces.

5 CONCLUSION AND FUTURE WORK

In this paper, we proposed HR-Cache, a novel learning-based caching framework for edge environments. It learns from hazard rate ordering decisions to identify cache-averse objects and prioritizes them for eviction. HR-Cache comprises two main components: it reconstructs the hazard rate ordering on a window of requests using kernel hazard estimation and a decision tree classifier that learns to predict the cache-friendliness of incoming requests. We evaluated our framework using real-world data traces and compared it with several state-of-the-art caching strategies. Our results indicate that HR-Cache significantly improves the byte hit rate compared to LRU

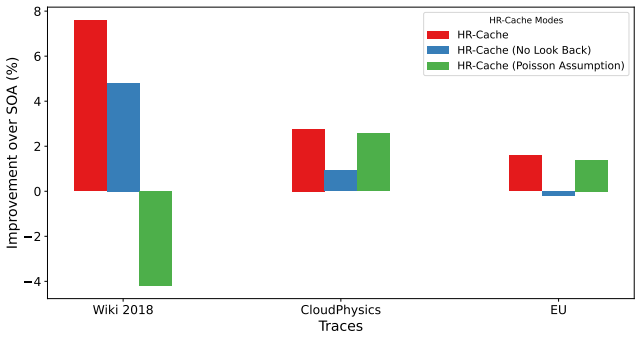


Figure 6: Percentage improvement of HR-Cache modes over the state-of-the-art method.

and surpasses a wide range of state-of-the-art policies, all while maintaining minimal prediction overhead compared to contemporary learning-based cache policies. Further experiments were conducted to confirm the positive impact of our specific design choices and assumptions on HR-Cache's performance, highlighting their validity and effectiveness.

We envision two main ways for future exploration to enhance HR-Cache applicability and performance in real-world scenarios. The first involves extending HR-Cache for distributed caching environments. Exploring the potential of HR-Cache in distributed environments opens avenues for leveraging federated learning to pool insights from diverse data sources, enhancing model accuracy while adhering to privacy concerns. This approach, however, introduces challenges, particularly with non-IID data, which could affect model performance. Additionally, adapting HR-Cache to hierarchical cache architectures allows us to optimize cache utilization across different levels and closely mirrors the operational structures of real-world CDN and edge caches. This adaptation, however, necessitates navigating the intricacies of cache dynamics within such structures. The second avenue focuses on integrating HR-Cache into production cache systems, necessitating adjustments to overcome hardware limitations and ensure seamless operation. This effort will extend to evaluating HR-Cache's impact on latency, resource consumption, and scalability under practical conditions.

ACKNOWLEDGMENT

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the NSERC CREATE grant, "Dependable Internet of Things Applications (DITA)."

REFERENCES

[1] Alfred V Aho, Peter J Denning, and Jeffrey D Ullman. 1971. Principles of optimal page replacement. *Journal of the ACM (JACM)* 18, 1 (1971), 80–93.
[2] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. 2000. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review* 27, 4 (2000), 3–11.
[3] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.
[4] Daniel S Berger. 2018. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. 134–140.
[5] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 1–38.

[6] Renato Costa and Jose Pazos. 2017. *Mlcache: A multi-armed bandit policy for an operating system page cache.* Technical Report. Technical report, University of British Columbia.

[7] Daryl J Daley. 2014. *Introduction to the Theory of Point Processes: Elementary Theory and Methods.* Springer.

[8] Andrés Ferragut, Ismael Rodríguez, and Fernando Paganini. 2016. Optimizing TTL caches under heavy-tailed demands. *ACM SIGMETRICS Performance Evaluation Review* 44, 1 (2016), 101–112.

[9] Giulio Garbi, Emilio Incerto, and Mirco Tribastone. 2020. Learning queuing networks by recurrent neural networks. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 56–66.

[10] Yu Guan, Xinggong Zhang, and Zongming Guo. 2019. Caca: Learning-based content-aware cache admission for video content in edge caching. In *Proceedings of the 27th ACM International Conference on Multimedia*. 456–464.

[11] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. 2014. Trade-offs in optimizing the cache deployments of CDNs. In *IEEE INFOCOM 2014-IEEE conference on computer communications*. IEEE, 460–468.

[12] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. 2022. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*. 72–90.

[13] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 167–181.

[14] Akanksha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 78–89.

[15] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[16] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K Sitaraman. 2019. Rl-cache: Learning-based cache admission for content delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*. 57–63.

[17] Dong Liu, Binqiang Chen, Chenyang Yang, and Andreas F Molisch. 2016. Caching at the wireless edge: design aspects, challenges, and future directions. *IEEE Communications Magazine* 54, 9 (2016), 22–28.

[18] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*. PMLR, 6237–6247.

[19] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2020. Bao: Learning to steer query optimizers. *arXiv preprint arXiv:2004.03814* (2020).

[20] Kianoosh Mokhtarian and Hans-Arno Jacobsen. 2014. Caching in video CDNs: Building strong lines of defense. In *Proceedings of the ninth European conference on computer systems*. 1–13.

[21] Hans-Georg Muller and Jane-Ling Wang. 1994. Hazard rate estimation under random censoring with varying kernels and bandwidths. *Biometrics* (1994), 61–76.

[22] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 985–1000.

[23] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. 2010. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review* 44, 3 (2010), 2–19.

[24] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[25] Nitish Kumar Panigrahy. 2021. Resource Allocation in Distributed Service Networks. (2021).

[26] Nitish K Panigrahy, Philippe Nain, Giovanni Neglia, and Don Towsley. 2022. A new upper bound on cache hit probability for non-anticipative caching policies. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 7, 2-4 (2022), 1–24.

[27] Georgios Paschos, Ejder Bastug, Ingmar Land, Giuseppe Caire, and Mérouane Debbah. 2016. Wireless caching: Technical misconceptions and business barriers. *IEEE Communications Magazine* 54, 8 (2016), 16–22.

[28] Anirudh Sabnis and Ramesh K Sitaraman. 2021. TRAGEN: a synthetic trace generator for realistic cache simulations. In *Proceedings of the 21st ACM Internet Measurement Conference*. 366–379.

[29] Anirudh Sabnis and Ramesh K Sitaraman. 2022. JEDI: model-driven trace generation for cache simulations. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 679–693.

[30] Alireza Sadeghi, Gang Wang, and Georgios B Giannakis. 2019. Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. *IEEE Transactions on Cognitive Communications and Networking* 5, 4 (2019), 1024–1033.

[31] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 413–425.

[32] Junaid Shuja, Kashif Bilal, Waleed Alasmary, Hassan Sinky, and Eisa Alanazi. 2021. Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey. *Journal of Network and Computer Applications* 181 (2021), 103005.

[33] Jiongjiong Song, Min Sheng, Tony QS Quek, Chao Xu, and Xijun Wang. 2017. Learning-based content caching and sharing for wireless networks. *IEEE Transactions on Communications* 65, 10 (2017), 4309–4324.

[34] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. 2020. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 529–544.

[35] Stefano Traverso, Mohamed Ahmed, Michele Garetto, Paolo Giaccone, Emilio Leonardi, and Saverio Niccolini. 2015. Unravelling the impact of temporal and geographical locality in content caching systems. *IEEE Transactions on Multimedia* 17, 10 (2015), 1839–1854.

[36] Shreshth Tuli and Giuliano Casale. 2022. Optimizing the Performance of Fog Computing Environments Using AI and Co-Simulation. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. 25–28.

[37] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving cache replacement with {ML-based}{LeCaR}. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[38] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient {MRC} construction with {SHARDS}. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 95–110.

[39] Jane-Ling Wang et al. 2005. Smoothing hazard rates. *Encyclopedia of biostatistics* 7 (2005), 4986–4997.

[40] Xiaofei Wang, Chenyang Wang, Xiuhua Li, Victor CM Leung, and Tarik Taleb. 2020. Federated deep reinforcement learning for internet of things with decentralized cooperative edge caching. *IEEE Internet of Things Journal* 7, 10 (2020), 9441–9455.

[41] Gang Yan, Jian Li, and Don Towsley. 2021. Learning from optimal caching for content delivery. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 344–358.

[42] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. 2018. A deep reinforcement learning-based framework for content caching. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 1–6.

Function Offloading and Data Migration for Stateful Serverless Edge Computing

Matteo Nardelli
matteo.nardelli@bancaditalia.it
Bank of Italy
Italy

Gabriele Russo Russo
russo.russo@ing.uniroma2.it
Tor Vergata University of Rome
Italy

ABSTRACT

Serverless computing and, in particular, Function-as-a-Service (FaaS) have emerged as valuable paradigms to deploy applications without the burden of managing the computing infrastructure. While initially limited to the execution of stateless functions in the cloud, serverless computing is steadily evolving. The paradigm has been increasingly adopted at the edge of the network to support latency-sensitive services. Moreover, it is not limited to stateless applications, with functions often recurring to external data stores to exchange partial computation outcomes or to persist their internal state. To the best of our knowledge, several policies to schedule function instances to distributed hosts have been proposed, but they do not explicitly model the data dependency of functions and its impact on performance.

In this paper, we study the allocation of functions and associated key-value state in geographically distributed environments. Our contribution is twofold. First, we design a heuristic for function offloading that satisfies performance requirements. Then, we formulate the state migration problem via Integer Linear Programming, taking into account the heterogeneity of data, its access patterns by functions, and the network resources. Extensive simulations demonstrate that our policies allow FaaS providers to effectively support stateful functions and also lead to improved response times.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **General and reference** → General conference proceedings; • **Computing methodologies** → *Distributed computing methodologies.*

KEYWORDS

serverless, scheduling, data migration, edge computing, cloud computing

ACM Reference Format:
Matteo Nardelli and Gabriele Russo Russo. 2024. Function Offloading and Data Migration for Stateful Serverless Edge Computing. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3649293

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05…$15.00
https://doi.org/10.1145/3629526.3649293

1 INTRODUCTION

Serverless computing enables the deployment of applications without the burden of managing the computing infrastructure (e.g., [5]). In the last few years, we are witnessing the increasing adoption of serverless for organizations using cloud services; for example, in 2023, Datadog declared that up to 70% of its customers use at least one serverless function[1]. The diffusion of serverless has been also boosted by the Function-as-a-Service (FaaS) offering by cloud providers, where customers can execute code (or functions) in response to events, thus drastically simplifying the infrastructure operations typically associated with microservices applications.

Recently, two major trends have involved serverless computing. First, besides traditional cloud environments, serverless has been increasingly adopted at the edge of the network as well (e.g., [20, 22]). Second, its adoption is not limited to stateless functions any more, with functions often recurring to external data stores to exchange partial computation outcomes or to persist their internal state (e.g., [5, 19, 31]). Function state usually consists of key-value pairs (e.g., [13, 26, 30, 31]). While in cloud environments it is reasonable to rely on centralized storage services to save state information (e.g., object storage, in-memory stores), in edge-cloud environments state can be distributed across edge and cloud nodes to increase data locality. Therefore, traditional cloud serverless platforms, such as OpenWhisk and OpenFaaS, do not well fit the features of the emerging environment (e.g., [8]). First, they do not consider latency and bandwidth between edge-cloud resources, which can be particularly relevant for functions with stringent latency requirements. Second, by not explicitly considering data dependencies of functions, they do not often optimize for data locality or data movement. Most of recent research efforts on stateful serverless focus on running functions in logically centralized clouds, thus neglecting the impact of heterogeneous resources on performance (e.g., [9, 26, 27]). As functions usually recur to external data stores to persist their state, other research efforts propose to make explicit data intents of functions (e.g., [20, 27]), to co-locate functions with their data (e.g., [26, 28]), or to optimize over an explicit data dependency model (e.g., [2, 19]). While the first approach moves the complexity of managing data dependencies to users and developers, the others leverage rather simple data models, which do not consider the heterogeneity of data and access patterns, as well as its geographic distribution. To the best of our knowledge, the efficient execution of stateful serverless edge functions is still an open issue.

[1] https://www.datadoghq.com/state-of-serverless/

In this paper, we study the allocation of functions and associated state in geo-distributed environments. Since the performance uncertainty affecting FaaS platforms is a frequent concern for developers, we envision a setting where developers and serverless providers stipulate a Service Level Agreement (SLA), enabling the autonomous execution and management of stateful serverless functions with possibly stringent performance requirements. The SLA helps define the target function performance, through Service Level Objectives (SLOs), to be met at run-time as well as the penalties paid by the platform provider whether these expectations are disregarded. We focus on policies for such a serverless edge computing environment. Differently from existing works (e.g., [2, 19, 20]), we explicitly model the heterogeneity of computing, storage, and networking devices as well as data dependencies of functions. The core contributions of this paper are as follows. First, we present a conceptual *Decentralized Stateful Serverless Platform* (DS2P) that includes the mechanisms where the proposed policy can be installed. Second, we present a *SLO-aware offloading* strategy for allocating functions to edge and cloud nodes that explicitly considers the time to access their data and aims to meet the SLOs stipulated in SLAs. Then, we also propose a *state-aware data migration* policy that relocates data at run-time, aiming to further improve application performance while limiting penalty costs paid by the platform provider. Ultimately, we extensively evaluate the proposed policies by investigating the impact of different configuration parameters as well as their scalability.

The remainder of the paper is organized as follows. We present the system model, the problem under investigation, and the main assumptions in Sect. 2. In Sect. 3, we overview the proposed solution by presenting DS2P, which offers mechanisms for running the policy we focus on in this paper. The function offloading and data migration policies are then presented in Sect. 4 and evaluated in Sect. 5. In Sect. 6, we discuss related works and conclude in Sect. 7.

2 SYSTEM MODEL AND PROBLEM STATEMENT

We consider serverless applications, where user-defined code can be executed without allocating and managing virtualized servers and resources, or being concerned about other operational aspects. The responsibility for operational aspects is offloaded to the service provider. Serverless applications consist of multiple functions, whose trigger is usually an HTTP request, a cloud event, or a scheduler. Functions are commonly used to implement APIs, asynchronous processing, batch tasks, or operations tasks [5]. Most functions are very short-lived, running for less than 1 minute (also due to execution time constraints imposed by service providers). Functions can be stateful or stateless, whether they need persistent data to answer requests. Hereafter, we focus on stateful functions.

A serverless platform manages serverless applications to make them available to users. Users from different network locations can request function execution to obtain a service (e.g., [3, 18]) or to manipulate data (e.g., [4, 11, 29]). To preserve the state across invocations, serverless functions usually rely on external store to persist data (e.g., [5, 11, 19]). According to [5], data volumes handled by serverless applications follow a bimodal distribution: although 53% of applications handle less than 1 MB, 16% between 1 MB and 10 MB, a second peak of 16% of them handle more than 1 GB.

To run function instances, the serverless platform could leverage geographically distributed edge-cloud computing environment, where edge and cloud nodes provide computing and storage resources. Let F be the set of all functions. Each function $f \in F$ is characterized by a memory demand $m_f \in \mathbb{R}$ and an execution time on a reference architecture, $r_f \in \mathbb{R}$. We model application state as key-value pairs, where different keys can be allocated to different storage nodes (e.g., [1, 14, 26, 31]). Let K be the set of key-value pairs, where each key $k \in K$ is associated with a value, whose size is l_k. We assume that each function $f \in F$ can access a subset of key-value pairs in K (with some keys possibly accessed by multiple functions). As computing environment, we consider a system comprising N distributed cloud and edge nodes. Each node in $i \in N$ has limited computing C_i, memory M_i, and storage S_i capacity. Nodes $i, j \in N$ are interconnected with non-negligible network latency $d_{i,j}$, i.e., the logical network link (i, j) has $d_{i,j} \geq 0$. We denote by $b_{i,j}$ the bandwidth of the logical network link (i, j). Different disjoint subsets of keys $K' \subseteq K$ can be hosted on different nodes, whereas a single key is hosted on a single node. We consider a per-key placement, e.g., through client-side partitioning[2], enabling keys to be possibly distributed independently from one another. Although sophisticated replication strategies can be designed, during our first iteration of this study, we assume no data replication. Each node handling a portion of the state can be used for both reading and writing operations.

Problem Statement. Determining the allocation of functions and data is critical to run latency-sensitive serverless functions. In the computing environment under investigation, users request function execution from different locations. These functions can access different portions of the data store, i.e., subsets of the keys in K, which can be possibly located on far away nodes. Moving data across different network locations introduces latency, which, in turn, increases the overall function completion time. Moving large data volumes might also introduce prohibitively long delays, calling for adaptive relocation of function code and its execution on a node close to the data store. Each function can exhibit its data access pattern, which may also vary depending on the geographical zone where the function is executed. The key issue relates to the hardness of statically (i) distributing keys to data store instances and (ii) deciding where function execution requests should be scheduled (i.e., locally or to a remote computing node via offloading).

We assume that function developers stipulates a Service Level Agreement (SLA) with the serverless platform provider, including Service Level Indicators (SLIs), i.e., metrics, and Service Level Objectives (SLOs), i.e., predicate over SLIs. Data location impacts on the function running time. Therefore, we consider that each function $f \in F$ exposes a SLO on the maximum data access latency, i.e., $T_f^{\max}$, that should be met at run-time.[3] Otherwise, the serverless platform provider incurs in a penalty cost $\rho_f \in \mathbb{R}^+$ per unit of extra time to access data that exceeds $T_f^{\max}$.

[2] https://redis-doc-test.readthedocs.io/en/latest/topics/partitioning/

[3] As it will be evident in the following, our approach can be readily adapted to cope with different requirements (e.g., maximum response time).

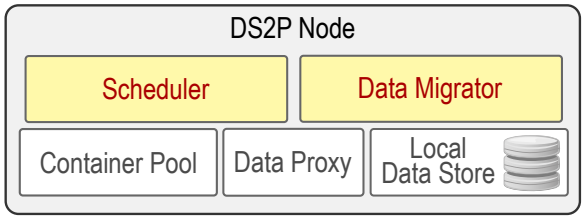


Figure 1: Overview of the architecture of a DS2P node.

Two key problems are considered. First, we need to identify a *function execution and offloading* policy that determines where to run a function instance among all the available (possibly geo-distributed) computing resources. Second, we need to define a *data migration* policy that possibly relocates data at run-time, so as to satisfy the SLA between the function developer and the serverless platform provider, in face of the current workload.

Assumptions. Any client can request the execution of any function in F. To run a function, a client sends an execution request to the closest node $n \in N$ in the computing environment (likely, an edge node in their proximity). Therefore, assuming that invocation requests are always directed to the closest available node, we do not explicitly model client locations in this work. Instead, we only account for the locations of nodes in N, in terms of network latency among them.

The node $n \in N$ receiving the client request is responsible for scheduling the execution of the requested function: the node can possibly decide to offload the execution to another node and, hence, determine how data should be retrieved. To this end, we assume that each node can instantiate any function as it caches a copy of the code for all functions. To support offloading, nodes periodically exchange state information, e.g., using a gossiping protocol (e.g., a similar approach is implemented in [22]).

3 SOLUTION OVERVIEW

We propose Decentralized Stateful Serverless Platform (for short, DS2P), an abstract decentralized FaaS platform designed for edge-cloud computing environments. DS2P manages resources for function execution upon invocation and data storage. As most of FaaS platforms, including OpenWhisk and OpenFaaS, functions run within software containers, which are spawned as needed and initialized with the required code and libraries. A DS2P node can run on edge and on cloud resources; it implements the functionality to receive execution requests, run functions, and interact with other DS2P nodes. In DS2P, there are not privileged entry points for function invocation: every node is able to schedule the execution of incoming requests. Since edge nodes can have limited resource capacity, DS2P allows nodes to *offload* execution requests to other nodes, when needed.

The architecture of a DS2P node is depicted in Fig. 1. Its main components include a Scheduler, a Local Data Store, a Data Proxy, a Data Migrator, and a Container Pool. The *Scheduler* oversees resource allocation for function execution, as it decides whether to execute requests locally, to offload them to other nodes, or to drop them (e.g., during heavy-load periods). To this end, the Scheduler can be equipped with custom policies, such as the one we present in Sect. 4.1. The *Local Data Store* is a key-value store aimed to store function data. It can be implemented using, e.g., Redis, Hazelcast, or Anna. The *Data Proxy* has the responsibility of (i) proxying data access operations, and (ii) characterizing the workload of each function (i.e., which keys they use, type of operation, frequency). The Data Proxy internally exposes a *data distance table*, which reports the estimated latency due to data read/write operations, to the Scheduler and Data Migrator. The *Data Migrator* can periodically relocate key-value pairs by interacting with other DS2P nodes. In the decentralized architecture of DS2P, each Data Migrator handles a subset of keys, i.e., those physically co-located on the DS2P node. Nonetheless, in the following, we assume a logically centralized Migrator that manages all keys stored in the system. Sect. 4.2 proposes a policy to determine when and which key should be more conveniently relocated.

Table 1: Main notation adopted in the paper

Symbol	Description
F	Set of functions
m_f	Memory demand of $f \in F$
r_f	Execution time on a reference architecture of $f \in F$
$T_f^{\max}$	Maximum data access latency expressed by $f \in F$
ρ_f	Unit penalty cost paid when $T_f^{\max}$ is exceeded
K	Set of keys
l_k	Size of value associated to key $k \in K$
$p_{f,k}$	Estimated key access probability $k \in K$ by $f \in F$
N	Set of edge nodes
C_i	Computing capacity of node $i \in N$
M_i	Memory capacity of node $i \in N$
S_i	Storage capacity of node $i \in N$
s_i	Computation speed-up of node $i \in N$
$d_{i,j}$	Network latency between $i, j \in N$
$b_{i,j}$	Network bandwidth between $i, j \in N$

4 FUNCTION AND DATA ALLOCATION POLICIES

In this section, we propose the policies for solving the function execution and offloading problem, as well as the data migration problem. Main notation is reported in Table 1.

4.1 Function Scheduling and Offloading

We propose a *SLO-aware offloading* policy that aims to allocate functions by minimizing their completion time and by fulfilling the SLO on the data access time. Basically, SLO-aware offloading estimates the function completion time both on local node and on other nodes. Since we consider stateful functions, this time is influenced by the access latency of keys used by the function. Then, the policy selects the configuration leading to minimum function completion time, among those satisfying the SLO.

For each function $f \in F$, we maintain an estimate of its key access probability, i.e., $p_{f,k} \in [0, 1]$ with $k \in K$. Such information can be estimated by tracing function data accesses (either in historical traces or at run-time) or, if possible, through static code analysis.

Local Execution. When a node $i \in N$ receives an execution request for function $f \in F$, it checks the execution configuration (i.e., local or offloading) leading to lower response time. First, it estimates the completion time T_i^f assuming that the function f will be executed locally on i:

$$T_i^f = T_{f,i}^{\text{exec}} + T_{f,i}^{\text{data}} \tag{1}$$

this term consists of two contributions, respectively the code execution time of f, $T_{f,i}^{\text{exec}}$,[4] and the average data access time $T_{f,i}^{\text{data}}$, which can be defined as follows:

$$T_{f,i}^{\text{exec}} = \frac{r_f}{s_i} \tag{2}$$

$$T_{f,i}^{\text{data}} = \sum_{k \in K} p_{f,k} T_{f,i,\text{loc}(k)}^{\text{data}(k)} \tag{3}$$

where r_f is the function execution time on a reference architecture, s_i is the computation speed-up of node i, $p_{f,k}$ is the access probability of k by the instance of f, and $\text{loc}(k)$ returns the node hosting key k. $T_{f,i,j}^{\text{data}(k)}$ is the access time of state k by f, when f runs on i and k is stored on $j \in N$, with $j = \text{loc}(k)$; it models the delay to copy the value corresponding to the key from its location j to i:

$$T_{f,i,j}^{\text{data}(k)} = d_{i,j} + \frac{l_k}{b_{i,j}} + d_{j,i} \tag{4}$$

where $d_{i,j}$ is the network delay between i and j, l_k is the size of data associated to k, and $b_{i,j}$ is the bandwidth between nodes i and j.[5]

Local execution of f is *admissible*, if $T_{f,i}^{\text{data}} \leq T_f^{\max}$ and its required memory fits the spare capacity of i, i.e., $m_f \leq \bar{M}_i$.

Offloading. Afterwards, the node i estimates the time to run $f \in F$ when the computation is offloaded to another node. Due to data dependencies, we only consider as candidate the nodes hosting at least a key accessed by f. Having to transfer function inputs and later collect its output, the completion time of f, when offloaded from i to j, with $i, j \in N$, is defined as follows:

$$T_{i,j}^f = T_{i,j}^{in} + T_{j,i}^{out} + T_{f,j}^{\text{exec}} + T_{f,j}^{\text{data}} \tag{5}$$

where $T_{i,j}^{in}$ is the time to move the function input from i to j, $T_{i,j}^{out}$ is the time to collect the computation results from j back to i, $T_{f,j}^{\text{exec}}$ is the time to execute f on j (as in (2)), and $T_{f,j}^{\text{data}}$ is the average data access time of f in j (as in (3)). The first two terms can be formally defined as follows:

$$T_{i,j}^{in} = \frac{l_f^{\text{in}}}{b_{i,j}} + d_{i,j} \tag{6}$$

$$T_{j,i}^{out} = \frac{l_f^{\text{out}}}{b_{j,i}} + d_{j,i} \tag{7}$$

where l_f^{in} and l_f^{out} represent respectively the size of input and output data.

Offloading $f \in F$ to $j \in N$ is *admissible*, if $T_{f,j}^{\text{data}} \leq T_f^{\max}$ and $m_f \leq \bar{M}_j$, where $\bar{M}_j$ is the spare memory capacity of j.

Scheduling. At this point, the node i has all the ingredients to determine where to schedule the function. Among admissible configurations, the policy chooses the one leading to lower expected completion time: the best candidate for offloading is the node j^* with minimum completion time T_{i,j^*}^f, i.e., $j^* = \arg\min_{j \in N'} \left(T_{i,j}^f\right)$, among admissible nodes N'. If local execution is eligible and $T_i^f \leq T_{i,j*}^f$, then f is executed on i; otherwise, f is offloaded to j^*. If no configuration is admissible, the node i returns an error as f cannot be executed while meeting the agreed SLA.

4.2 Data Migration

The Data Migrator can periodically relocate keys among nodes so to reduce the completion time of functions by lowering their data access time. We formulate *state-aware data migration* as an Integer Linear Programming (ILP) problem.

Let $x_i^{(k)} \in \{0, 1\}$ be a binary variable such that $x_i^{(k)} = 1$ if $k \in K$ is allocated on $i \in N$; 0 otherwise. The data migration policy can relocate keys across resources aiming to fulfill the SLA of functions. We consider SLAs with a SLO on the average data access time.

Data Access Time. We define as $T_f^{\text{data}(k)}$ the average data access time of state $k \in K$ experienced by function $f \in F$:

$$T_f^{\text{data}(k)} = \sum_{i \in N} \sum_{j \in N} p_{f,k} T_{f,i,j}^{\text{data}(k)} x_j^{(k)} \tag{8}$$

where $T_{f,i,j}^{\text{data}(k)}$ is the average access time of state $k \in K$ stored in $j \in N$, when f runs on $i \in N$; it is defined in (4).

Migration Time. Data relocation does not come for free, as we need to transfer one or more key-value pairs across the network. In some cases, this activity may be too expensive and we may prefer to completely avoid it. In the data migration policy we model migration time as follows:

$$T_{i,j}^{\text{mig}(k)} = \left(d_{i,j} + \frac{l_k}{b_{i,j}} + d_{j,i}\right) y_{i,j}^{(k)} \tag{9}$$

where $y_{i,j}^{(k)}$ is a binary variable indicating whether $k \in K$ has to migrated from $i \in N$ to $j \in N$.

We can define $y_{i,j}^{(k)}$ through auxiliary constant terms, $\bar{x}_i^{(k)}$, indicating where k was hosted on i before solving the ILP. Conceptually, $y_{i,j}^{(k)}$ represents the logical AND between $\bar{x}_i^{(k)}$ and $x_j^{(k)}$; formally, we define $y_{i,j}^{(k)}$ as:

$$y_{i,j}^{(k)} \leq x_j^{(k)} \qquad \forall i, j \in N \tag{10}$$

$$y_{i,j}^{(k)} \leq \bar{x}_i^{(k)} \qquad \forall i, j \in N \tag{11}$$

$$y_{i,j}^{(k)} \geq \bar{x}_i^{(k)} + x_j^{(k)} - 1 \qquad \forall i, j \in N \tag{12}$$

[4] In this work, we assume that invocation requests do not experience any queueing delay, similarly to what happens in commercial FaaS platforms (e.g., AWS Lambda). Moreover, we assume that performance interference among functions is negligible in the considered computing environment. Note that—with no loss of validity of the proposed approach—more complex models of the execution time can be considered instead.

[5] We consider a simple model of data transfer times. More accurate models accounting, e.g., for possible packet re-transmissions and pipelining issues, can be plugged into our approach with no loss of validity.

SLA Violations. To identify the SLA violations for function f, we resort to *excess* variables $\delta_f \in \mathbb{R}^+$ and $\delta_{f,k} \in \mathbb{R}^+$, and *slack* variables $z_{f,k} \in \mathbb{R}^+$; they indicate how much the data access time is, respectively, above or below the SLO. Formally, we have:

$$T_f^{\text{data}(k)} + z_{f,k} = T_f^{\max} + \delta_{f,k} \qquad \forall k \in K, f \in F \tag{13}$$

$$\delta_f \geq \delta_{f,k} \qquad \forall k \in K, f \in F \tag{14}$$

We assume that, when the SLA is violated, the serverless platform provider pays, to the owner of f, a penalty P_f proportional to δ_f:

$$P_f = \rho_f \delta_f \qquad \forall f \in F \tag{15}$$

where ρ_f is the unit of penalty cost. Other formulations of P_f could be easily considered.

Objective Function. To compute the migration strategy, we need to express the goal function, which formally defines the idea of "best" data placement among all possible solutions. We resort to a simple function $\mathcal{F}$ that minimizes the violations of the functions' SLA and the time to migrate key-value pairs (9). The latter controls how often data should be relocated: ideally, we want to avoid moving *heavy* data too often. We formulate the objective function of state-aware data migration as:

$$\textbf{min} \quad \mathcal{F} + Z \tag{16}$$

with:

$$\mathcal{F} = \sum_{f \in F} w_p P_f + \sum_{k \in K} \sum_{i \in N} \sum_{j \in N} w_m \frac{T_{i,j}^{\text{mig}(k)}}{T_{\max}^{\text{mig}(k)}} \tag{17}$$

$$Z = \sum_{f \in F} \sum_{k \in K} z_{f,k} \tag{18}$$

where $\mathcal{F}$ is the objective function and Z is a technical term to correctly define the slack and excess variables. Moreover, $w_p, w_m \geq 0$, $w_p + w_m = 1$, are weights for the different contributions of the objective function, and $T_{\max}^{\text{mig}(k)}$ is a normalization term defined as $T_{\max}^{\text{mig}(k)} = \max_{i,j \in N \times N} \left(T_{i,j}^{\text{mig}(k)} \right)$.

Constraints and Domain. Storing a key-value pair on a node requires modeling capacity constraints, namely:

$$\sum_{k \in K} l_k x_j^{(k)} \leq \bar{S}_j \qquad \forall j \in N \tag{19}$$

$$\sum_{j \in N} x_j^{(k)} = 1 \qquad \forall k \in K \tag{20}$$

where $\bar{S}_j$ is the spare storage capacity of node j before solving the optimization problem, and (20) ensures that a specific key is allocated on a single node at a time. $\bar{S}_j$ can be readily defined as $\bar{S}_j = S_j - \sum_{k \in K} l_k \bar{x}_j^{(k)}$.

Table 2: Average execution time, memory demand and default arrival rate of the functions used in the experiments.

Function f	r_f	m_f	λ_f
f_1	0.40 s	512 MB	8 req/s
f_2	0.20 s	512 MB	16 req/s
f_3	0.30 s	128 MB	42 req/s
f_4	0.25 s	1024 MB	6 req/s
f_5	0.45 s	256 MB	14 req/s

The ILP formulation also includes domain constraints for each variable used in all the previous equations:

$$\delta_f \in \mathbb{R}^+ \qquad \forall f \in F \tag{21}$$

$$\delta_{f,k} \in \mathbb{R}^+ \qquad \forall k \in K, f \in F \tag{22}$$

$$z_{f,k} \in \mathbb{R}^+ \qquad \forall k \in K, f \in F \tag{23}$$

$$y_{i,j}^{(k)} \in \{0,1\} \qquad \forall k \in K, i, j \in N \tag{24}$$

$$x_i^{(k)} \in \{0,1\} \qquad \forall k \in K, i \in N \tag{25}$$

Equations (8)–(25) formulate the state-aware data migration problem.

5 EVALUATION

We evaluate the proposed policies by simulation. We implement the simulator in Python.[6] In this section, we first describe the experimental setup and then present the results.

5.1 Experimental Setup

Infrastructure. We consider a FaaS system comprising a total of 10 nodes, divided as 5 cloud nodes and 5 edge nodes, which can both execute functions and store key-valued state. The infrastructure comprises an additional cloud node, indicated as *data store*, which only stores state information. Memory capacity is set to 64 GB for cloud nodes and 4 GB for edge nodes. For simplicity, we consider an identical CPU speedup value for all the nodes. We randomly generate network latency values for all the pairs of nodes in the infrastructure. Specifically, edge-to-edge latency is uniformly sampled from $[1, 20]$ ms; latency between edge and cloud nodes (including the data store) is uniformly sampled from $[5, 100]$ ms; cloud-to-cloud latency is uniformly sampled from $[1, 10]$ ms for FaaS nodes, and from $[1, 20]$ ms for cloud-to-data store communication. As regards network bandwidth, we set it to 100 Mbps for edge-to-edge and edge-to-cloud communication, and 1 Gbps for cloud-to-cloud communication.

Functions and invocations. We consider 5 functions, with different resource demands, as indicated in Table 2, and exponentially distributed execution times. Invocation requests to the functions are modeled as independent Poisson arrival processes, where the invocation rate (reported in Table 2) is chosen to have equal expected memory utilization for all functions.

We consider two workload settings, where (i) invocation requests for all the functions are uniformly distributed across edge nodes (**W1**); and, (ii) invocations request for each function are directed

[6]The software is publicly available: https://zenodo.org/doi/10.5281/zenodo.10590302

Table 3: Experimental results in the workload scenario W1.

	Policies		Avg. Penalty			Resp. Time (s)	
T_f^{max} (s)	Data Placement	Offloading	($/req) $\times 10^{-1}$	SLO Viol. (%)	Migr. Data (MB)	P_{50}	P_{95}
0.1	No Migration	Greedy-local	1.7522	61.3518	0.0	0.5833	1.6209
0.1	No Migration	Greedy-data	0.0093	5.3667	0.0	0.3112	1.0684
0.1	No Migration	SLO-aware	0.0087	5.9523	0.0	0.3154	1.0630
0.1	Random	Greedy-local	1.2317	69.8047	384.0	0.5783	1.4500
0.1	Random	Greedy-data	0.7868	56.0006	388.4	0.4748	1.2800
0.1	Random	SLO-aware	0.0835	15.0189	345.9	0.3461	1.1689
0.1	Greedy	Greedy-local	1.7504	65.7991	254.6	0.6242	1.5632
0.1	Greedy	Greedy-data	0.0006	0.2587	204.2	0.3194	1.0763
0.1	Greedy	SLO-aware	0.0005	0.3752	232.6	0.2834	1.0426
0.1	State-aware	Greedy-local	1.4318	53.2727	7.4	0.5386	1.5652
0.1	State-aware	Greedy-data	0.0015	0.7404	7.4	0.3187	1.0765
0.1	**State-aware**	**SLO-aware**	0.0004	0.5055	8.3	0.2900	1.0482
0.2	No Migration	Greedy-local	1.5213	47.2753	0.0	0.6083	1.6715
0.2	No Migration	Greedy-data	0.0000	0.0000	0.0	0.3337	1.0918
0.2	No Migration	SLO-aware	0.0000	0.0000	0.0	0.3188	1.0754
0.2	Random	Greedy-local	0.7323	46.5192	339.1	0.5797	1.4774
0.2	Random	Greedy-data	0.5394	38.8161	390.4	0.5113	1.3215
0.2	Random	SLO-aware	0.0620	11.8700	307.5	0.4011	1.2429
0.2	Greedy	Greedy-local	1.1246	48.9581	206.2	0.6159	1.5650
0.2	Greedy	Greedy-data	0.0000	0.0000	256.3	0.3147	1.0715
0.2	Greedy	SLO-aware	0.0122	1.4703	225.9	0.2928	1.0497
0.2	State-aware	Greedy-local	1.0295	40.3417	7.4	0.5415	1.5803
0.2	State-aware	Greedy-data	0.0000	0.0000	7.9	0.3131	1.0699
0.2	**State-aware**	**SLO-aware**	0.0000	0.0000	7.5	0.2888	1.0473

to a single edge node, with functions assigned to edge nodes in a round-robin fashion (**W2**).

Policies. For comparison against our proposed policies, we consider the following baseline approaches for function offloading:

- **Greedy-local** (**GrL** for short): local execution is always preferred; if the local node runs out of memory, requests are offloaded to the closest cloud node;
- **Greedy-data** (**GrD**): a simplified variant of our state-aware approach, where we always schedule function execution so as to minimize the amount of state data that must be retrieved through the network.

For data migration, we consider the following baselines:

- **No Migration**: keys are never migrated;
- **Random**: randomly assign keys to nodes;
- **Greedy**: move each key to the node that ranks first in remote accesses to that key; to compute the ranking, we use a custom metric defined as the product of the data requests rate coming from the node and the estimated network latency between the node and the current key location.

State. We consider a total of 100 state keys in the system, and assume that each function accesses (i.e., reads or updates) up to 5 keys during execution. The set of keys accessed by each function is randomly sampled, and we consider two different scenarios where the probability of accessing a key follows, respectively, (i) the Zipf and (ii) the uniform distribution. In the latter scenario, the same key is rarely accessed by more than a single function, whereas few popular keys exist in the former. Each state access for every function occurs with a probability value uniformly sampled from $\{0.1, 0.25, 0.5, 0.75, 0.9, 1.0\}$. As regards the size of the data associated with each key, inspired by the observations reported in [21], we consider a bimodal distribution defined as a mixture of two Gamma random variables with shape $k = \mu 10^{-4}$ and scale $\theta = 10^4$, with $\mu \in \{10^4, 10^7\}$ bytes. At the beginning of each experiment, we assume all the key-value pairs to be stored in the data store node in the cloud.

Other parameters. We simulate system execution for one hour, replicating every experiment 10 times. Data migration policy is activated every 120 seconds. We consider different values for the maximum data access time requirement $T_f^{max} \in \{0.1, 0.2\}$s, assuming it to be identical for all the functions. The penalty cost unit associated with SLO violations is set as $\rho_f = 1$ \$/s.

5.2 Offloading and Data Migration Policies

Table 3 reports the simulation results for the first considered workload scenario (W1) and the Zipf distribution for key popularity. In this workload scenario, invocation requests are uniformly distributed to edge nodes, meaning that the keys needed by each function will be necessarily accessed from multiple locations. Considering the stricter SLO requirement of having data accessed within 100 ms (also depicted in Fig. 2), we observe that guaranteeing the

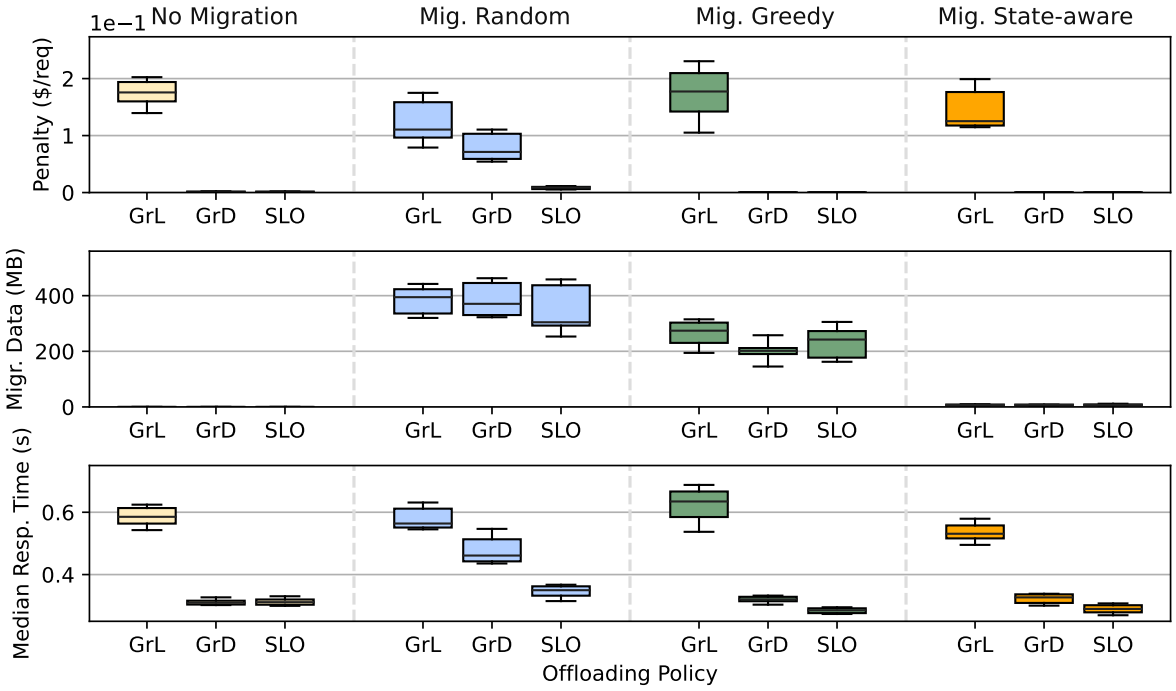


Figure 2: Results in the workload scenario W1, with $T_f^{max} = 0.1$s.

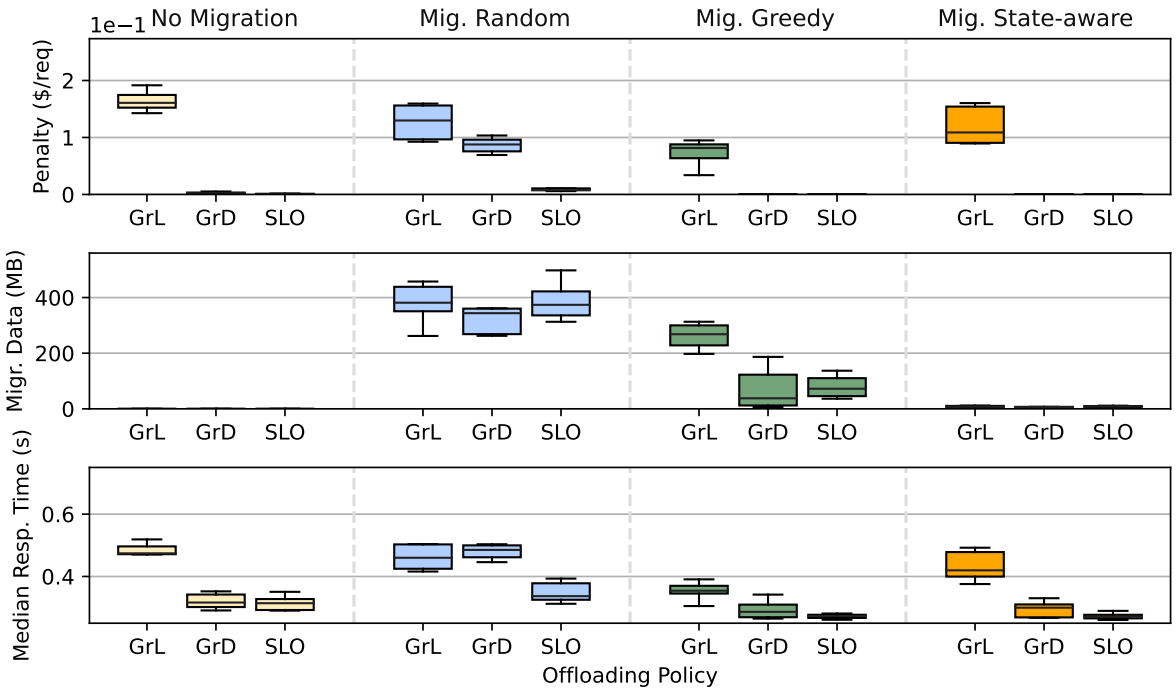


Figure 3: Results in the workload scenario W2, with $T_f^{max} = 0.1$s.

desired service level cannot be simply achieved using any set of policies. Specifically, the experiments demonstrate the importance of devising and adopting state-aware policies.

With no data migration policy enabled, it is particularly evident the impact of state-aware offloading. Indeed, the Greedy-local policy fails to meet the SLO for more than 50% of the served requests in this setting, demonstrating that simply ignoring data locality is not a successful approach in presence of stateful functions. State-aware policies (i.e., Greedy-data and SLO-aware) manage to significantly reduce the number of SLO violations, keeping them below 10% of the served requests.

To further improve the results, we need the ability to relocate data out of the centralized data store. For this purpose, we consider the baseline case of a random migration approach. However, random migrations—as expected—only worsen the situation for all the offloading policies. They also cause a significant amount of data to be migrated throughout the experiment (up to 400 MB in 75% of the cases).

Interestingly, the Greedy migration policy fails to improve the situation compared to the no-migration scenario when coupled with Greedy-local offloading. Conversely, when paired with state-aware offloading policies, SLO violations are less than 1% and the paid violation penalty significantly decreases as well, both for Greedy-data and SLO-aware. Compared to the Greedy migration approach, our State-aware approach achieves even better results. While it also keeps violations below 1%, it reduces the amount of migrated data from about 200 MB to less than 10 MB, avoiding the oscillations of the Greedy data migration policy.

Looking at the overall request response times, we can further observe that the SLO-aware offloading policy performs slightly better than Greedy-data, avoiding unnecessary execution offloading based on expected SLO satisfaction. Therefore, the combination of State-aware data migration and SLO-aware offloading (i.e., the two policies we propose in this paper) leads to the best results in this scenario.

When considering a less strict SLO requirement, i.e., $T_f^{max} = 200$ ms, all the policies experience fewer SLO violations and the impact of data migration decreases. Nevertheless, adopting a proper offloading policy is still fundamental to achieve acceptable performance, with Greedy-local largely failing to meet SLO requirements even in this scenario. State-aware offloading policies completely avoid paying SLO violation penalties in this scenario. The impact of data migration is still visible looking at overall response times. Compared to the baseline scenario with no migrations, the state-aware key placement reduces median response times by about 10%.

Similar results are observed in the alternative workload scenario W2, where invocation requests for each function are directed to a single edge node. As illustrated in Fig. 3, the different workload dynamics do not alter the relative performance of the policies we compare. It is only worth observing that the Greedy-local policy performs slightly better, especially in terms of overall response times. This is explained by the fact that having a limited number of functions to handle at each edge node favors container reuse and, hence, reduces memory contention for local execution. However, this policy still fails to meet SLO expectations in terms of data access time and causes the provider to pay a significant amount of penalties.

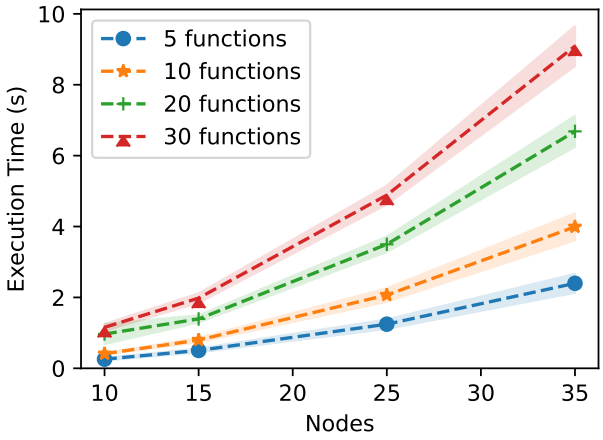


Figure 4: Execution time of the State-aware migration policy.

Due to space limitations, we do not show the results obtained with reduced key contention among functions (i.e., using uniform key popularity rather than Zipf), as they lead to the same considerations reported above.

Overall, our experiments demonstrate that the proposed State-aware data migration and SLO-aware offloading represent the best policies, allowing the provider to minimize the monetary penalties associated with SLO violations, while also leading to the lowest response times.

5.3 Scalability

Throughout the experimental campaign, we kept track of the computational overhead imposed by policy execution. As expected, the State-aware data placement policy, based on ILP resolution, is the most demanding one. Figure 4 shows the execution time of the policy varying the number of functions and computing nodes in the system, with the ILP resolved using IBM CPLEX® (version 12.6.2.0).

It can be observed that the computational overhead is lower than 2 s with 10 computing nodes and grows to less than 10 s with 35 nodes. The observed overhead looks acceptable, especially as the computation of data migrations can be performed asynchronously with respect to function execution. Nonetheless, we plan to explore heuristic resolution strategies in the future to further reduce the execution time of the policy.

6 RELATED WORK

The serverless paradigm requires platform providers to transparently manage infrastructure operations, possibly in face of (stringent) SLAs. We classify the existing research efforts into two categories, according to their main focus. A large set of papers considers functions and their scheduling on computing resources, whereby data strictly depends on functions; data dependencies are modeled at different degrees of granularity, including also implicit modeling. Stressing the importance of data, other works treat data as first-class

citizens, whereby it is assumed to be more convenient to execute or relocate functions whenever needed. In this case, the key topics are caching, replication, and consistency of data. In the following, we analyze the key results, aiming to highlight the core contribution of our work. To the best of our knowledge, the execution of stateful serverless functions in edge-cloud environments is still an open issue.

Function scheduling. Different solutions have been proposed so far to allocate stateful functions. Basically, four main approaches are pursued to model the dependency between functions and their data: make explicit the data intent of functions through APIs or annotations; co-locate functions with their data; optimize data passing among functions; and model data dependencies. Hence, different function scheduling policies are designed.

To make explicit the data dependencies of functions, developers enrich the metadata of functions with references to data (or bucket of data) required during execution (e.g., [20, 25, 27]). Lambdata [27] is the first approach that makes the intent of a function's input and output explicit for improved locality. Although the framework works in a cloud environment, such information is exploited to optimize functions execution, e.g., by running functions that read same data on the same worker. Similarly, to ensure that functions are scheduled close to their data, FaDO [25] provides a unified interface to the various storage services, and defines a special HTTP header to specify the required bucket on the function invocation requests. In this case, functions are allocated on the requested bucket, if possible. Data intent information is especially useful in geographically distributed environments. To determine a trade-off between data and computation movement in edge systems, Rausch et al. [20] propose a greedy multi-criteria policy that selects the best node for function scheduling by leveraging a scoring function with four contributions: (1) proximity to data storage nodes at edge locations; (2) proximity to the cloud-based container registry; (3) availability of specialized hardware; and (4) matching of user-annotated scheduling preferences between edge and cloud locality.

The second class of approaches resorts to co-location, thus considering data dependencies only implicitly. Co-location can be implemented either through replication or caching (e.g., [26]), or by running the function code on the storage node (e.g., [28, 32]). For example, Cloudburst [26] is a popular stateful serverless platform that co-locates compute and data. It employs an architecture with distributed storage and caching on machines hosting functions. In a preliminary work, Tiwary et al. [28] propose using WebAssembly to run serverless functions: exploiting lightweight virtualization, such functions can be more efficiently executed directly on data sources. The idea of pushing WebAssembly functions into storage has been also exploited by Shredder [32], which realizes a single-node multi-tenant cloud store that allows small units of computation to be performed directly within storage nodes. Hetzel et al. [9] suggest using an actor-based model, where each actor has its own locally stored state. Actors only interact by exchanging messages.

Serverless applications are usually composed of multiple functions that exchange intermediate (ephemeral) data. To this end, three general approaches emerged: leverage a shared (remote) storage (e.g., [6, 11, 15, 18]), exploit data locality to improve performance (e.g., [10, 12, 14, 24, 26–28]), and perform direct transfer (e.g., [14]). For example, SyncMesh [7] stores and processes data locally, so as to provide it to other nodes only on-demand. Conversely, Sonic [14] optimizes application performance and cost by selecting among different data-passing methods for each (namely, local storage, direct passing, remote storage). To this end, they model cost and time needed to exchange data and use a Viterbi algorithm, which guarantees to find the true maximum a-posteriori solution. Wukong [3] enhances the locality of DAG-based parallel workloads, by explicitly considering data that a function passes to downstream functions directly connected in the graph. Xu et al. [30] propose a more general approach, where functions can access data generated by upstream function (even if not directly connected). To rule the complexity of model and uncertainty of data volumes and delays, the authors resort to an online learning heuristic that aims to co-locate functions and their data on groups of edge resources. A slightly different perspective is presented in Zion [23], which considers data flowing from storage gateways to storage nodes and enables triggering functions on a data-driven basis (instead of an event-driven basis).

To conclude, a few works explicitly model the relationship between functions and data (e.g., [2, 19]). For example, Puliafito et al. [19] distinguish between stateless functions executed on serverless, which can access a remote cloud storage, and stateful containers, with a locally attached persistence volume. The scheduling problem, formulated as a mixed ILP (MILP), aims to find a trade-off between the cost of transferring input and output data using the network and a data access cost. First, stateful containers are allocated; then, stateless functions are placed so to balance load among nodes. The data model is rather simple, as it assumes that functions exchange a fixed amount of data with a centralized cloud storage system at each invocation. NEPTUNE [2] is a K3s-based platform for running latency-sensitive serverless applications on geo-distributed edge topologies. The authors formulate the application placement as a MILP, considering CPU and GPU requirements and minimizing the overall network delay. Currently, NEPTUNE only models network traffic exchanged between functions, whereas their access to external data stores is only partially modeled: the time to read from and write on a database is modeled as non-controllable stationary disturbance of the response time.

Data placement. The advent of serverless fostered the development of novel approaches to store data. A few works build persistent storage leveraging serverless, being cheaper than cloud computing resources (e.g., [4, 11, 29]). These works do not explicitly consider the presence of functions for computation, but provide convenient storage abstractions.

Other works focus on data and consider serverless functions as volatile, lightweight, and easily relocatable (e.g., [18, 31]). These works usually do not optimize the allocation of resources for functions, but mainly focus on data placement (e.g., [1]), replication (e.g., [16, 17]), and caching (e.g., [26]). Building on Cloudburst [26], Pheromone [31] proposes a data bucket abstraction that can be configured with triggers specifying *when* the target functions should be invoked and *how* their output should be passed to the next functions in a workflow. To improve data locality, Pheromone uses a two-layered scheduler, with a global coordinator that balances load across nodes and drives the execution of function across multiple

buckets, and a local scheduler that triggers functions execution on each node. Pfandzelter et al. [17] propose a data replication middleware that operates in a geographically distributed environments and with heterogeneous resources. Although the middleware can be used to optimistically replicate data on cloud and edge devices running serverless functions [16], currently it only blindly replicates key-groups on every replica node (thus introducing inefficiencies for writing operations). Conversely, the policy we propose helps to identity the locations where it is more convenient to move data.

Contribution. To the best of our knowledge, policies for SLA-aware execution of stateful functions in geo-distributed environments are missing. We consider serverless functions accessing external data stores, dealing with heterogeneous data access patterns and non-uniform data size for different keys. This model agrees with the evidences in [21].

The closest works to ours are [2, 19, 20]. As in [20], we consider an edge-to-cloud computing environment, where computing and storage nodes are interconnected with non-negligible network delay. Nevertheless, we do not require the user to manually indicate the data intent of functions. These access patterns can be estimated (or learned) at run-time. Moreover, we propose a state-aware data migration policy. Differently from [2, 19], we propose a more detailed formulation of the data dependency model, by explicitly considering the data access patterns of functions. We leverage this information to solve two main tasks, namely function offloading and data migration. Both tasks consider that functions expose requirements through a SLO on data access latency: hence, functions and data cannot be located on distant nodes, otherwise the serverless platform provider will pay penalties for SLA violation. To the best of our knowledge, this is the first contribution that considers a SLA-aware allocation of functions and data in edge-cloud environment.

Finally, we acknowledge that our approach is far from being fully-fledged. Indeed, we postpone to future work the integration of more sophisticated techniques for improving replication, fault-tolerance, and caching of data.

7 CONCLUSION

We investigated the problem of executing stateful serverless functions in edge-cloud environments, providing guarantees about the time required to access key-valued state. We proposed a set of policies to control both *function offloading* (i.e., when and where function execution should be offloaded to a remote node), and *data placement* (i.e., how to relocate state data to better accommodate the current workload). Simulated experiments demonstrate that our approach provides the desired SLO guarantees, while avoiding frequent data migrations and improving overall response times.

For future work, we plan to integrate our approach in an existing FaaS framework, and investigate efficient heuristics for the resolution of the data migration problem.

ACKNOWLEDGMENTS

This work has been partially supported by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di "campioni nazionali" di R&S (M4C2-19) - Next Generation EU (NGEU).

REFERENCES

[1] Koichiro Amemiya and Akihiro Nakao. 2020. Layer-Integrated Edge Distributed Data Store for Real-time and Stateful Services. In *Proc. of IEEE/IFIP NOMS '20*. 1–9.
[2] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. 2022. NEPTUNE: Network- and GPU-Aware Management of Serverless Functions at the Edge. In *Proc. of SEAMS'22*. ACM, 144–155.
[3] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, et al. 2020. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In *Proc. of ACM SoCC '20*. ACM, 1–15.
[4] Marcin Copik, Alexandru Calotoiu, Konstantin Taranov, and Torsten Hoefler. 2023. FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example. arXiv:2203.14859 [cs.DC]
[5] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, et al. 2022. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Trans. Softw. Eng.* 48, 10 (2022), 4152–4166.
[6] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, et al. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proc. of USENIX ATC '19*. USENIX Association, 475–488.
[7] Daniel Habenicht, Kevin Kreutz, Soeren Becker, Jonathan Bader, et al. 2022. SyncMesh: Improving Data Locality for Function-as-a-Service in Meshed Edge Networks. In *Proc. of EdgeSys '22*. ACM, 55–60.
[8] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, et al. 2018. Serverless Computing: One Step Forward, Two Steps Back. arXiv:1812.03651 [cs.DC]
[9] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. 2021. μActor: Stateful Serverless at the Edge. In *Proc. of MobileServerless'21*. ACM, 1–6.
[10] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proc. of ACM ASPLOS'21*. ACM, 152–166.
[11] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, et al. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proc. of OSDI'18*. USENIX Association, 427–444.
[12] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *Proc. of USENIX ATC '21*. USENIX Association, 805–820.
[13] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, et al. 2023. Serverless Computing: What It Is, and What It Is Not? *Commun. ACM* 66, 9 (2023), 80–92.
[14] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, et al. 2021. {SONIC}: Application-aware data passing for chained serverless applications. In *Proc. of USENIX ATC '21*. 285–301.
[15] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proc. of ACM SIGMOD '20*. ACM, 131–141.
[16] Tobias Pfandzelter and David Bermbach. 2023. Enoki: Stateful Distributed FaaS from Edge to Cloud.
[17] Tobias Pfandzelter, Nils Japke, Trever Schirmer, Jonathan Hasenburg, and David Bermbach. 2023. Managing data replication and distribution in the fog with FReD. *Softw. - Pract. Exp.* 53, 10 (2023), 1958–1981.
[18] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proc. of USENIX NSDI '19*. USENIX Association, 193–206.
[19] Carlo Puliafito, Claudio Cicconetti, Marco Conti, Enzo Mingozzi, and Andrea Passarella. 2023. Balancing local vs. remote state allocation for micro-services in the cloud–edge continuum. *Pervasive Mob. Comput.* 93 (2023), 101808.
[20] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. 2021. Optimized container scheduling for data-intensive serverless edge computing. *Future Gener. Comput. Syst.* 114 (2021), 259–271.
[21] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, et al. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proc. of ACM SoCC'21*. ACM, 122–137.
[22] Gabriele Russo Russo, Tiziana Mannucci, Valeria Cardellini, and Francesco Lo Presti. 2023. Serverledge: Decentralized Function-as-a-Service for the Edge-Cloud Continuum. In *Proc. of IEEE PerCom '23*. 131–140.
[23] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. 2017. Data-Driven Serverless Functions for Object Storage. In *Proc. of ACM/IFIP/USENIX Middleware '17*. ACM, 121–133.
[24] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proc. of USENIX ATC '20*. USENIX Association, 419–433.
[25] Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. 2022. FaDO: FaaS Functions and Data Orchestrator for Multiple

Serverless Edge-Cloud Clusters. In *Proc. of IEEE ICFEC '22*. 17–25.
[26] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, et al. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (2020), 2438–2452.
[27] Yang Tang and Junfeng Yang. 2020. Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit. In *Proc. of IEEE CLOUD '20*. 294–303.
[28] Mayank Tiwary, Pritish Mishra, Shashank Jain, and Deepak Puthal. 2020. Data Aware Web-Assembly Function Placement. In *Companion Proc. of WWW '20*. ACM, 4–5.
[29] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, et al. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *Proc. of USENIX FAST '20*. USENIX Association, 267–281.
[30] Zichuan Xu, Lizhen Zhou, Weifa Liang, Qiufen Xia, Wenzheng Xu, Wenhao Ren, Haozhe Ren, and Pan Zhou. 2023. Stateful Serverless Application Placement in MEC with Function and State Dependencies. *IEEE Trans. Comput.* (2023), 1–14.
[31] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *Proc. of USENIX NSDI '23*. USENIX Association, 1489–1504.
[32] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and Its State with Storage Functions. In *Proc. of ACM SoCC'19*. ACM, 1–12.

No Clash on Cache: Observations from a Multi-tenant Ecommerce Platform

Anna Lira
anna.lira@lsd.ufcg.edu.br
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil

Ruan Alves
ruan.alves@lsd.ufcg.edu.br
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil

Thiago Emmanuel Pereira
temmanuel@computacao.ufcg.edu.br
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil

Fabio Morais
fabio@computacao.ufcg.edu.br
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil

João Ramalho
joao.ramalho@lsd.ufcg.edu.br
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil

Mariana Mendes
mariana.mendes@vtex.com
VTEX
Rio de Janeiro, Rio de Janeiro, Brazil

ABSTRACT

Caching is a classic technique for improving system performance by reducing client-perceived latency and server load. However, cache management still needs to be improved and is even more difficult in multi-tenant systems. To shed light on these problems and discuss possible solutions, we performed a workload characterization of a multi-tenant cache operated by a large ecommerce platform. In this platform, each one of thousands of tenants operates independently. We found that the workload patterns of the tenants could be very different. Also, the characteristics of the tenants change over time. Based on these findings, we highlight strategies to improve the management of multi-tenant cache systems.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; • **Theory of computation** → **Caching and paging algorithms**; • **General and reference** → **Performance**.

KEYWORDS

Web, Cache, Workload, Multi-Tenant, Characterization

ACM Reference Format:
Anna Lira, Ruan Alves, Thiago Emmanuel Pereira, Fabio Morais, João Ramalho, and Mariana Mendes. 2024. No Clash on Cache: Observations from a Multi-tenant Ecommerce Platform. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3629526.3645039

1 INTRODUCTION

Caching is a classic that never dies. From the bottom layers of hardware CPU caches to the upper layers of web caches of cloud applications, caching reduces client-perceived latency and service load.

To get the most out of a cache, one needs to understand the characteristics of the workload submitted to the cache and configure it accordingly. Failing to match parameters such as cache capacity and eviction algorithms to the workload leads to direct impacts on the quality of service (usually observed in miss/hit ratio indicators) or resource waste (i.e., when the cache capacity is over-provisioned and additional capacity does not improve performance).

As an example, when configuring a web cache system, a typical starting point is to estimate the load level. The number of servers used in a large cache service depends on this load-level information since cache systems degrade when overloaded. Despite being useful, one needs more than load levels to define other cache parameters. For example, to determine cache capacity, it is necessary to consider the popularity of cached items. Item popularity matters because many requests sent to a cache might be related to a small number of cached items, thus reducing the need for more cache capacity. Also, caches typically show temporal locality [2]. Consequently, the cache needs to retain only the current working set.

Practitioners are well aware of the importance of considering these advises. However, one factor deviates practice from good practice: multi-tenancy.

To illustrate, consider multi-tenant ecommerce platforms, the case study of this paper. In these platforms, each tenant is an enterprise independent from the others; each tenant has its clients and products. There must be tenants that sell more than others. There must be tenants with large and small product inventories. There must be tenants with seasonal and sporadic selling patterns. However independent, the tenant's ecommerce sites run on shared resources and services (including caches) owned and managed by the platform. Considering that cache services are sensitive to load characteristics, as we described, and tenants are unlikely to have the same workload, what are the challenges a multi-tenant cache operator has to deal with?

To uncover these factors and highlight the challenges of multi-tenant caching, we collected and analyzed a trace of one of the cache services from VTEX, a large-scale ecommmerce platform[1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645039

[1] https://vtex.com

The observed cache service supports a few thousand tenants for a 10 hours observation period.

We observed that the cache load varies up to three times, reaching hundreds of thousands of requests per minute. Also, the load follows the same overall platform traffic trend (high traffic until late at night and lower traffic at dawn). However, we have a completely different picture when we analyze tenants in isolation. Some tenants exhibit stable load, others show a periodic load pattern, while some show peak periods (even at unexpected hours). Also, the load is highly concentrated among tenants: the 10% most loaded tenants account for almost 80% of the aggregated load.

The item popularity is also concentrated. Around 10% of items received about 50% of all requests. While there is a subset of very popular items, there are many items on the opposite extreme; 69% of the items are requested only one time (also known as one-hit wonders).

We also analyzed the degree of repeated access to the same items for each tenant. A high repetition degree indicates that the load is cache-friendly. This is because, the higher the repetition, the higher the chances of cache hits. Half of the tenants show less than 40% of repetition; a direct implication is that these tenants cannot have more than the 40% hit ratio, regardless of the cache configuration and capacity. Repetition also varies over time. While some tenants sustain low or high repetition during the observed period, others change their behavior as time passes.

The remainder of this paper is organized as follows. Section 2 describes the procedure for collecting data from the ecommerce platform in production. Section 3 describes the context and metrics applied in the workload characterization presented in Section 4. In addition, Section 5 describes the implications of the observed workload characteristics for cache management. Section 6 presents this research's findings and future work.

2 SYSTEM OVERVIEW AND INSTRUMENTATION

This section provides an overview of the Catalog system, the component of the platform from which we collected cache information (Section 2.1). We also describe the instrumentation procedure for data collection on production servers (Section 2.2). Finally, we explain the data used in the analysis and describe how data normalization is performed for anonymization purposes (Section 2.3).

2.1 Catalog Service

We collected data from a global business-to-consumer (B2C) and business-to-business (B2B) ecommerce provider. This provider offers tools and services to support ecommerce companies in creating and operating their online stores. This study focused on the Catalog system. The Catalog system is responsible for managing non-ephemeral product data (e.g., titles, descriptions, product identifiers, and enterprise identifiers) for all the tenants in the platform using a cache layer to improve their Quality of Service (QoS).

The Catalog system is composed of three layers: i) the database layer is responsible for the persistence of data and works as an index for the product information; ii) the logic and load balancing layer, responsible for providing access to product information and load balancing of requests; and iii) the cache layer is the service that temporarily stores product information for performance improvement. Figure 1 shows an overview of the system with the three layers and their relationships.

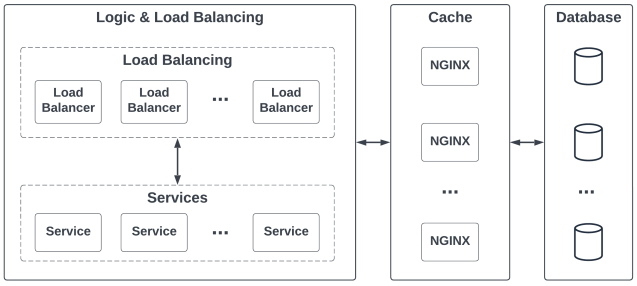


Figure 1: Overview of the architecture of ecommerce platform Catalog system. The cache layer stores database responses. The logic layer requests data from the database, using the cache layer as a proxy. In case of a MISS, requests are forwarded to the databases, and responses are cached. In the case of a HIT, a cached response is directly returned from the cache layer.

The Catalog services access the production information stored on the database through HTTP using a key-value format to structure the data. A key is generated by hashing the URI that uniquely identifies the request made to the database, and the value is the descriptive information for the related product. A cache layer, implemented as a cluster of NGINX [1] servers, is a proxy of the database to the service layer. The cache layer returns product information already cached by previous requests, thus reducing the load on the database system and reducing the latency perceived by the clients.

As the ecommerce system provides Catalog services for different online stores, the cache layer is shared by different tenants. Each request for product information is related to a specific product of a specific tenant in the same cache system. Thus, the cache stores data from products of all tenants' catalogs.

2.2 Data Collection

During our observation period, we activated a more verbose NGINX logging mode to collect the data. The Catalog operates within a high-traffic production environment, where misguided data instrumentation may cause disruption or system performance degradation. Therefore, we collected a 10% random sample of all requests in 5-minute intervals to avoid affecting the reliability of the Catalog. An agent running on the cache service continuously sends the log files to another server, to avoid fulfilling the NGINX servers' disks. The observation period ranged from 21:00 to 7:00 GMT-3, on December 13 and 14, 2022. After that, we stopped the observation and resumed the NGINX logging level to its default value.

To support our analysis, the data we collected includes the information associated with the products requested to the Catalog, cache status, and timing metadata. In summary, we considered the following information for each observed request:

- The request URI. Product and tenant identifications are encoded as parameters in the URI;

- Cache response status, which indicates when the cache answered a request as a MISS or HIT (and other NGINX possible statuses);
- The timestamp the request arrived at the cache layer;
- The identification of the NGINX node that received the requests;
- The request-response wait time.

2.3 About the Data

The collected workload has up to tens of millions of requests. These requests refer, approximately, to another tens of millions of products from approximately a few thousand online stores. Notably, each tenant exhibits unique and specific access patterns, which Section 4 will explore further.

To protect business-sensitive information, we anonymized the dataset. All the data discussed in the following sections was anonymized by applying the min-max normalization function. All the identifiers were anonymized using a hash function. The data normalization allows temporal comparison for all aggregated tenants, for one tenant, as much as comparisons among tenants. For example, Figure 2 presents the normalized number of incoming requests received per minute throughout the data collection period, illustrating the overall variations and trends in request rates.

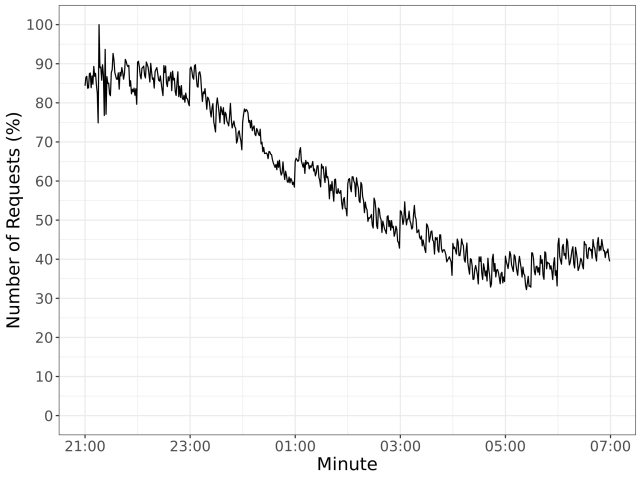


Figure 2: Normalized incoming request rates per minute for all aggregated tenants. It highlights variations in the request rate per minute and trends over the data collection period where peak demand occurs at night, with a typical decline in the dawn, followed by a resurgence of demand in the following hours of the morning.

Furthermore, for this work, we have focused on three key data fields crucial for understanding a multi-tenant cache system. These fields include request timestamp, tenant identifier, and product (or item) identifier.

3 BACKGROUND

In this section, we define the metrics we adopted to our characterization analyzed in Section 4. While these metrics were already described in the literature, some of them were not formally defined.

To evaluate the overall cache efficiency, we used the Hit Ratio. This metric is the proportion of data requests that were successfully met from the cache (referred to as Hit Status) relative to the total number of requests made. Consider two multisets: R, containing all data requests made during a specific period, and H, containing only those requests fulfilled by the cache within the same period ($H \subseteq R$). The Hit Ratio can be calculated using the following relation:

$$\text{Hit Ratio} = \frac{|H|}{|R|} \tag{1}$$

For example, consider a cache with a capacity of two items and a sequence of requests for items, R = {"A": "MISS", "B": "MISS", "A": "HIT", "A": "HIT", "C": "MISS", "B": "MISS", "A": "MISS"}, and a few hits, represented by H = {"A", "A"}. To calculate the hit ratio, we divide the number of hits by the total number of requests made, which in this case is 2/7 or 0.29. This means that 29% of the requests were successfully satisfied by the cache during the given time period.

To understand the demand, in addition to the number of requests submitted to the cache, we considered the **Footprint**. The Footprint metric measures the amount of data accessed within a specified time window by quantifying the number of distinct requested items in a given period [18]. This metric is essential for analyzing system capacity utilization, as it allows us to see how many items are requested in a given period. It also shares a close relationship with the Working Set Size (WSS) theory, which helps to understand an application's memory requirements. Considering the sequence of requests R and the set F (a subset of R) of distinct items requested, the Footprint is defined as follows:

$$\text{Footprint} = |F| \tag{2}$$

The **Item Repetition Ratio (IRR)** metrics, in its turn, gives an indication on how a workload would take advantage of the cache [9]. The IRR is an upper bound for the cache hit ratio for a determined workload. Given the multiset P of repeated requests for items in the period, the IRR is defined as follows:

$$IRR = \frac{|P|}{|R|} \tag{3}$$

For the previous example, P = {"A", "A", "B", "A"} and the IRR is 4/7 or 57.14%.

We also considered **Temporal Locality** in our characterization. Temporal locality refers to the tendency of the same item to be referenced within short intervals. It differs from concentration, which refers to the aggregate reference counts for items, regardless of the referencing order. Some metrics can be used to measure temporal locality, for example, the LRU stack-depth [12] and Inter-Reference Time [14]. In this work, we will focus on Inter-Reference

Table 1: Normalized statistical measures of the number of requests per minute. These statistics represent a substantial difference between the mean and median to the maximum value, suggesting a significant difference for the peaks.

Mean	Median	Max	Min	Std. Dev
0.41	0.37	1	0	0.20

Time [2]. This metric represents the time between references to the same item. We calculated the mean of the Inter-Reference Time (MIRT). MIRT sets itself apart from LRU stack-depth analysis by focusing on the timing of data access rather than the order within a stack. While LRU stack-depth provides insight into the sequence in which a specific number of requests occurred within a particular interval, MIRT's primary objective is to define and analyze these time intervals.

In addition to the metrics defined above, in the discussion presented in section 5, we adopted LRU as eviction policy. There is a huge literature on eviction policies trade-offs and there are many policies that can be better than LRU [7, 10, 11, 19, 20]. Notwithstanding, we adopted LRU because: 1) it was the policy used by the cache we observed (NGINX); and 2) some analytical methods (as the one used in section 5) are based on LRU.

4 WORKLOAD CHARACTERIZATION

This section discusses the characteristics of the load submitted to the cache of the Catalog system we observed. Section 4.1 focuses on observing the demand for requests the caching system receives. Next, section 4.2 explores the irregular popularity of the items stored in the system, identifying "hot" and infrequently accessed items. Section 4.3 highlights Footprint results, while section 4.4 presents IRR. Finally, section 4.5 evaluates the temporal locality of the data by analyzing the Inter-Reference Time.

4.1 Request Load

The initial analysis centers on the volume of requests entering the system over ten hours, as detailed in Table 1. The table provides normalized statistical metrics derived from the load. The findings reveal a mean of 0.41 and a median of 0.37, indicating that both metrics deviate significantly from the maximum value. This suggests a distribution pattern wherein a substantial proportion of observations exhibit relatively low activity levels compared to the peak request rate.

Regarding the temporal pattern of the request arrival, Figure 2 shows the number of requests per minute over time. The peak of requests occurs in the evening — after working hours — when it is common for demand for web services to be high. While there is a tendency for demand to fall at daybreak, demand for services rises again in the early hours of the morning [17].

The load is very concentrated in a small group of tenants: only 10% of tenants account for approximately 80% of all recorded requests. Figure 3 illustrates the cumulative request count for each tenant, ranked accordingly.

Tenants with a high number of requests can eventually worsen the performance of tenants with a low number of requests. This

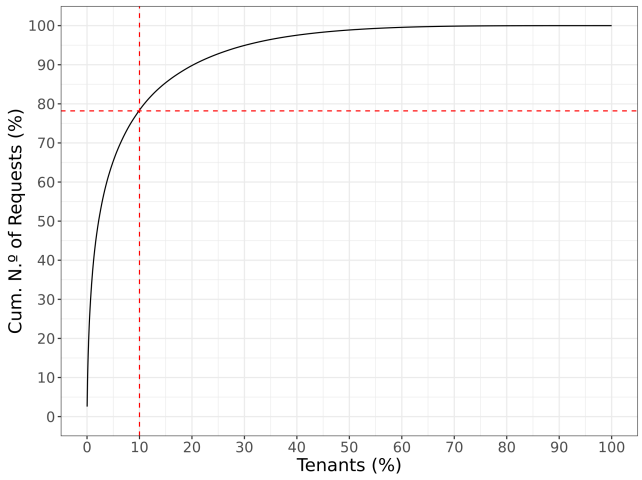


Figure 3: The cumulative sum of received requests, ranked by descending number of requests by a tenant. A small number of tenants are responsible for a large number of requests.

interference could happen when a burst of requests for new data from tenants evicts other tenants' popular data, so future requests that can reuse this data will result in cache misses. Thus, tenants with a high number of requests can quickly evict items and occupy a big slice of the cache. This highlights the importance of the promotion of fairness and mitigation of interference between tenants in shared caches [3, 6, 13, 16].

The request load for each tenant also shows temporal variability. This introduces an extra layer of complexity in resource allocation. Figure 4 illustrates the variation in the number of requests over time for a sample of the biggest tenants. Some tenants exhibit peaks in requests, indicating periods of significantly higher demand than their usual levels. This may result from seasonal events, special promotions, or product launches. Conversely, some tenants maintain relatively constant activity over time, with no significant fluctuations in request quantity. This suggests a more stable and predictable traffic profile associated with a regular customer base. Furthermore, some tenants may display intermittent request patterns, alternating between intense activity periods and relatively calm moments. This oscillation may be influenced by peak shopping times or specific marketing actions.

These behaviors may indicate that system management could be more dynamic and sensitive to tenants' needs since the traditional static resource allocation models can lead to provisioning problems. A dynamic resource adjusts resource allocation based on the current workload, trying to ensure that resources are neither wasted nor insufficient. In this scenario, tenants with request peaks may require more substantial cache allocations during these periods of high demand. Conversely, tenants with stable request patterns may benefit from conservative allocations.

In addition to the dynamic allocation of resources, the presence of tenants with high demand at certain times also reinforces the adoption of strategies to promote fairness and avoid inter-tenant interference. This can reduce the effects caused by higher demand in lower-demand customers.

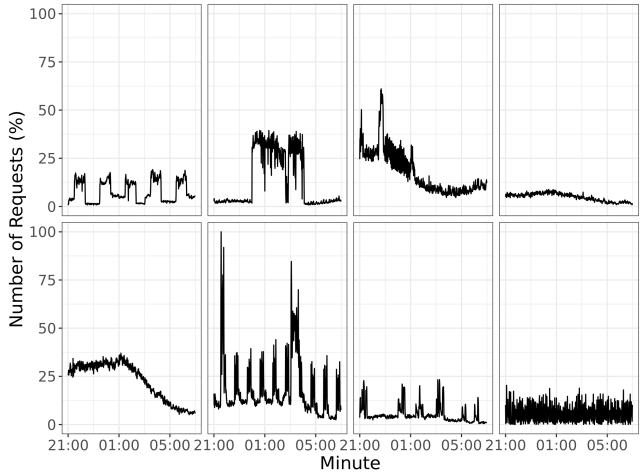


Figure 4: Request number variation over time for a sample of the biggest tenants. Different access patterns emerge, with some tenants experiencing request peaks, others displaying fluctuations, and some exhibiting intermittent patterns.

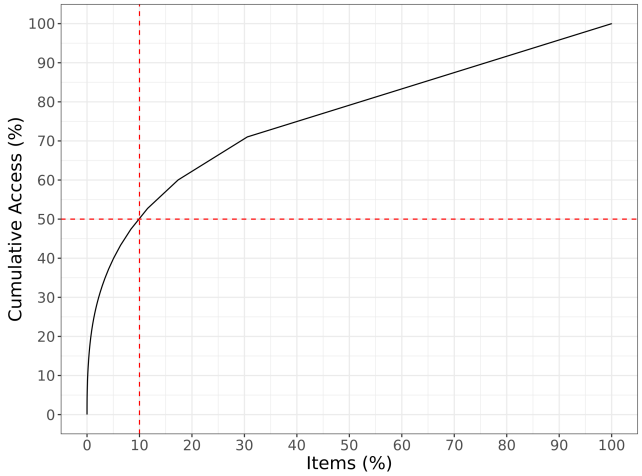


Figure 5: The cumulative sum of accesses per item, ranked by decreasing the number of accesses. The curve experiences rapid growth for a set of items, commonly called "hot" items. A small number of items are accessed more frequently, while the remaining are accessed sporadically. This pattern suggests distinct popularity dynamics among the items.

Table 2: Frequency type distribution. There is a huge fraction of the items that is requests only once.

Item Frequency Type	Percentage
One Request	69%
Two Requests	13%
More Than Two Requests	18%

4.2 Concentration Of Access

Looking for requested items and evaluating the frequency of access, we can see a discrepancy in popularity. Some can be classified as "hot" items with a high access volume. These items are characterized by their great popularity and constant demand. In contrast, other items are rarely accessed and, in some cases, not accessed.

Figure 5 shows the non-uniform pattern in item referencing behavior. Around 50% of all requests are directed at only 10% of the distinct items in the system. This phenomenon suggests the existence of a subset of items that are highly requested, while other items show considerably less repetition. Braun and Claffy describe this concentration phenomenon as a common characteristic of Web traffic [4].

Also, as observed in Table 2, the portion of items that are accessed only once (one-hit wonders) is very large. Items with no repeated access could not take advantage of the cache. However, simply ignoring an item in its first request can be a wrong choice because this item can be re-accessed in the near future, not being a one-hit wonder. Furthermore, the same table shows the percentage of two-hit wonders, which must be understood if is a good choice to store items like them too, since they will be reused only once. Therefore, it is necessary to seek strategies that reduce the number of n-hit wonders according to the particular cache objectives. Many policies seek how to avoid storing unpopular items [5, 7, 8, 10, 20].

4.3 Footprint

As mentioned in previous sections, the Footprint value depends on the size of the window range chosen. For our trace, with a time window of 10 hours, the Footprint is half the total of requests made (tens of millions of requests). When we look at the Footprint per tenant, we see a discrepancy in the distribution of Footprint values. Figure 6 illustrates the distribution of Footprints among tenants, highlighting a concentration of Footprint for a small portion of tenants. In summary, the 10% of tenants represent a portion of around 75% of the total Footprint, suggesting that these entities may have distinct usage patterns, potentially utilizing the system more intensively or requiring more resources than the remaining majority. On the other hand, the remaining 90% of tenants contribute only a quarter of the total Footprint.

Additionally, the tenant's Footprint varies over time, influenced by their different workload patterns, such as changes in the number of orders and access. As mentioned, variations in the observed time window can lead to changes in the Footprint value. Figure 7 further illustrates this dynamic, showing the variations in Footprint for the six largest tenants. This emphasizes that a tenant's Footprint can change over time. Observing the Footprint in time windows can help us understand the storage space requirements of each tenant in the system over a given period.

4.4 Item Repetition Ratio

The data, with a time window of 10 hours, indicates variation in IRR values among different tenants, suggesting that some tenants are more adept at utilizing caching benefits, while others may not exhibit behaviors favorable to caching. Figure 8 shows many tenants have low IRR. For example, around 50% of tenants have IRR less than 40%. With low repetition, these tenants could be candidates for

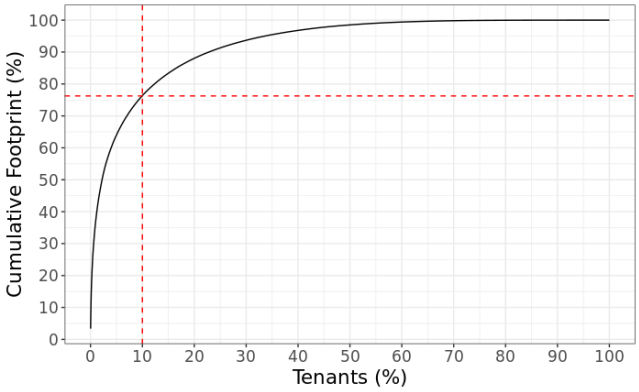


Figure 6: Cumulative distribution of Footprint. This illustrates the concentration of Footprint in a small portion of tenants, around 75% of the total Footprint corresponds to 10% of the tenants, so a small number of tenants occupy a large slice of cached storage.

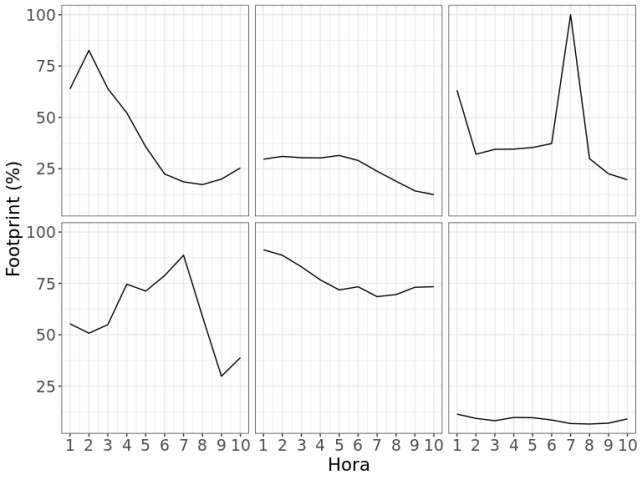


Figure 7: Footprint over time for a selected sample of the biggest tenants by number of requests. This highlights the variability in a tenant's usage of resources throughout different periods of the day. Some tenants exhibit peaks and valleys over time, while others consistently maintain either a high or low usage Footprint.

not having items in the cache since their requests in a time window are for different items, not contributing to the cache hit.

By decreasing the window size for adjacent one-hour windows, we observe some IRR behaviors over time, as shown in Figure 9. Some tenants show high IRR with slight variation, others low IRR with little variation, and others show considerable variation in IRR between windows. In addition to the tenants with a slightly variable IRR, we also observed some that showed a large temporal oscillation. This dynamic behavior emphasizes the importance of evaluating the adequacy of the cache at specific time intervals. This approach makes it possible to discern tenants whose caching behavior may

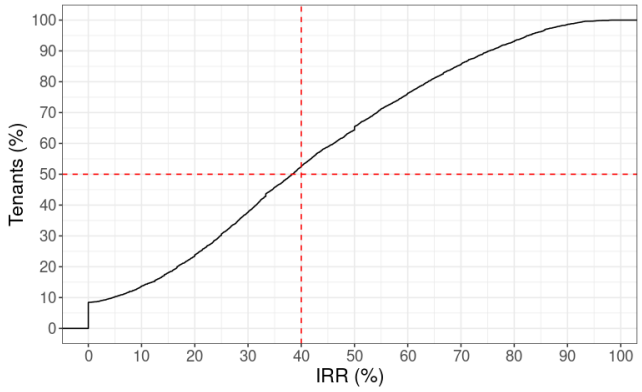


Figure 8: Cumulative distribution of Item Repetition Ratio. Some tenants are better at reusing the cached data, taking advantage of the cache usage. While others may not demonstrate cache-friendly behavior, occupying memory and worsening performance.

vary over time, thus requiring adaptive cache management strategies for optimal performance. In this case, the system should ideally have a mechanism for calculating IRR periodically that plays a central role in fine-tuning cache capacity by selecting which tenant should or should not be in the cache at a given time.

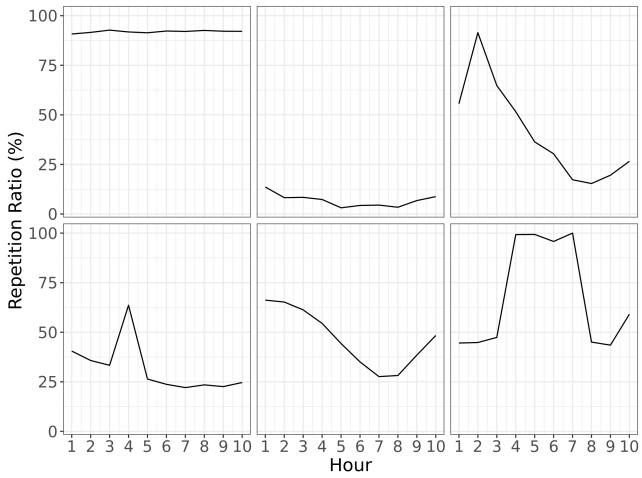


Figure 9: IRR over time for a selected sample of the biggest tenants by number of requests. This illustrates diverse patterns and variations in a tenant's cache-friendliness.

Before entering the cache system, analyzing the tenants' IRR could be a strategy for allowing them in or not, section 5 will describe this approach. For example, if the tenant's IRR is high and has little variation, it is a good candidate for cache entry. On the other hand, if the IRR is low and with little variation, it is a candidate for not entering the cache at any time.

Table 3: MIRT statistical measures for the items.

Mean	Median	Max	Min
0.26	0.19	1	0

4.5 Temporal Locality

The Inter-Reference Time can help us to understand the tendency of reference to the items. Figure 10 exhibits the MIRT of the items in the workload. As explained in Section 3, the MIRT is calculated by finding the mean of the time intervals between references to the same item. It is worth noting that items referenced only once lack Inter-Reference Time values, making it impossible to calculate the mean.

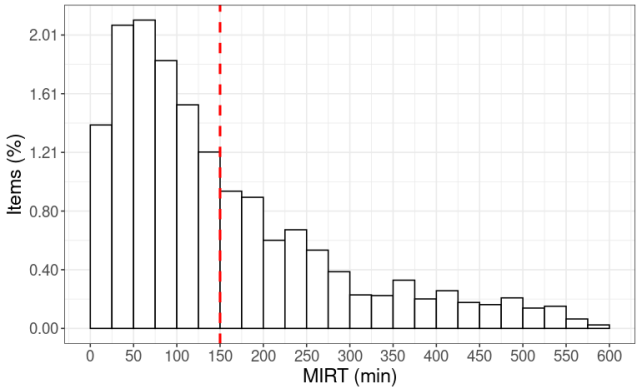


Figure 10: Percentage of items by Mean Inter-Reference Time ignoring items referenced only once. Most have MIRT lower or equal to 150 minutes.

A significant portion of items have a MIRT lower or equal to 150 minutes, specifically 57.20% of them (including items referenced once). Table 3 also reinforces this by showing that the measures of central tendency (mean and median) are low. This is an indication of good cache usage by the items. On the other hand, 42.80% of the items have MIRT longer than 150 minutes. This indicates that many items are idle during specific periods, taking up space in the cache.

Figure 11 shows the dynamic trend in item access patterns for ten adjacent one-hour time windows, where each box corresponds to an analyzed hour. The frequency of access to items changes over different periods, indicating a dynamicity in the access patterns over time. The first few hours have more items with low MIRT, where the number of access is high, and the average number of hits is concentrated mainly between 10 and 15 minutes. After that, the number of accesses decreases and becomes increasingly dispersed, with a reduction in items with low MIRT.

In general, we observe that the pattern of temporal locality changes over time. This shows that tenants' and items' cache requests are variable over time. This further reinforces the need for dynamic management of the cache.

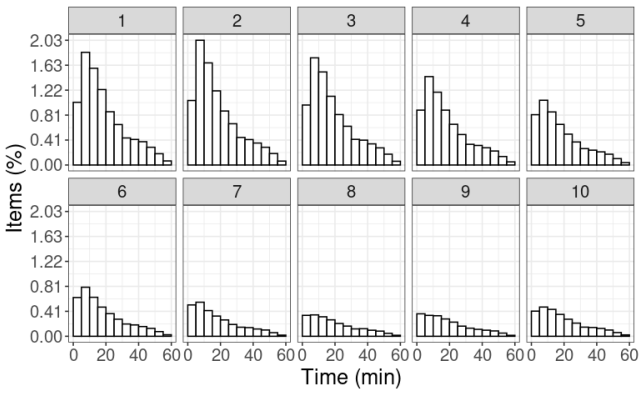


Figure 11: Percentage of items by Mean Inter-Reference Time for each one-hour window of items that are referenced more than once. Showing the dynamic tendency in item access patterns.

5 PERFORMANCE IMPLICATIONS AND ISSUES

In this section, we discuss the possible consequences of the characteristics observed in our multi-tenant trace. In particular, we focus on adapting cache management strategies to be aware of the tenants.

Unpopular items: A Case for Exclusion from Cache. Items with low repetition may not justify their inclusion in the cache. Allocating cache resources to items with sporadic access may lead to sub-optimal resource utilization. In this case, cache policies and systems that consider admission control like Lazy Adaptive CacheReplacement [10], CacheSack [21], and others [5, 7, 8, 11, 20].

One interesting possibility to explore is to consider data about the tenants as hints to the admission control algorithm. For example, would be useful to have more aggressive admission control policies for tenants that have a low item popularity profile?

Item Repetition Ratio as a Filter for Tenants. As section 4 indicated, some tenants are not cache-friendly (because they have low IRR). In addition to the **item** admission control, one can consider a **tenant** admission control (using IRR as an indicator). To provide a short overview of this idea, we evaluated the impact of removing some tenants from our multi-tenant trace. We used a trace-drive simulation that gives the overall hit ratio based on a given trace capacity. Considering an optimal capacity to attend all the tenants found on the trace as the baseline, we simulated scenarios removing the tenants that have IRR below three different thresholds (20%, 30%, and 50%). As Figure 12 shows, when tenants with 20% or fewer IRR values were excluded, the optimal cache capacity was reduced to 90% of the original capacity, resulting in an optimal hit ratio of 61%. Similarly, imposing a stricter criterion of 30% IRR led to an optimal capacity of 78% of the original, leading to a higher hit rate of 64%. Excluding tenants with IRR values of 50% or less further improved the efficiency of the cache, producing an optimal capacity of 49% of the original capacity and a commendable hit rate of 71%. Obviously, other trade-offs need to be analyzed, including

the corresponding impact on the latency and server load caused by the removal of tenants. In any case, the impact on resource usage could be important, as indicated by this analysis.

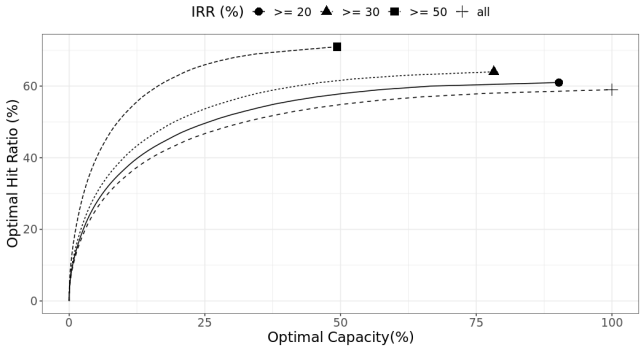


Figure 12: Optimal hit ratio based on optimal capacity for each one of workloads. Each line is one type of workload (i.e. the line IRR >= 20 means that only tenants with IRR above or equal to 20% are on the workload). This suggests that some tenants with low IRR can be removed to reduce the capacity while maintaining the same level of performance or increasing it.

Cache partition. As mentioned in our characterization, a few big tenants concentrate the most on cache usage. We also show that the load of the tenants varies over time. It is not clear the impact a spike in the load of a big tenant can cause on the QoS of the other tenants (in particular, if the capacity of the cache is not too large). One possible to avoid the interference caused by tenants sharing the same cache is to create and enforce partitions on the shared cache to host the tenants [3, 6, 13, 15, 16].

6 CONCLUSIONS

This paper has provided an analysis of the workload characteristics in a multi-tenant cache environment using data from a prominent web cache service in a large-scale ecommerce platform. The findings revealed a significant imbalance in request distribution among tenants, with approximately 80% of requests directed towards 10% of them. Moreover, we observed distinct access behavior patterns, ranging from steady loads to periodic spikes in request volumes.

Furthermore, the study highlighted that many requests concentrated on a small number of items (hot items). On the other hand, a majority part of the items experienced sporadic access. Additionally, it was evident that some tenants demonstrated low cache-friendliness, resulting in a cache hit ratio decrease due to infrequent item retrievals.

The cache-friendliness is mainly related to tenants' request patterns and product inventory. Significant request patterns were identified in tenants' workloads: Some tenants are more searched than others, and this implies a more significant number of requests to the system; can present seasonal request patterns, others sporadic patterns; can present peaks and/or valleys in request patterns, or can present stable patterns over time. In addition, there are tenants with an extensive number of products, which can mean that they need more storage space than others. In addition, we found that access patterns vary over time, emphasizing the dynamic nature of cache requests. This underlines the importance of dynamic cache management to adapt to changing access patterns and optimize the use of resources.

As the next steps in this industry collaboration, we plan to investigate how can we take into consideration tenants' profiles into cache management policies (admission control, eviction, and partition). Also, it is necessary to consider the mechanisms to adapt to changes in the behavior of the tenants. Although some of these issues are discussed in the literature, in particular for hyperscalers [21], there are still open questions on how to support regular web caches.

7 ACKNOWLEDGMENTS

This work has been funded by MCTIC/CNPq-FAPESQ/PB (EDITAL Nº 010/2021) and by VTEX BRASIL (EMBRAPII PCEE1911.0140).

REFERENCES

[1] Nginx documentation. https://nginx.org/en/docs/. Accessed: 2023-11-05.
[2] Arlitt, M. F., and Williamson, C. L. Internet web servers: workload characterization and performance implications. *IEEE/ACM Trans. Netw. 5*, 5 (1997), 631–645.
[3] Berger, D. S., Berg, B., Zhu, T., Sen, S., and Harchol-Balter, M. Robinhood: Tail latency aware caching - dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 195–212.
[4] Braun, H.-W., and Claffy, K. C. Web traffic characterization: an assessment of the impact of caching documents from ncsa's web server. *Computer Networks and ISDN systems 28*, 1-2 (1995), 37–51.
[5] Chai, Y., Du, Z., Qin, X., and Bader, D. A. WEC: improving durability of SSD cache drives by caching write-efficient data. *IEEE Trans. Computers 64*, 11 (2015), 3304–3316.
[6] Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017* (2017), D. D. Silva and B. Ford, Eds., USENIX Association, pp. 321–334.
[7] Einziger, G., Friedman, R., and Manes, B. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage 13*, 4 (2017), 35:1–35:31.
[8] Eisenman, A., Cidon, A., Pergament, E., Haimovich, O., Stutsman, R., Alizadeh, M., and Katti, S. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 65–78.
[9] Gu, R., Li, S., Dai, H., Wang, H., Luo, Y., Fan, B., Basat, R. B., Wang, K., Song, Z., Chen, S., Wang, B., Huang, Y., and Chen, G. Adaptive online cache capacity optimization via lightweight working set size estimation at scale. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023* (2023), J. Lawall and D. Williams, Eds., USENIX Association, pp. 467–484.
[10] Huang, S., Wei, Q., Feng, D., Chen, J., and Chen, C. Improving flash-based disk cache with lazy adaptive replacement. *ACM Trans. Storage 12*, 2 (2016), 8:1–8:24.
[11] Megiddo, N., and Modha, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA* (2003), J. Chase, Ed., USENIX.
[12] Ponciano, L., Andrade, N., and Brasileiro, F. V. Bittorrent traffic from a caching perspective. *J. Braz. Comput. Soc. 19*, 4 (2013), 475–491.
[13] Pu, Q., Li, H., Zaharia, M., Ghodsi, A., and Stoica, I. Fairride: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016* (2016), K. J. Argyraki and R. Isaacs, Eds., USENIX Association, pp. 393–406.
[14] Sediyono, A. Dynamic average of inter-reference time as a metric of web cache replacement policy. In *Proceeding of International Conference on Rural Information and Communication Technology* (2009).
[15] Seyri, A., Pan, A., and Vamanan, B. Dynamically sharing memory between memcached tenants using tingo. In *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2019, Companion Volume, Orlando, FL, USA, December 9-12, 2019* (2019), ACM, pp. 40–42.
[16] Suh, G. E., Rudolph, L., and Devadas, S. Dynamic partitioning of shared cache memory. *J. Supercomput. 28*, 1 (2004), 7–26.

[17] Vallamsetty, U., Kant, K., and Mohapatra, P. Characterization of e-commerce traffic. *Electronic Commerce Research 3*, 1 (2003), 167–192.

[18] Xiang, X., Ding, C., Luo, H., and Bao, B. HOTL: a higher order theory of locality. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013* (2013), V. Sarkar and R. Bodík, Eds., ACM, pp. 343–356.

[19] Yang, J., Qiu, Z., Zhang, Y., Yue, Y., and Rashmi, K. V. FIFO can be better than LRU: the power of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023* (2023), M. Schwarzkopf, A. Baumann, and N. Crooks, Eds., ACM, pp. 70–79.

[20] Yang, J., Zhang, Y., Qiu, Z., Yue, Y., and Vinayak, R. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 130–149.

[21] Yang, T., Pollen, S., Uysal, M., Merchant, A., Wolfmeister, H., and Khalid, J. Cachesack: Theory and experience of google's admission optimization for datacenter flash caches. *ACM Trans. Storage 19*, 2 (2023), 13:1–13:24.

MemSaver: Enabling an All-in-memory Switch Experience for Many Apps in a Smartphone

Prajwal Challa
vxc5208@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

Baohua Song
21cnbao@gmail.com
Coredump Limited
Auckland, New Zealand

Song Jiang
song.jiang@uta.edu
University of Texas, Arlington
Arlington, Texas, USA

ABSTRACT

The availability of diverse applications (apps) and the need to use many apps simultaneously have propelled users to constantly switch between apps in smartphones. For an instantaneous switch, these apps are often expected to stay in the memory. However, when a user opens more apps and memory pressure increases, Android kills background apps to relieve the memory pressure. When the user switches a killed app back to the foreground, the user experiences a laggy response that compromises his experience. To delay this killing under memory pressure for a smoother user experience, we propose *MemSaver*, a low-cost approach for preemptively swapping selected pages of the background apps out of memory to avoid or postpone the killing of apps while ensuring their near-ideal switch time. MemSaver uses pages accessed during events similar to the switch and about the same app context for predicting the pages to be accessed in the next switch. Evaluations on OnePlus 9 Pro using representative apps show that up to 60% of app's memory (RSS) can be saved while maintaining the switch time within the acceptable range.

CCS CONCEPTS

• **Software and its engineering → Memory management**; • **Human-centered computing → Ubiquitous and mobile computing systems and tools**.

KEYWORDS

Android, Memory reclamation, App hot launch

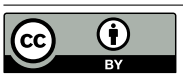
ACM Reference Format:
Prajwal Challa, Baohua Song, and Song Jiang. 2024. MemSaver: Enabling an All-in-memory Switch Experience for Many Apps in a Smartphone. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3629526.3645050

1 INTRODUCTION

The explosive advancement of mobile technology has carved itself into the daily lives of people. Combining ever-increasing computational power along with a wide variety of feature-rich mobile applications (apps), use of smartphones have become an integral part in one's daily life. To accommodate their diverse needs, studies have shown that smartphone users usually run ten or more apps daily [7], often requiring a large memory capacity for a smooth user experience. However, smartphone manufacturers often have to limit the amount of DRAM due to a trade-off with affordability and battery capacity. This leads to high memory pressure when the user wants to keep more apps alive in memory.

Mobile OSes, like Android OS, follow the foot steps of traditional desktop/server OSes and support virtual memory, which is an approach that allows multiple processes to run concurrently even with limited physical memory via page swapping. The swapping strategy searches the space occupied by any of the in-memory processes for pages that are less likely to be accessed and swap them out to the secondary storage to make more free memory available. However, this widely-accepted practice of memory usage control becomes highly undesirable in the smartphone environment. In a smartphone, there is only one foreground app at a time that is actively interacting with the user and the remaining apps run in the background. Performance of the foreground app is of the highest priority as it directly determines user experience. In mobile devices, the swapping strategy, despite its benefits, may potentially bring unacceptable performance issues for the foreground app. As some of its pages may be selected for swapping due to lack of recent accesses, a shift of working set to access these pages again will result in a surge of page faults and turbulent user experience. Even worse, using the swapping approach for tackling high memory pressure could lead to memory page thrashing and render the mobile device inoperative. Disrupting use experience of the app the user is actively interacting with should be avoided at any cost and be used only as a last line of defence under extremely high memory shortage.

To this end, Android chooses to relieve high memory pressure by first killing background apps using its Low Memory Killer Daemon (*lkmd*) [12]. To protect the foreground app from being subject to swapping, lkmd kills the least essential app(s) to free memory as a response to high memory pressure. Killing apps is carried out in a selected manner where apps are assigned different priorities based on their execution state. For example, an app that contains Android Activities that are still visible to the user or runs background services is of higher priority than apps that are not visible to the user (background state). It is noted that the killing usually does not affect the app's functionalities. When an Android app correctly implements its Activity component for saving its current state, the state will be saved before its killing. When the user switches back to the app, its Activity will restore all its visible state as if it remained in the background.



This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645050

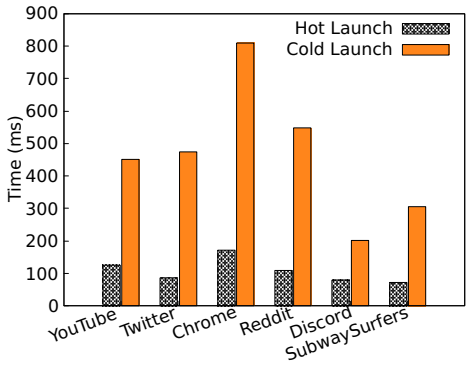


Figure 1: Hot and cold launch latencies for various apps in OnePlus 9Pro.

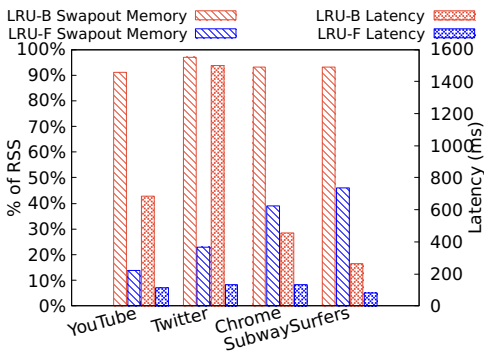


Figure 2: Amount of memory swap-out and hot launch latency with different LRU strategies for various apps, including LRU-background (denoted "LRU-B") and LRU-foreground (denoted "LRU-F").

As users interact only with the app displayed on the screen and usually keep many apps in the background, frequent switches between the apps are expected. Studies have found that users switch between apps over 100 times a day [2]. With such a high switch frequency, switch time is highly impactful on users' perceived smartphone service quality. An app may be switched from its background state to the foreground state when it still runs in the background. This switch is named *hot launch*. In contrast, a switch of an app to the foreground after it has been killed is named *cold launch*. To understand the impact of lkmd on the switch time (which turns a hot launch into a cold launch), we select some popular apps and experimentally compare their hot and cold launch latencies in an Android smartphone (see Section 4 for details). These apps include entertainment app (YouTube), social media apps (Twitter, Reddit, Discord), utility app (Chrome), and gaming apps (Subway Surfers). The results are shown in Figure 1. The hot launch latency of each of the apps is much lower than its cold launch latency. As shown, the hot launch latency is around 100 ms. Studies have suggested that when an event's response time is less than 100 ms people feel that the event is instantaneous [1]. And a response time less than 150ms does not compromise user satisfaction [14]. A time that is significantly higher than the 150 ms latency indicates a laggy response. For example, cold launch latencies of most of the apps are over 400 ms. Repeated cold-launch experience due to the aggressive lkmd's action to reclaim memory is annoying, though the effort is currently deemed necessary and often unavoidable.

While it is known that system-wide page swapping takes the risk of compromising foreground app's user experience, selected killing of background apps leads to laggy smartphone use experience. In this paper, we propose a solution, named *MemSaver*, that reclaims memory pages to ease memory pressure without killing apps or compromising the foreground-app's user experience. It carries out page swapping for background apps with minimal impacts on their hot launch latency. To this end, there are some significant challenges to address, including how to predict pages currently in the memory space occupied by a background app, or its RSS (Resident Set Size), that will be (or equivalently, will not be) accessed in its next hot launch, how to collect history access information for the prediction without disruption of foreground app's execution, how to store the information in a light-weight way, and how to strike a trade-off between saving memory and keeping launch latency low.

To this end, we make a number of contributions in the paper to address the challenges.

- We show that the commonly-used LRU-like history access information becomes much less relevant for prediction of pages to be used in the next hot launch.
- For the first time we found that pages accessed in the history hot launches of a context are highly predictive of those to be accessed in the upcoming hot launch of the same context.
- We developed a series of low-cost approaches to collect and record the relevant history accesses.
- We implemented Memsaver into Android and extensively evaluate its performance on six commonly-used apps. The results show that application's RSS can be reduced by up to 60% while keeping the hot0launch latency in an acceptable range.

2 RELEVANT HISTORY FOR PREDICTION

In the swapping of a background app, we need to predict its pages that are likely to be accessed in its next hot launch and only keep them in the memory. Obviously, we have to make the prediction based on access history. A common wisdom is to look into the recent accesses and identify pages that have been recently and frequently accessed - the LRU strategy used in Linux (the Android's kernel).

2.1 Recency-based History

The LRU policy is based on recent access history, or recency-based history. To understand the impact of using recent access in a page swapping strategy on an Android app, we design two controlled experiments. When an app is in the background, a time is chosen as its *swap moment* when its selected pages are swapped out of the memory to the flash. In this study, the intended swap moment is usually well before the available memory is to be exhausted to keep enough idle memory in the system always available. This is important for smooth user experience because the foreground app

or newly started apps may need substantial amount of free memory at any time to quickly expand its memory allocation.

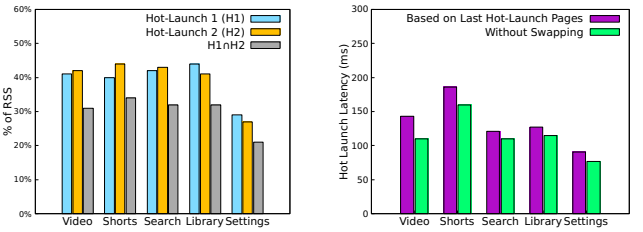
The two experiments differ at how the "recently" in the LRU policy is defined. In the first experiment named *LRU-background*, only the time period from when the app has completed its transition to the background and before the swap moment is considered as "recent". In the second experiment named *LRU-foreground*, the time period starts when the app is still in its previous foreground (10 seconds before a switch to the background). That is, it covers recent accesses in the last foreground execution. In each of the experiments, only pages that are accessed during its defined recent time period are kept in the memory in the swapping. This is a simulation of LRU's behavior.

We are interested in knowing (1) how many pages can be swapped and (2) how longer the next hot launch latency will become after a swapping. The desired result is that significant memory can be saved without substantial increase of the latency. Figure 2 shows percentage of the app's RSS that can be reclaimed and the corresponding hot launch latency for various apps. We observe that in *LRU-background* very few pages are accessed and most of the RSS pages (over 90%) can be swapped. However, this large memory saving comes with an unacceptably high hot launch latency. For example, for YouTube the latency is increased by over 6X over its 120ms ideal latency (with no swapping). In contrast, *LRU-foreground* has little increase over the latency. However, it saves much less memory (*LRU-background* can swap 2.1X-6.5X as many pages as *LRU-foreground* from the memory). However, the experiments suggest that making the choice by adjusting the recency does not lead to a solution with both goals (large memory swapout and low hot launch latency) well achieved. The key to a success relies on the accurate prediction of pages to be accessed in the upcoming hot launch.

2.2 Event-based History

While the recency-based history is not well indicative of pages required in the next hot launch (aka hot-launch pages), we need to turn to more relevant history. Similar to process scheduling, an app's hot launch involves a fixed set of operations and data accesses to re-establish its previous execution state. Accordingly, the set of pages in different hot launches are likely to bear some similarity, and provide a clue on which pages should be retained in the memory. As a hot launch is a user-triggered event, we are defining and exploiting a locality based on the same type of events in the history.

To observe whether such similarity exists across the hot launches in an app's execution, we examine and compare the pages accessed in consecutive hot launches of YouTube. To assess the potentially maximal similarity, we consider an app's activity context in the investigation. An app usually has a number of predefined activity contexts, such as video, shorts, search, and library in YouTube. A hot launch resumes its foreground execution in the same context as the one it stayed in immediately before its switch to the background. Resuming the execution in different contexts may require different context-specific pages. Therefore, in this experiment we only conduct the comparison between hot launches of the same context.



(a) Consecutive hot launch page's overlap

(b) Hot launch latency with and without swapping

Figure 3: Overlap of hot launch pages between two consecutive hot launches (H1 and H2) with the same context as well as impact of using the last hot-launch pages for swapping on the hot launch latency.

Figure 3a shows the size of hot-launch page set as a percentage of its RSS for each of the two consecutive hot launches of a specific context. The RSS represents the amount of memory held by the app in the background. The figure also presents size of the intersection of the two sets. As shown, the two sets contain about 80% of common pages, suggesting that accesses in a recent hot-launch event are predictive of that in the next hot launch. The hot launch pages are almost all in the memory before a swapping. This is the reason why a background app has a satisfactory hot launch latency if no swapping was conducted. If we only keep the last hot launch pages in the memory (about 30-45% of the RSS), about 55-70% of the memory pages can be swapped. The potential memory saving is significant. Meanwhile, Figure 3b shows next hot launch latency with or without swapping based on the event-based history. The time increase is about 10-30 ms, which is moderate and stays in the acceptable range.

2.3 Challenges of Exploiting History

While we reveal the more relevant locality in the event-based history, there are a number of challenges on translating this finding into an online system design. First, it can be too expensive to online detect pages accessed during a hot launch. A straightforward approach for the detection requires two scans of the app's entire page table: in the initial scan reset reference bits in the PTEs (Page Table Entry) and then check them in the following scan when the launch is completed. Each of the full-table scans can be very expensive. The cost is especially problematic for the first one, as it takes place when the user initiates the app's hot launch. It would essentially block user's interaction during the scan.

The second challenge is on the cost of recording the detected pages. Following the conventional wisdom, it is tempting to consider introducing a new data structure to accurately record the pages. However, this approach would carry significant time and space overheads. In particular, the data structure for tracking all accessed pages would increase the memory usage, which is in conflict with the goal of this work for reducing memory footprint. In addition, it may be necessary to record multiple sets of pages, each about a different history event/context, which further inflates the costs.

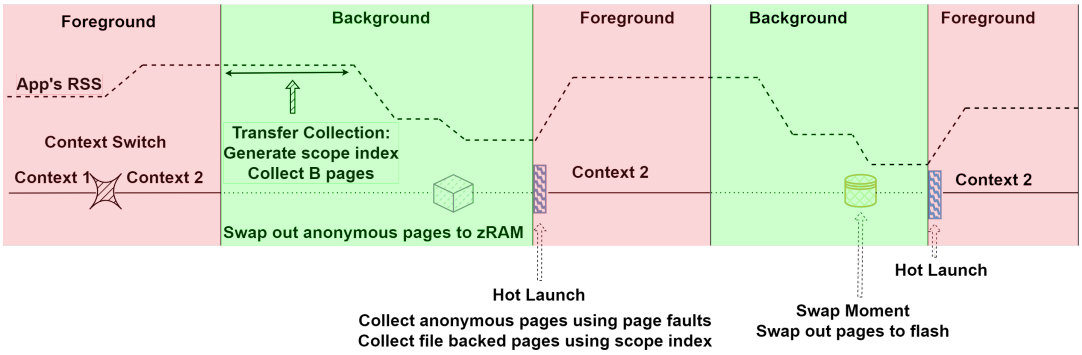


Figure 4: Representative events in an app's life time.

Third, assuming that a light-weight approach for detecting and storing hot-launch pages is available, we may have multiple hot-launch page sets recorded during an app's execution, each for one recent history hot launch. These hot launch events are of different recency, and may be associated with different contexts. It is unknown which of the page sets should be used in the decision of page swapping for more memory saving and lower launch latency. For example, in a situation where a hot launch of the same context hasn't been recorded, or the most recent hot launch is about a different context, it is not straightforward to understand the impact of using the sub-ideal history data and to make the best page swapping decision. The design of MemSaver addresses the challenges.

3 THE MEMSAVER DESIGN

An ideal design should have a light-weight approach to accurately detect history pages, along with an efficient way to store the pages. Such approach should also ensure that history information is utilized in a manner such that app's RSS is effectively reduced and does not lead to undesirable latency in the next hot launch. In this section we present MemSaver, an efficient design for identifying and recording pages that are likely to be accessed in an app's upcoming hot launch and selectively retaining them in memory and swapping the remaining pages to the flash. We dive into various design choices and take a close look at their corresponding performance implications. As MemSaver is deployed in Android, the design is tailored to the smartphone's hardware and software designs.

3.1 Phases in an Android App's Execution

It is necessary to know the phases and relevant events experienced during an app's execution that are relevant to the MemSaver's design. As illustrated in Figure 4, the app constantly alternates between the foreground state (the red zone in the figure) and background state (the green zone). In the red zone, any substantial overhead added by the external facility is likely to be felt by users and compromises their use experience. In contrast, a green zone allows such overhead without users' notice. Therefore, all MemSaver's operations are carefully carried out in the green zones.

As the swapping takes place at a moment during the background execution, we need to understand the memory usage during the time period to select the moment. In Android, after an app is sent to the background, within the first few seconds (about 5 seconds)

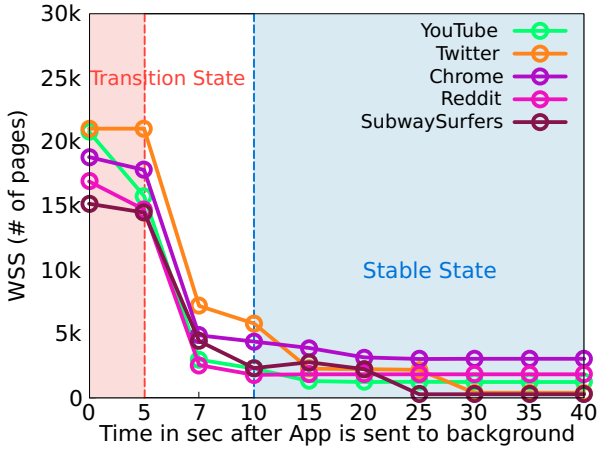


Figure 5: Background App's WSS reduction

we observe that the app's resident memory space (RSS) remains about the same as that of the app during its foreground execution, as shown in Figure 4. After this it reduces to a smaller size by releasing its allocated space and then stays at the memory footprint. We name these two time periods in the background as the transition and stable states, respectively. In the two states the size of actively accessed memory space (aka working set size or WSS) is very different. Figure 5 shows the change in the WSS of various apps over time after the app is sent to the background. During the transition state, the Android app is in the *onPause()* life-cycle activity state, a state where Android OS expects the app likely to go back to *onResume()* activity state (re-launch back to the foreground) soon. Consequently, an app's RSS is observed to remain the same as that of in the foreground to continue its normal execution in the background for a short time period in anticipation of a possible quick switch back. After this period WSS is dramatically reduced and stabilized as the app continues its execution into the stable state. This unique WSS/RSS behaviour of Android apps is one of the necessary elements the design of MemSaver will take into account. Integrating Android specific functionality along with careful design enables MemSaver to efficiently detect, collect, and use hot launch pages.

3.2 Detection and Storage of Hot-Launch Pages

To allow access prediction based on prior hot-launch events, MemSaver needs to collect pages accessed during a hot launch (hot launch pages). As a hot launch needs to be associated with an activity context, MemSaver integrates itself with Android OS to identify the current context when user interacts with the app in the foreground. As only the last context before a switch to the background will resume its execution in the next hot launch, MemSaver appends the last context's name before the app goes into the background to the next hot launch. That is, the context of a set of history hot launch pages is known when they are used for deciding swappable pages.

When MemSaver is informed by Android of an app's hot launch, MemSaver starts to collect accessed pages during the launch. As we have mentioned, the two full-scan approach requires a traversal of an app's entire address space, whose cost is unacceptable. As an example, for YouTube it represents a coverage of a 17GB address space, much larger than its RSS at the time of the hot launch (about 400MB). Accordingly, the time spent on one scan is about 100ms, which is too expensive.

In an app's address space, there are two types of pages: anonymous and file-backed pages. Since they have different properties, we design different light-weight approaches for access detection. As anonymous pages are more scattered in a larger space, it's not effective to identify a much reduced scope for efficient scanning. Instead, MemSaver artificially generates minor page faults to reveal what pages are accessed in a hot launch. Specifically, we leverage a Linux facility – zRAM – that is enabled in Android by default. zRAM is an in-memory compression-based swap space. When the app is in the background, MemSaver swaps out all its anonymous pages to the zRAM using *madvise()* syscall with the *MADV_PAGEOUT* flag. At this time the hot launch has not yet started. The swapout cost during the background execution is not felt by the user

During the hot launch, any accessed page will trigger a page fault for loading it from the zRAM. Consequently, MemSaver can identify the hot-launch pages by intercepting page faults during their handling in Linux. As page faults in zRAM do not involve any flash I/O operations, and only decompression of anonymous pages instead, the cost is moderate. Compared to that of true page faults from the flash, this cost is close to that of minor page faults.

The approach of using zRAM for generating page faults cannot be applied to the file backed pages, as these pages would only be swapped to the file system on the flash. To address the issue, we choose to narrow the scan scope. This is made possible by the observation that file-backed pages (1) are often clustered and (2) are mostly also accessed in the foreground phase. The idea is to collect the VMAs (the data structure the Linux kernel uses to manage contiguous virtual memory areas) covering file backed pages that have been accessed in the foreground. This set of VMAs will be the focused scope where MemSaver will look for accessed file-backed pages in the next hot launch.

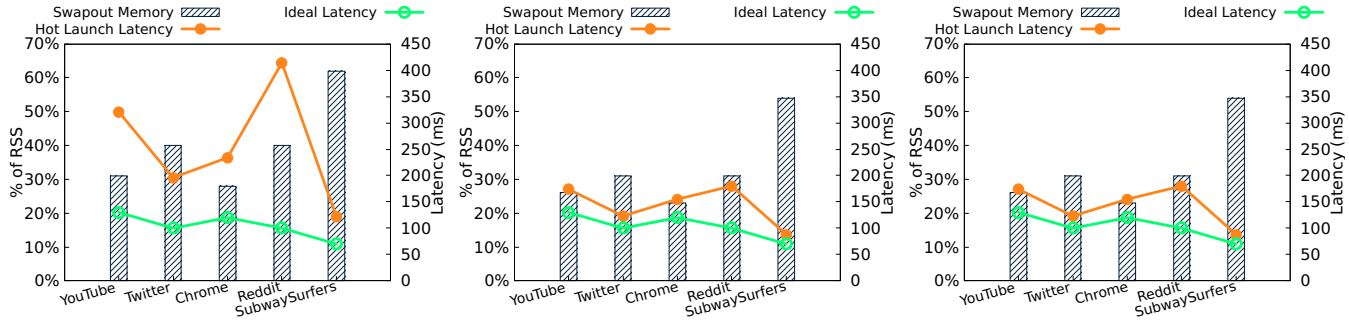
However, detection of the VMAs for accessed file-backed pages during the foreground execution is intrusive to the foreground app's execution. To address this issue, we leverage the unique WSS behaviour of Android apps, which is the existence of the transition state. The brief pseudo background period is in the green zone but retains foreground's access behavior. Therefore, MemSaver moves its detection of the focused scope from the foreground to this special background period. As this operation, named *transfer collection*, is in the green zone, it is affordable to use two scans to identify the VMAs that have been accessed. The VMAs will then be the scope for the two-scan operation in the next hot launch for file-backed pages. To further reduce the scope we use the 2MB aligned memory region in a VMA as the unit (representing 512 4KB-pages) for tracking file-back accesses. All detected VMA regions are recorded in an index (the scope index) for quick access. Within this focused page-scan scope, the two scans in the reduced scope takes only about 1 ms. This time overhead becomes acceptable even in the red zone.

Instead of using a data structure to store the detected hot launch pages, MemSaver records them in the unused PTE bits (bits 60-63) in the page table. At the time when an accessed page is detected, its PTE is likely to be the CPU cache. The cost for updating the PTEs is negligible. To determine pages for swapping, MemSaver does need to scan the page table to know the recorded hot launch pages. However, this will take place only when the app is in the background. Furthermore, using the idle PTE bits avoids inflating memory usage. The availability of four bits for recording accesses in the history presents MemSaver with choices of tracking multiple selected events. As a page swapping policy, MemSaver needs to know the specific events whose page accesses should be collected and which of the history events should be involved in a swapping decision making. To this end, we adopt a heuristic approach whose design is driven by targeted experiments (to be discussed).

3.3 Incorporating Working Sets in the Recent History

We have suggested retaining history hot launch pages in memory during the swapping. In the meantime, there are two sets of actively accessed pages (working sets) that may need to be kept in memory. One of them is that right before the app switches to the background. A hot launch is essentially a resumption of the last foreground execution. Pages in the working set at the end of the foreground phase is likely to be accessed at the beginning of the next foreground phase (i.e., the hot launch time period). The other working set is the one during the stable state in the background. Its pages need to be kept in memory even after the swapping to support its background execution. One challenge is that the detection of pages in the foreground working set would take place in the red zone in Figure 4. Its cost is simply not acceptable during the foreground execution. Fortunately, the existence of the transition state right after the switch to the background makes a non-intrusive detection of the pages possible. As it is in the background, we can use the two-scan approach to get its working set. Because the app maintains its foreground activities in the transition state, the working set right after the switch can be an approximation of the one before the switch. We denote this background working set B_1. In contrast, the working set in the stable state is denoted B_2.

MemSaver initiates its swapping during the background execution after the transition state. To understand the impact of additionally keeping B_1 and/or B_2 on the swapping efficacy in terms of swap-out memory amount and hot launch latency, we experiment with keeping different combinations of the working sets along with



(a) Keep hot launch pages H_1 and Working set B_2 in memory

(b) Keep hot launch pages H_1 and Working sets B_1 and B_2 in memory

(c) Keep hot launch pages H_1 and Working set B_1 in memory

Figure 6: Impact of Background Active pages on RSS reduction and Hot launch latency

the set of pages accessed in the last hot launch in memory (denoted as H_1). Note that we assume the app stays in the same context across the events. Apparently, it is necessary to keep B_2 in memory, as pages in B_2 continue to be accessed after the swapping (during the background execution period). The question is whether B_1 also needs to be considered. In the first experiment, we leave out B_1. Figure 6a shows the amount of swap-out memory (as a percentage of its RSS at the time of the swapping) and the corresponding hot launch latency after swapping any pages in the RSS that are not in either H_1 or B_2 for various apps. As seen, though an app's RSS can be effectively reduced (up to 60%), the hot launch latency is much higher than the ideal hot launch latency (the one without swapping). As it is necessary to maintain an acceptable hot launch latency, it is not sufficient to only consider B_2 in the swapping decision.

In an attempt to reduce the hot launch latency, we add the B_1 to the set of pages that are kept in memory (i.e., B_1, B_2, and H_1) in the swapping decision. Figure 6b shows its swapout memory amount and the corresponding launch latency. As expected, the amount of swapout memory is reduced. However, the reductions represent small percentages of respective RSSes. In the meantime, the hot launch latencies become close to their respective ideal ones. To know if it is still necessary to consider B_2 after B_1 is included, we then remove B_2 (i.e., only B_1 and H_1 pages are not swapped out). Figure 6c shows that both the memory swap-out amount and the hot launch latency do not have any substantial changes. Examining background active pages, we observe that some of the B_1 pages continue to be active over the execution of the app in the background and consequently cover about 95% of B_2 pages. Figure 7 shows the overlap of B_1 and B_2 for various apps. For this reason, regarding B_1 and B_2, B_1 MemSaver only collects B_1 pages of an app and retains these pages in memory (if they have been in memory before the swapping).

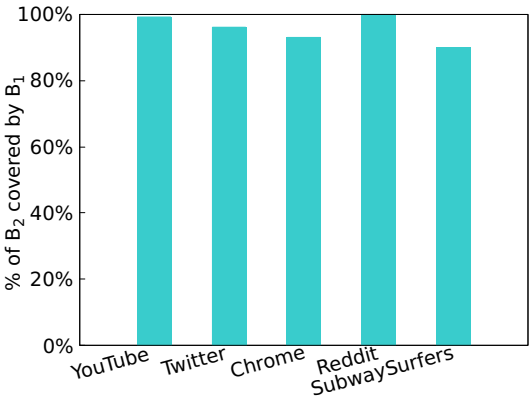


Figure 7: Overlap between B_1 and B_2

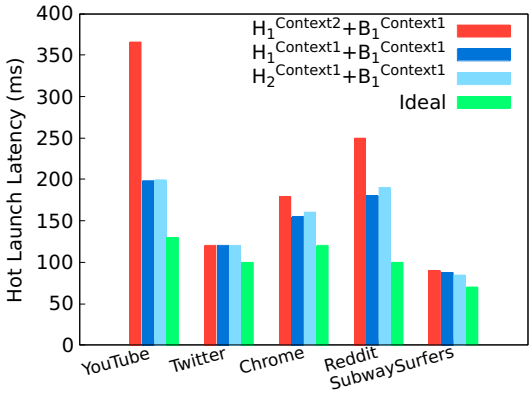


Figure 8: Hot launch latency when different history hot launch page set is used for the swapping before the hot launch.

3.4 Incorporating Context-aware Hot-launch Pages

As discussed, the set of pages that have been accessed during the prior hot launch is a potentially strong indicator of pages to be used in the upcoming hot launch. Multiple such prior page sets may have been recorded, and they may be associated with different app contexts. Intuitively, the most relevant one among all possibly recorded page sets is the one about the most recent hot launch and of the same context as the one for the upcoming hot launch. To

Table 1: Apps and Contexts

App	Context1	Context2
YouTube	Video	Shorts
Twitter	Search	Tweet
Chrome	Web page	Tabs
Reddit	Message	Search
Subway Surfers	Game Pause	Shop

have a sensible design, we need to experimentally confirm or disconfirm this conjecture and the impact of different sets of history hot launches. In the interest of clarity, the set of pages accessed in a history hot launch (H) is denoted H_k^{cxt}, where cxt is the context when the hot launch take places, k indicates how recently the launch is. For $k = 1$, it is the most recent one, or the last one, and for $k = 2$, it is the second to the last one, and so on. Figure 8 shows the hot launch latency when different history access information was used for a swapping in the background before the hot launch for various apps. The latency is compared with the ideal latency, the one when swapping is not conducted. In the experiment, two contexts were selected for each of the apps (Youtube, Twitter, Chrome, Reddit, and SubwaySurfers), namely *context1* and *context2*. Table 1 depicts the contexts for each of the apps. In the experiments, *context1* is designated as the current background context, which is also the one for the upcoming hot launch. *Context2* is a context appearing in the past. In a swapping, we keep pages in the most recent background working set B_1 (with *context1*) and pages in an H set. As shown in the figure, if the history H is $H_1^{context1}$ (the assumed most relevant H), or pages in the $H_1^{context1} UB_1^{context1}$ are retained in memory, the next hot launch latencies stay in the acceptable range (about less than 200ms). Admittedly, some of them are substantially higher than the ideal one (e.g., Reddit). In practice, smartphone use experience is more impacted by the unexpectedly long latency than by latency variations within an acceptable range.

While an H is associated with a context, we need to understand how strongly it is correlated with its context. This is important because sometimes a history H with *context1* (the current context) is not available. In the situation, MemSaver may have to choose an H whose context is different from *context1* for its swapping decision. Figure 8 also shows the hot launch latency after retaining pages in $H_1^{context2} UB_1^{context1}$ in memory during the swapping. Using an H with an unmatched context (*context2*), the hot launch latency can increase to an unacceptable level (370ms for YouTube). The latency for Reddit also increases to over 250ms. These drastic increases do not take place for all the apps (e.g., the increases for Twitter and SubwaySurf are moderate.). To be a reliable design, memSaver chooses to be conservative by not carrying out swapping if an H with a matched context has not yet been recorded.

A more common situation is that a history H with a matched context does exist but it is not the most recent one. To understand the impact of the recency on the swapping effectiveness, we experiment with the case where pages in $H_2^{context1} UB_1^{context1}$ are retained in memory. Figure 8 also shows its hot launch latency. As shown, using $H_2^{context1}$ produces the hot launch latency almost the same as that using $H_1^{context1}$. This observation suggests that

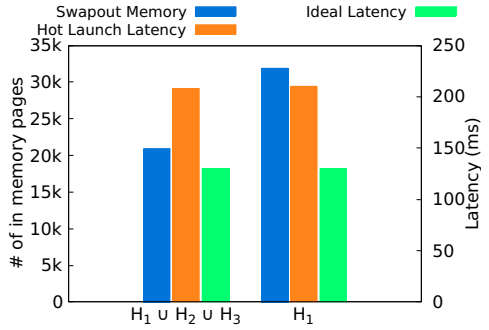


Figure 9: Comparison of swapout memory and hot launch latency for YouTube between using its three recent Hs and using only one recent H. All Hs are associated with the same context ('video').

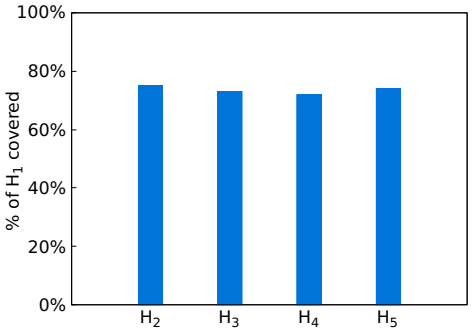
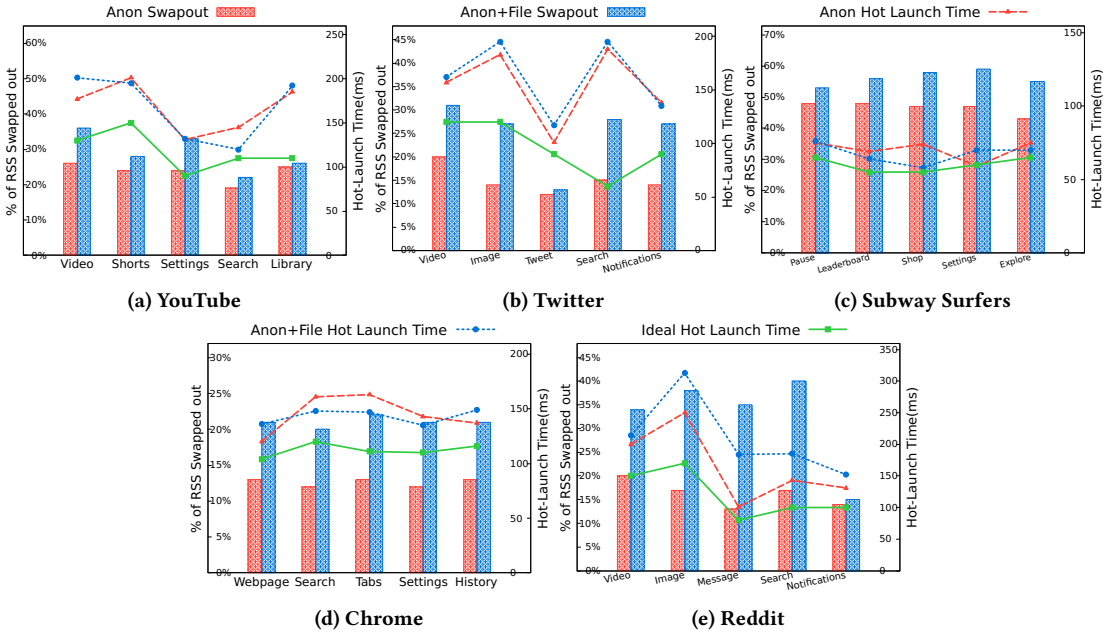


Figure 10: Overlap of earlier H with the most recent H (H_1 for YouTube. All Hs are associated with the same context ('video').

recency, or time-defined locality, is less relevant for identifying useful access history. This is consistent with the rationale of adopting event-based history in this study. Therefore, MemSaver chooses to use the latest H with the matched context (if available), whether it is for the latest hot launch or not, in its recorded history for swapping out the app's memory.

The last design question is on the choice of H when there are multiple Hs with the matched context in the recorded history. One might expect that using all the 'qualified' Hs would help to keep more pages to be used in the next hot launch in memory and thus further reduce the hot launch latency. To understand its impact, we carried out swapping for YouTube with 'video' as *context1* (the current context) and using either one or three history Hs of the same *context1*. Figure 9 shows their respective hot launch latency and number of swapout pages. Using additional Hs doesn't help to further reduce the hot launch time. It is not necessary to consider more than one (earlier) Hs. Furthermore, using multiple Hs has a side effort. As shown, using three Hs reduces the number of swapout pages by about 32%. Figure 10 shows overlap of each of the four earlier Hs with the H_1 (all Hs are associated with the same context). The overlap is consistently about 75%. This suggests that this 75% subset represents an invariant that persists over the different hot launches. It is this invariant that contributes to



(a) YouTube (b) Twitter (c) Subway Surfers

(d) Chrome (e) Reddit

Figure 11: RSS reductions and hot launch times with different swapping methods for apps at different contexts.

MemSaver's effective prediction. The remaining 25% subsets in the respective Hs are not essential for the prediction. This corroborates the finding shown in Figure 8 about using a non-most-recent H. As long as it is an H of the same context, the H can effectively serve the purpose. For this reason, as an optimization MemSaver doesn't have to collect and record H for every hot launch.

3.5 A Summary of MemSaver Policy

MemSaver consists of two operations: access collection and page swapping. There are four bits for each page in an app's page table entry for recording history, each for one history event. Because of limited number of bits and cost of access collection, MemSaver follows this collection policy.

(1) One bit is reserved for the B_1 pages. B_1 is collected with each switch to the background (during the transition phase);
(2) The remaining three bits are used for up to three Hs. They are logically organized as an LRU stack. The most recently recorded H is at the stack top. The one at the bottom is to be replaced by recording of a new H;
(3) For a new hot launch with *context1*:
  (a) If there hasn't been an H of *context1* recorded in the stack, collect its accessed pages and place the corresponding H at the stack top (by replacing the one at the bottom);
  (b) Otherwise, an H of *context1* exits in the stack. In the case, if it's in the stack bottom, collect a new H to replace it. Otherwise, skip the collection for this hot launch. This is for three reasons. (1) A history H of the same context is still usable; (2) Making fewer collections helps improve efficiency. And (3) an H earlier than two other Hs of different contexts is updated for freshness.

In the meantime, MemSaver follows this swapping policy.

(1) If there are not any H of the matching context, do not swap.
(2) Otherwise, use the H of the matching context and B_1 to decide the pages that will be retained in the memory and swap out other in-memory pages.

4 ADDITIONAL EVALUATION RESULTS

To understand the efficacy of MemSaver, we implemented and evaluated it in Android using various representative apps (YouTube, Twitter, Subway Surfer, Chrome, and Reddit). We compared its hot launch latency after an app's RSS has been reduced against the hot launch latency of all-in-memory apps (the ideal case). The evaluations were performed on a OnePlus 9 Pro phone with Qualcomm® Snapdragon™ 888 processor containing 12GB RAM and running Android version 11.

As mentioned, MemSaver has two objectives for a background app, which are to reduce its memory footprint and keep its hot launch latency within an acceptable range. While MemSaver uses different methods for detecting hot-launch pages (page faults from zRAM for anonymous pages vs. page table scans in a reduced scope for file-backed pages), we break down their impacts in the experiments. For each app, we assume availability of $H_1^{context1}$ with various current contexts (*context1*). Figure 11 shows the RSS reduction in percentage and the corresponding hot launch latency when swapping is applied only on anonymous pages or on both anonymous and file-backed pages. It also shows the ideal hot launch latency.

There are some interesting observations. First, RSS can be substantially reduced. With only anonymous pages are considered, up to 48% of the memory can be saved with an average of 23%. If file-based pages are also considered, up to 60% of memory can be saved with an average of 34%. Second, the increase of hot launch latency is mostly in the range of 10-50 ms, leaving the latency usually under 150 ms and thus making users mostly feel it as an instantaneous

launch. This is an impressive result. As an example, with an average of 33% memory saving, a smartphone that currently can hold 10 background apps in the memory without being killed will be able to keep 15 apps alive in the background with MemSaver. For a user whose number of actively used apps is moderately over existing limit, with MemSaver his bumpy app switch experience is removed. Third, additionally considering file-backed pages for swapping either doesn't increase the launch time substantially or sometime even reduce the time. This is because file-backed pages are usually sequentially accessed and prefetching could be activated. Four, we do observe that for some apps (e.g., Figure 11b and Figure 11e) at some contexts (e.g, search and image) MemSaver may occasionally produce a hot launch latency as high as 200-300 ms due to excessive number of page faults. In summary, in general MemSaver can reduce apps' RSS while maintaining near-ideal hot launch latency. In a few cases the latency can be high when compared to the ideal time. However, if compared to the often much higher cold launch times (see Figure 1), the time still represents an improvement.

5 RELATED WORKS

User experience is often dictated by the availability of memory in smartphones. Prior works have focused on improving user experience on many fronts including improving app's memory management and launch performance. To improve app switch time, ASAP [13] uses multiple threads to prefetch pages that are likely to be accessed during a switch. IORap [4] in Android 11 predicts data required by an app ahead of time by profiling its I/O in several cold runs. FALCON [15] uses information including user location and temporal access patterns to predict an app's launch time and preload its data.

Under high memory pressure Android triggers *lkmd* [9] to kill background apps to ease memory pressure. To avoid the killing, Marvin [6] swaps out memory that is less likely to be used in the object granularity with ahead-of-time swap, which requires Android Run Time (ART) modifications. In contrast, MemSaver swaps out pages unlikely to be used in an app's upcoming hot launch in an attempt to postpone the inevitable killing. It doesn't require any modification of apps themselves. SmartSwap [16] predicts least likely used app using information like location and usage history to swap out pages of those apps. Rather than predicting which app will be relaunched, MemSaver only considers app's event-based access history to selectively swap pages out of memory. To avoid disruption of foreground app, Acclaim [8] frees pages from background apps and provide them to foreground apps. A2S [5] integrates process-level kill approach and page-level swap approach by using a threshold to decide processes for killing and by estimating page lifetime to decide pages for swapping out. Instead of predicting which pages to swap out or prefectch or predicting apps as a whole [10, 11], MemSaver uses more intricate information of app-specific context to reduce app's RSS while maintaining near-ideal hot launch time. A more extensive coverage and analysis of related efforts on the improvement of app launch time can be found in the survey paper [3]. It helps to further understand that MemSaver's unique approach and techniques represent a step forward in the improvement of smartphones' use experience.

6 CONCLUSIONS

In this paper we present MemSaver, a low-cost approach for preemptively swapping selected pages of the background apps out of memory while ensuring their near-ideal hot-launch time. Different from conventional LRU-like strategies for selecting memory pages for swapping, MemSaver uniquely resorts to event-specific history to accurately determine the pages for swapping. Evaluation of an implementation of MemSaver in Android shows that up to 60% of application's RSS can be reduced while ensuring that its hot launch time remains in the range friendly to users for real apps of representative types.

REFERENCES

[1] Stuart K Card, George G Robertson, and Jock D Mackinlay. 1991. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*. 181–186.

[2] Tao Deng, Shaheen Kanthawala, Jingbo Meng, Wei Peng, Anastasia Kononova, Qi Hao, Qinhao Zhang, and Prabu David. 2019. Measuring smartphone usage and task switching with log tracking and self-reports. *Mobile Media & Communication* 7, 1 (2019), 3–23.

[3] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. 2022. A Survey of Performance Optimization for Mobile Applications. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2879–2904. https://doi.org/10.1109/TSE.2021.3071193

[4] IORAP. 2023. IORAP. https://medium.com/androiddevelopers/improving-app-startup-with-i-o-prefetching-62fbdb9c9020

[5] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. 2017. Application-aware swapping for mobile systems. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–19.

[6] Niel Lebeck, Arvind Krishnamurthy, Henry M Levy, and Irene Zhang. 2020. End the senseless killing: Improving memory management for mobile operating systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 873–887.

[7] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2020. Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association. https://www.usenix.org/conference/atc20/presentation/liang-yu

[8] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2020. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 897–910.

[9] LKMD. 2023. Low Memory Killer Daemon. https://source.android.com/docs/core/perf/lmkd

[10] Nagarajan Natarajan, Donghyuk Shin, and Inderjit S. Dhillon. 2013. Which App Will You Use next? Collaborative Filtering with Interactional Context. In *Proceedings of the 7th ACM Conference on Recommender Systems* (Hong Kong, China) *(RecSys '13)*. Association for Computing Machinery, New York, NY, USA, 201–208. https://doi.org/10.1145/2507157.2507186

[11] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. 2013. Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Zurich, Switzerland) *(UbiComp '13)*. Association for Computing Machinery, New York, NY, USA, 275–284. https://doi.org/10.1145/2493432.2493490

[12] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W. Lee, and Hongil Yoon. 2021. ASAP: Fast Mobile Application Switch via Adaptive Prepaging. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 365–380. https://www.usenix.org/conference/atc21/presentation/son

[13] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W Lee, and Hongil Yoon. 2021. ASAP: Fast Mobile Application Switch via Adaptive Prepaging.. In *USENIX Annual Technical Conference*. 365–380.

[14] N. Tolia, D.G. Andersen, and M. Satyanarayanan. 2006. Quantifying interactive user experience on thin clients. *Computer* 39, 3 (2006), 46–52.

[15] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (Low Wood Bay, Lake District, UK) *(MobiSys '12)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2307636.2307648

[16] Xiao Zhu, Duo Liu, Kan Zhong, Jinting Ren, and Tao Li. 2017. SmartSwap: High-performance and user experience friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

Systemizing and Mitigating Topological Inconsistencies in Alibaba's Microservice Call-graph Datasets

Darby Huye
Tufts University
Medford, MA, USA
darby.huye@tufts.edu

Lan Liu
Tufts University
Medford, MA, USA
lan.liu@tufts.edu

Raja R. Sambasivan
Tufts University
Medford, MA, USA
raja@cs.tufts.edu

ABSTRACT

Alibaba's 2021 and 2022 microservice datasets are the only publicly available sources of request-workflow traces from a large-scale microservice deployment. They have the potential to strongly influence future research as they provide much-needed visibility into industrial microservices' characteristics. We conduct the first systematic analyses of both datasets to help facilitate their use by the community. We find that the 2021 dataset contains numerous inconsistencies preventing accurate reconstruction of full trace topologies. The 2022 dataset also suffers from inconsistencies, but at a much lower rate. Tools that strictly follow Alibaba's specs for constructing traces from these datasets will silently ignore these inconsistencies, misinforming researchers by creating traces of the wrong sizes and shapes. Tools that discard traces with inconsistencies will discard many traces. We present Casper, a construction method that uses redundancies in the datasets to sidestep the inconsistencies. Compared to an approach that discards traces with inconsistencies, Casper accurately reconstructs an additional 25.5% of traces in the 2021 dataset (going from 58.32% to 83.82%) and an additional 12.18% in the 2022 dataset (going from 86.42% to 98.6%).

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; *Reliability*; • **Software and its engineering** → **Traceability**.

KEYWORDS

Distributed tracing, Cloud Computing, Mitigating data loss

ACM Reference Format:
Darby Huye, Lan Liu, and Raja R. Sambasivan. 2024. Systemizing and Mitigating Topological Inconsistencies in Alibaba's Microservice Call-graph Datasets. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3629526.3645043

1 INTRODUCTION

Today, organizations build distributed applications using a microservice architecture [5, 10]. This architecture—which involves decomposing applications' functionalities into many lightweight services that coordinate over well-defined APIs to process user requests—has many advantages. It facilitates development teams' independence, increases deployment velocity, and enables fine-grained scaling [8, 20]. But, apart from this shared understanding, microservice deployments' concrete characteristics are invisible outside of their respective organizations. This lack of visibility depresses research into microservices. Especially affected are efforts on scheduling, problem mitigation, and debugging, which rely on knowing deployments' scale and emergent properties of how and which services interact to process requests. Published microservices research on these topics [9, 12, 23, 26, 27, 29] are often informed by simple testbeds [4, 10, 30], making their applicability to the large-scale organizations that benefit most from microservices questionable.

Large-scale organizations, such as Alibaba [16], Google [24], and Meta [13] recently published quantitative analyses of their respective microservice architectures. These published studies provide much needed insight into the scale and complexity of the architectures as well as detailed studies of request workflows within them—i.e., how services interact to process requests. Many research efforts are using these analyses to inform their work [3, 7]. Unfortunately, these studies do not provide the raw datasets used for their analyses [24] or only provide summary statistics [13]. This prevents the community from independently verifying results, finding new insights themselves, or using the datasets directly in their work.

To address this concern, Alibaba released two datasets capturing traces of request workflows observed in their microservice architecture [1, 2]. Traces are call graphs, where nodes are services and edges indicate caller/callee relationships between them. Additional annotations on nodes and edges include (but are not limited to) response times, communication protocols used, and service instance ID. The datasets store traces in tabular form with rows corresponding to calls between service. The 2021 dataset covers a 12-hour time-period, totaling 20 million traces. The 2022 dataset covers a 13-day period, totaling over 13 billion traces (estimated). These datasets are treasure troves for research and they are already being used extensively [15, 17, 18, 31]. But, without independent analyses to verify datasets' fidelity, researchers run the risk of basing their work on inaccurate trace data.

This paper provides the needed independent analysis. We analyze the entire 2021 dataset and a 12-hour period of the 2022 dataset. We find that numerous 2021 traces are stored in the dataset in ways inconsistent with Alibaba's specifications. The 2022 traces also exhibit inconsistencies, but at a lower rate. As a result of these inconsistencies, trace graphs built from the datasets will not represent requests' workflows, misleading users. We find that the inconsistencies can be explained by two types of events within the distributed-tracing infrastructure [16] responsible for capturing traces: data-loss and context-propagation errors. The former occurs when log messages (or parts of them) representing individual calls are dropped. The latter occurs when services fail to differentiate

This work is licensed under a Creative Commons Attribution-ShareAlike International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645043

different calls made on behalf of a single request.

To mitigate these inconsistencies, we present Casper, a trace construction toolkit that uses hidden redundancies in the datasets to recover from data loss and disambiguate merged calls. We show that Casper creates larger and wider traces than other construction methods—e.g., ones that operate unaware of the inconsistencies or which discard traces with inconsistencies.

We present the following contributions:

(1) We identify and systematize cases where trace data stored in the datasets is inconsistent with the stated specifications. Specifically, 99.48% of traces in the 2021 dataset suffers from these inconsistencies and 85.77% of traces the 2022 dataset suffer from missing calls, contradicting values within rows, and rows that appear more times than expected. We discuss how these inconsistencies affect traces' shapes.
(2) We describe how many of these inconsistencies are explained by: *1)* data-loss events and: *2)* services incorrectly propagating trace context to differentiate inter-service calls. We show how redundancies between `rpcid` values in trace context and caller/callee names within datasets' rows allow many inconsistencies to be circumvented.
(3) We present Casper, a trace construction algorithm that circumvents the inconsistencies[1]. For the 2021 dataset, Casper creates traces that have average sizes, max depths, and max widths that are: 1.14x, 1.22x, and 1.08x larger than a construction method that blindly ignores inconsistencies and 2.71x, 1.32x, and 2.02x larger than a method that discards inconsistent traces. It creates an additional 25.5% of traces with complete connectivity between services for the 2021 dataset (increasing the total to 83.82%) and an additional 12.18% with complete connectivity for the 2022 dataset (increasing the total to 98.6%).

2 ALIBABA MICROSERVICE DATASETS

This section describes the Alibaba datasets' tabular format and the specifications describing traces that are stored within in them (§2.1). We also describe how traces can be constructed from the tabular data using the specifications (§2.1). §3 discusses how the datasets are inconsistent from the specifications and their impact on traces constructed assuming consistency.

We start with a brief description of Alibaba's distributed-tracing infrastructure, which was responsible for capturing the traces. Please see Luo et al. [16] and the datasets READMEs [1, 2] for a more detailed description of the tracing infrastructure and the datasets respectively.

Alibaba's distributed-tracing infrastructure for capturing request-workflow traces: Like most distributed-tracing infrastructures [14, 21, 25], Alibaba's infrastructure works by propagating context with requests' execution. For Alibaba, context includes a per-request unique ID (`traceid`) and a per-call path unique ID (called the `rpcid`). The `traceid` uniquely identifies calls made on behalf of a single request. The `rpcid` uniquely identifies calls and their depth within the trace call graph. It is specified as a series of delimiter '.' + IDs. Each service adds a delimiter and adds a unique ID before calling a downstream service. The number of delimiters is equal to the depth of the call. When requests execute log messages, records of them are enriched with context and stored in long-term storage. The trace datasets are comprised of the subset of these logs indicating caller/callee relationships.

[1]Source code and sample data: https://doi.org/10.7910/DVN/SS9SIY

2.1 Tabular format & storage specifications

Format: Figure 1a shows a simplified version of the tabular format, which is largely the same for both datasets. It also shows the graph representation of the trace in Figure 1b. Various columns provide information needed to create the trace topology (nodes and edges) and add annotations. Rows represent log messages or edges of the trace graph—i.e., communication calls between an upstream service (caller) and downstream service (callee). Up to two rows may correspond to a single communication call.

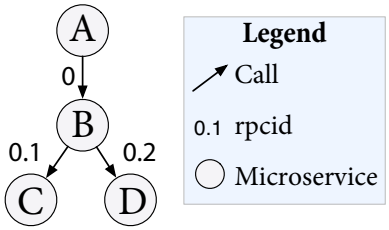
Specifications: The `traceid`, `rpcid`, `UM`, and `DM` columns encode traces' topological information. The first three fields are propagated in context as described above. For a given call the `UM` and `DM` fields identify the corresponding upstream service (caller) and downstream service (callee).

The remaining columns are used to annotate trace nodes or edges. We discuss only ones relevant to our analyses. The `rpctype` column describes the protocol used for a given call. It can be either *RPC*, *HTTP*, *mc* (Memcache), *mq* (Message queue), or *db* (database). The `rt` column describes the call's response time and `ts` denotes a timestamp indicating when the row was recorded by a service. There are two rows for each RPC and HTTP-based call. The first row records the end-to-end response time as measured by the upstream service. The second row records the processing time of the request within the downstream service—i.e., the latency between receiving the request and sending a reply.

Differences between 2021 and 2022 datasets In the 2021 dataset, the row corresponding to end-to-end latency records a positive response time whereas the row corresponding to downstream processing records a negative one [1]. Both response-time values are positive in the 2022 dataset [2]. The 2022 dataset includes additional

	Topology				Annotations	
ts	traceid	rpcid	UM	DM	rpctype	rt
166370	1	0	A	B	http	8
166374	1	0	A	B	http	-7
166376	1	0.1	B	C	db	0
166372	1	0.2	B	D	mc	0

(a) An encoded trace in tabular form



(b) Corresponding constructed trace

Figure 1: A trace in tabular form & its constructed version. Not all annotations are shown in the tabular version. Annotations are omitted from the graph. Table follows the 2021 data format.

annotation fields.

2.2 Constructing traces

The construction process described below is general to accommodate traces that start at any arbitrary point of request workflows' execution (i.e., not necessarily at frontend services where workflows typically originate). Responses to questions about the datasets from Alibaba indicate such intermediate starting points are possible [11]. Figure 1b shows the trace that would be constructed from the tabular data in Table 1a.

To build a trace: *1)* Extract all rows of the table with the same `traceid`. *2)* Group rows with the same `rpcid` together as they represent the same call. *2)* Find roots, which are named by the `UM` or `DM` field of calls with the fewest number of `'.'` delimiters in their `rpcid` values. The `UM` is the root if it is defined, else `DM` is the root. The former accommodates intermediate starting points and the latter frontends. *3)* Find calls made by roots, which have the same `rpcid` prefix as roots with one extra `'.'` delimiter, and attach their `DM` values as children. *4)* Repeat step 3 recursively for all leaves in the trace until there are no remaining calls left. Nodes and edges can be optionally annotated during this process.

3 TRACE INCONSISTENCIES

We identified four types of inconsistencies in the Alibaba trace datasets that invalidate the assumptions mentioned in §2. We found these inconsistencies to be prevalent within the datasets with almost all traces having at least one inconsistency. For the remainder of this section, we discuss inconsistencies within the context of a single trace. We focus mainly on the 2021 trace dataset since it has higher error rates, but all inconsistencies discussed were also observed in the 2022 dataset.

Analysis of Alibaba's responses to questions about the datasets [6, 19, 28] indicate that these inconsistencies are explained by data loss and context-propagation errors (CPEs). Data loss results in rows being dropped or fields in rows missing values. Context propagation errors occur when a user does not correctly increment the `rpcid`, assigning the same `rpcid` to many calls. This non-unique `rpcid` is propagated downstream, resulting in downstream calls also having non-unique `rpcids`. The last subsection (§3.5) gives an example on how many of these inconsistencies may arise given a context propagation error.

Table 1 lists the percentage of unique traces affected by each inconsistency. We next describe each inconsistency in detail.

3.1 Missing rows

There are many missing rows in the trace datasets. We categorize missing rows into two groups: missing duplicate rows and missing `rpcids`. The duplicate row (i.e. the call or reply) for two-way communication (*http* or *rpc*) is often missing, leaving only one row with either a positive or negative `rt` value. Most traces are missing a duplicate row in both datasets.

A missing `rpcid` is defined to be when we are missing all rows for a `rpcid`. Since `rpcids` encode topological information about the request workflow, we know all rows for a `rpcid` are missing when we are missing a `rpcid` that is ancestrally between two captured `rpcids` and is needed to form a call path. We call these missing `rpcids` internal `rpcids` (since they are internal nodes in a call graph). A trace may have missing rows before the smallest `rpcid` or after the largest `rpcid`, but there is no way to detect this.

Inconsistency	2021	2022
Missing duplicate row	99.48%	85.77%
Missing `rpcid`	35.23%	11.42%
Unexpected row	30.16%	6.86%
Contradicting `UM`	33%	7.94%
Contradicting `DM`	26.84%	2.56%
Missing value	94.2%	67.05%

Table 1: Inconsistency frequencies. The portion of traces that have at least once occurrence of each inconsistency.

Example: Table 2 gives an example of a missing row and `rpcid`. We are missing the negative `rt` row for the *http* request 0.2. Additionally, we are missing all rows for the `rpcid` 0.2.1, which is the connection between 0.2 and 0.2.1.1.

Implication: Missing duplicate rows does not impact the shape of the trace since the remaining row contains the call path information. We only lose the `rt` for one direction of communication. When we are missing `rpcids` in a trace, naive rebuilding (using the assumptions outlined in §2) would result in a disconnected trace, under-counting the number of calls and not preserving the true topology.

How to address the inconsistency: Data loss causes us to lose rows, which can present as either a missing duplicate row or a missing `rpcid`. We can use redundant information about the structure of a trace in the `rpcids` to replace missing internal `rpcids`, when it's available. For example, in Table 2, the missing `rpcid` 0.2.1 should have `UM` B and `DM` C to form a valid call path.

3.2 Additional unexpected rows

As described in Section 2, `rpcids` are assumed to be unique for each call in the system. We expect to see one row per message sent; for two-way communication, there should be two rows (call & reply) and for one-way communication there should be one row. Additionally, when we have a reply row for an `rpcid`, all structural information (e.g. `UM`, `DM`, `rpctype`) should be identical since it references a single call. Despite this assumption, we often see additional rows past these thresholds for a single `rpcid`. In the 2021 traces, 42.18% of traces have at least one `rpcid` with unexpected rows.

Example: Table 3 shows an example of additional unexpected rows where 0.3.1 is repeated many times. We have four rows to `DM` C (counting both the + and - `rt` rows) and one row to `DM` D.

rpcid	UM	DM	rpctype	rt
0.2	A	B	http	+
0.2.1.1	C	D	db	+

Table 2: Missing rpcids. The rpcid 0.2.1 is missing from the table since it's needed to connect 0.2 and 0.2.1.1. Additionally, a duplicate row is missing for the *http* request 0.2.

rpcid	UM	DM	rpctype	rt
0.3	A	B	http	+/-
0.3.1	B	C	rpc	+/-
0.3.1	B	C	rpc	+/-
0.3.1	B	D	mq	+
0.3.1.1	C	E	mq	+

Table 3: Unexpected rows. `rpcid` 0.3.1 is repeated above the expected threshold for the `rpctype`. +/- `rt` is used to indicate we have both the positive and negative rows.

Implication: The assumption that `rpcid`s are unique is invalidated and rebuilding the trace naïvely would under-count the number of unique calls. Additionally, when there are multiple `UM`, `DM` pairs, it's not clear which should be used for the `rpcid`.

How to address the inconsistency Additional unexpected rows are caused by context propagation errors (CPEs), where the user of the tracing infrastructure did not increment the `rpcid` for each unique call. This appears in the table as additional rows with the same `rpcid` and `UM`, but potentially different `DM`s. In Table 3, the CPE originates from service B, which makes at least two calls to service C and one to D. The non-unique `rpcid`s are passed downstream, creating more non-unique `rpcid`s (and call paths) further downstream. For example, 0.3.1.1 (in Table 3) has `UM` C, but we do not know which of B's calls to C made the subsequent call to E.

3.3 Contradicting Values

There are inconsistencies in the datasets where rows that should contain identical values have conflicting values (e.g. the two rows corresponding to the call and reply for a two-way communication call should have the same `UM` and `DM`). We categorize contradicting values into two groups: contradicting `DM`s are when all rows with a `UM` has multiple `DM`s and contradicting `UM`s are when one or more of the `UM`s for an `rpcid` don't match the upstream call's `DM` (i.e. not forming a valid call path). Contradicting `UM`s are independent of the `DM` value. A single `rpcid` can have both types of contradicting values in their rows. A large portion of the 2021 traces (26%) have at least one `rpcid` with contradicting `DM` values and 37% of the traces have contradicting `UM`s.

Example: Table 4 shows two rows for `rpcid` 0.3.1.1. There is conflicting `UM` and `DM` information in the two rows. Since there are multiple `UM` values, at least one of the rows may not connect upstream resulting in an invalid path.

Implication: Naïvely rebuilding trace with contradicting values could create invalid call paths, depending on which row's information is added to the trace topology. Since each unique `rpcid` is assumed to have one `UM` and one `DM`, naïvely rebuilding would not check for this inconsistency.

How to address the inconsistency Contradicting values are the result of context propagation errors. Contradicting `DM`s appear when the user does not increment the `rpcid` when making calls to different downstream services. Upon cursory inspection, we found contradicting `UM`s are either downstream from CPEs or seem to have an incorrect `rpcid` that could be remedied by adding a '.' followed by an integer to connect the call one level downstream. We can use redundant information about the call paths to help determine accurate `UM`s and `DM`s (or if the `rpcid` is not unique).

rpcid	UM	DM	rpctype	rt
0.3.1.1	C	E	mq	+
0.3.1.1	D	F	mq	+

Table 4: Contradicting values.

3.4 Missing Values

Many values in the datasets are missing or contain `'(?)'/UNKNOWN` as the value. In fact, most traces in both datasets contain at least one missing `UM` or `DM` value (94.2% and 69%).

Implication: Naïvely rebuilding traces with missing values misses opportunities to uncover the true value, resulting in skewed metrics about the frequency of specific microservices.

How to address the inconsistency Missing values are the result of data loss, which is not uncommon in large distributed systems. For two-way communication, there is often a duplicate row for the `rpcid` which contains the missing information. If there is no duplicate row, these missing values can be recovered using call path information from an upstream or downstream call.

3.5 Combination of inconsistencies

Context propagation errors (CPE) often show up as a combination of the inconsistencies. We always see unexpected rows for CPEs, but this is typically combined with contradicting values. For example Table 3 showed both `DM` values C and D, which are contradicting.

Downstream from CPEs, the same `rpcid` is used as a seed for downstream calls. In the Table 3 example, 0.3.1 is the seed `rpcid` for all downstream calls made from C and D. If we added an additional row to this example with `rpcid` 0.3.1.1.1, `UM` X and `DM` Y, we would have a contradicting path (since X is not the same as upstream call 0.3.1.1's `DM` E). To make things more complicated, the contradicting path inconsistency would no longer exist if we assumed we were missing its' upstream `rpcid` (which could have `DM` X). The point here is that CPEs cause many inconsistencies since they invalidate the assumption that `rpcid`s are unique. This makes it challenging (and sometimes impossible) to decipher the trace topology.

4 CASPER

We introduce Casper, which aims to create the largest accurate request workflow topologies. Building on the intuition in §3, we discuss which inconsistencies can be circumvented (e.g., are recoverable), and how to recover from them (§4.1). We also discuss when inconsistencies are not recoverable (§4.2). We then present Casper's algorithm for rebuilding the traces (§4.3). We conclude with limitations of our approach (§4.4). Casper is implemented in 711 lines of Python code. It takes as input all rows for a trace and outputs the constructed trace in Alibaba tabular or OpenTelemetry JSON [21] format.

4.1 Recoverable inconsistencies

4.1.1 Data loss. **Missing internal calls**: Observing missing internal calls due to data loss, we can determine the exact number of missing calls on the call path using the `rpcid` structure. We find the call that is missing its upstream call and recursively drop the

trailing '.' from the rpcid until we reach an existing rpcid. Each new rpcid we create by dropping a '.' becomes a call in the trace.

Example: In figure 2, (1) shows how rpcids 0.1.1 and 0.1.1.1 are added to the trace to fill the hole between the existing rpcids.

Missing values captured in redundant rows: We use information captured in duplicate rows or through call paths to fill in missing UMs and DMs.

Example: In figure 2, (2) shows a recovered (missing) rpcid 0.2.1, which was added to connect the existing rpcids. In table format, we can refill 0.2.1's UM with DM B (from its upstream call) and DM C (from its downstream call).

4.1.2 Context propagation errors (unique paths). **Context propagation errors (CPEs) can be fixed at the source**: As stated in §3, a CPE originates from a service which incorrectly uses the same rpcid for different calls. We call a CPE's origin the source of the error. By differentiating each rpcid, we can always rebuild the trace structure at the source of a CPE.

First, we calculate the number of unique calls that were incorrectly assigned the same rpcid. The 2021 traces and 2022 traces use rts differently, so we handle each case independently. The 2021 traces use negative rts for reply rows. When the rt is below a certain threshold, it is rounded down to 0 (both for the call and reply edges). The call rt includes the child execution time plus the network latency while the reply rt is only the child execution time, so the reply edges must have a rt less than or equal to the call rt. We use these characteristics to calculate the number of calls for a given rpctype to each DM. For one-way communication, the rt values should all be positive (mostly 0), so the number of calls is equivalent to the number of rows.

For two-way communication in the 2021 dataset, we calculate number of calls as follows:

$$\begin{aligned} num_fast_calls &= \lfloor \frac{(rt==0)}{2} \rfloor \\ extra_fast_row &= (rt==0)\%2 \\ num_slow_calls &= \max(-\texttt{rt}, abs(+\texttt{rt} - extra_fast_row)) \\ num_calls_to_DM &= num_fast_calls + num_slow_calls \end{aligned} \quad (1)$$

-rt is the number of rows with negative response times and +rt is the number of rows with non-zero positive response times. Since two-way communication should have one positive and one negative row, where the negative row could be 0, we get the minimum number of unique calls if we maximize the number of +/- pairs made. num_fast_calls counts the pairs of rows with 0 rt. $extra_fast_row$ is 1 if there is an odd number of rows with 0 rt. We try to pair any leftover 0 rt rows with +rt rows. num_slow_calls tries to match the 0 rt row with a positive rt row (taking the absolute value when there are no positive rt rows), and then counts the pairs between the remaining + rt rows and - rt rows. Taking the max gives us the total number of pairs and the remaining unpaired rows. Finally, we add the fast and slow calls to get the total number.

The 2022 traces only have positive rts, the number of calls is calculated by:

$$num_calls_to_DM = ceiling(rt/2) \quad (2)$$

for two-way communication calls (and is the number of rows for one-way communication).

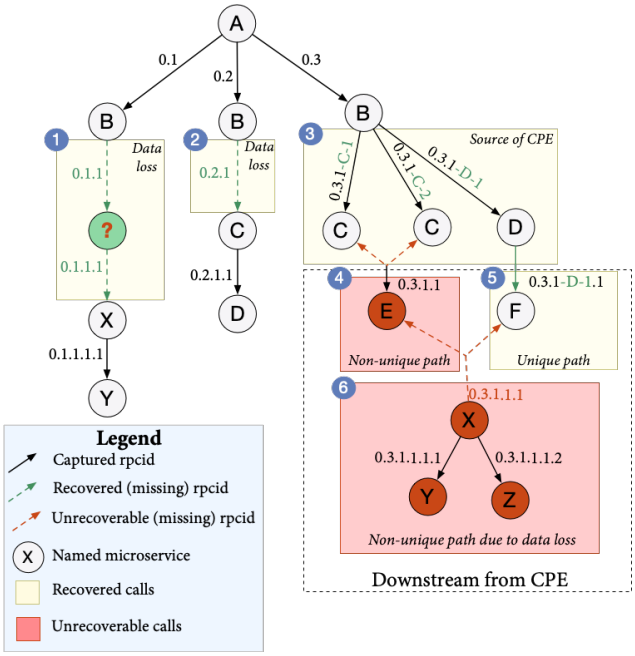


Figure 2: CASPER example trace. Regions of the trace that are corrected are highlighted in yellow and regions that are fundamentally unfixable are highlighted in red.

Since we can always be missing rows, the true number of unique calls is unknown. These calculations determine the minimum number of unique calls.

Using the minimum number of calls to each DM, we can fix the topology at the source of a CPE. We do this by updating the rpcids to be unique for each call. We modify the rpcids to be of the form $rpcid - DM - i$ where i is between 1 and the minimum number of calls to the DM.

Example: In figure 2, (3) shows how the rpcids are updated to be unique for a CPE. The tabular version of this data (table 3) has five rows for the rpcid 0.3.1. We calculate that there are two calls to C and one to D. We update the rpcids to be: 0.3.1-C-1, 0.3.1-C-2, and 0.3.1-D-1.

Unique call paths downstream from CPEs: When there is only one call to a DM at the source of a CPE, there is a possibility that we can rebuild the trace downstream from this service. If there are multiple calls to a DM, we cannot determine which instance of the DM made which downstream calls.

All downstream rpcids from CPEs are not assumed to be unique because they share a non-unique rpcid in their ancestry. As a result, we can only rely on the call depth information from rpcids as being accurate. We can use UM and DM information to rebuild call paths downstream from CPEs when the call path is 1) unique (i.e. the chain of UM, DM, and call depth information forms a single valid call path) and 2) complete (there is no data loss or unknown values in the call path).

Example: In figure 2, (5) shows a unique path downstream from a CPE that we can connect to the trace. Table 4 shows the tabular version of this portion of the trace, where there is a row for 0.3.1.1 with UM D, connecting to the single D node in the trace. We update the rpcid to be unique by changing its non-unique ancestor to be

unique (e.g. 0.3.1.1 → 0.3.1-D-1.1).

4.2 Unrecoverable inconsistencies

4.2.1 Data loss. **Data loss that has no redundancies**: Most instances of data loss have redundancies in the dataset. However, there are some instances where there are no redundancies and the data is unrecoverable. For example, when sequential rpcids are missing, not all UM and DM information is recoverable.

Example: In figure 2, (1) shows that we are able to recover the missing rpcids 0.1.1 and 0.1.1.1, but we cannot determine the service name for the additional node.

4.2.2 Context propagation errors (non-unique paths). **Non-unique call paths downstream from CPEs**: We cannot remedy calls downstream from CPEs when they do not form unique call path. In the presence of data loss, it is fundamentally not possible to replace the missing calls since we cannot determine a unique ancestry to reconnect via. When we have missing UM or DM values, they are not recoverable since there are often many unique possible values.

Example: In figure 2, (4) shows how the rpcid 0.3.1.1 (from table 3) cannot be uniquely connected to the trace. Since we cannot definitively connect this edge to the existing trace, we remove it and all rows downstream from it. Figure 2 (6), which visually represents the data in table 5, is affected by data loss. We are missing a row for the rpcid 0.3.1.1.1, which is needed to determine if the node X connects uniquely to the trace via the node F or non-uniquely to the trace via node E.

Conflicting paths, where the UM does not connect upstream: As described in section 3.3, conflicting UMs have a special case where they are not downstream from a CPE. In this case, there is only one service upstream. Any rows with UMs that do not match the upstream's DM are dropped as they form invalid call paths.

4.3 Casper algorithm

The Casper algorithm performs a best case reconstruction of the trace topologies and guarantees that the edges in the resulting graphs are accurate. We keep track of the number of unrecoverable rpcids that are omitted from the traces. Casper begins with general preprocessing of the data including filling missing values with duplicate rows. The meat of the program is in handling data loss and CPEs, which is outlined below.

At a high-level, Casper performs a breadth first search (BFS) traversal over the rpcids in each trace. When it identifies data loss upstream from a rpcid, it recursively fills in missing calls until it connects upstream. When it identifies a CPE, it fixes the error at the source and attempts to reconstruct the rpcids downstream from the CPE (algorithm 2) before returning to the BFS traversal (algorithm 1).

For each trace, Casper is initialized by sorting the rpcids in BFS order and identifying all root rpcids, which have no upstream

rpcid	UM	DM	rpctype	rt
0.3.1.1.1.1	X	Y	db	+
0.3.1.1.1.2	X	Z	db	+

Table 5: Unrecoverable call paths downstream from CPE, affected by data loss.

Algorithm 1 Casper algorithm for a single trace

```
1:  rpcids ← BFS sorted rpcids for this trace
2:  root_rpcids ← allroots
3:  for rpcid in rpcids do
4:      upstream_rpcid = rpcid.rsplit('.',1)[0];
5:      if Data loss then
6:          while upstream_rpcid not in trace do ‘
7:              Add edge for upstream_rpcid;
8:              upstream_rpcid ← next upstream_rpcid;
9:          end while
10:     end if
11:     if Context propagation error then
12:         DMs ← unique DMs for rpcid;
13:         for DM in DMs do
14:             min_calls_to_DM ← max(R+,R−);
15:             for i in min_calls_to_DM do
16:                 Add edge for rpcid_DM_i;
17:             end for
18:         end for
19:         Rebuild downstream CP rpcids (Alg 2);
20:     end if Add edge for rpcid;
21: end for
```

rpcids (alg 1, lines 1–2). It then iterates over the rpcids in BFS order from each root and performs:

(1) Check for data loss: if the rpcid is not a root, drop the trailing '.' and check for a *upstream_rpcid* (alg 1, lines 4–5).
   (a) If the *upstream_rpcid* does not exist, recursively add calls (with UM/DM values when possible) until we connect to the trace (alg 1, lines 6–9).
(2) Check for CPE: Calculate the minimum number of calls made by this rpcid. If there are more than the expected number of rows (alg 1, line 11):
   (a) Extract the list of unique DMs called by the UM. For each DM, calculate the number of calls to the DM. Create a unique rpcid for each call to the DM of the form rpcid=rpcid−DM−*i* where i ranges from 1 to the number of call (alg 1, lines 12–18).
   (b) Extract all rpcids downstream from the CPE and rebuild independently. These rows have different assumptions (i.e. that the rpcid is not unique) so must be handled separately (alg 1, line 19).
(3) Add edge to the trace, if no errors (alg 1, line 20)

Algorithm 2 describes how we remedy rpcids downstream from CPEs. At a high-level, Casper first identifies and removes subtrees in the trace that are downstream from data loss (i.e. disconnected subtrees). Next, Casper performs call path validation, filtering out non-unique call paths. Algorithm 2 is initialized with only the rpcids downstream from the source of a CPE, which are sorted in BFS order (alg 2, lines 1–2).

(1) Check for downstream data loss: For each rpcid, check if it has a dangling tree below it. If the rpcid has no direct children, but has descendants (alg 2, lines 3–4):
   (a) Get the list of descendant rpcids and delete all rows with these rpcids (alg 2, line 5–6).

(2) For each *row* that is downstream from the CPE and not affected by data loss, calculate the number of calls this row connects to upstream. This is done by generating the *parent*_rpcid, filtering the parent rows that connect to this row's UM, and calculating the number of calls for those rows (alg 2, line 9–10):
   (a) If this row connects to a unique call path: update it's rpcid to be unique, by replacing its ancestry rpcid with its corrected *parent*_rpcid (alg 2, line 11–12).
   (b) If this row connects to a multiple call paths: delete the row and any connecting downstream call paths. (alg 2, line 13–14)

Algorithm 2 supports fixing sequential CPEs as long as the call paths are unique.

Algorithm 2 Casper rebuild calls downstream from CPE

```
1:  rows ← rows with rpcids downstream from CPE;
2:  rpcids ←BFS sorted rpcids downstream from CPE;
3:  for rpcid in rpcids do
4:      if No child rpcid exists but exists descendents then
5:          decendent_rpcids ← rpcid.* from rpcids;
6:          delete rows for decendent_rpcids;
7:      end if
8:  end for
9:  for row in rows do
10:     num_connecting_calls ← num calls row connects upstream to;
11:     if num_connecting_calls == 1 then
12:         update rpcid in row;
13:     else
14:         delete row & connecting downstream rows;
15:     end if
16: end for
```

4.4 Limitations

Missing rpcids before or after all recorded rpcids: Missing rpcids that are smaller than the smallest rpcid in the table or larger than the largest rpcid in the table. As mentioned in [6], root rpcids can be anything (although it's most commonly 0 or 0.1). Additionally, we found there can be multiple roots in a single trace. When the root rpcid is larger than 0.1 (i.e. starting at a 'depth' of greater than 2), it could be the true root of the trace or it could be missing rpcids before it.

Corrupted values: We must make assumptions about the trace data that allow us to rebuild the topology. In addition to the assumptions provided by Alibaba, we assume traceids are accurate. If traceids (or any other fields for that matter), are corrupted in unpredictable ways, we cannot guarantee we will identify it and can remedy it.

5 EVALUATION OF CASPER

We seek to answer the following regarding Casper's efficacy in circumventing inconsistencies in the Alibaba datasets.

Q1: How are traces generated by Casper different from traces generated with other approaches? More specifically, what trace topological characteristics change using different reconstruction approaches?

Q2 How much do the recovery mechanisms in Casper impact trace topologies?

Q3 How many additional complete traces does Casper reconstruct compared to filtering out all traces with any inconsistencies?

The answers to *Q1* are a cautionary tale to researchers using the Alibaba trace dataset that reconstructing traces ignoring errors will lead to vastly skewed results that may impact the design or evaluation of their research artifacts. The answers to *Q2* evaluate the effectiveness of the recovery mechanisms in Casper. The answer to *Q3* informs us how much the built-in side-channel redundancies in the Alibaba traces are able to help correct the inconsistencies without dropping communication calls or filtering entire traces.

For all of these analyses, we use a randomly sampled 10.8% (2,240,550) of the 2021 trace data and 1% (5,079,746) of the 2022 trace data. We use a lower sampling rate for the 2022 dataset because it contains many more traces than the 2021 version. We used six r6.xlarge instances to split the datasets as per traceid and one c5a.23xlarge EC2 instance w/90 threads to run the Casper algorithm and other construction approaches described in this section. The other construction methods are variants on the Casper implementation and run inline with its execution.

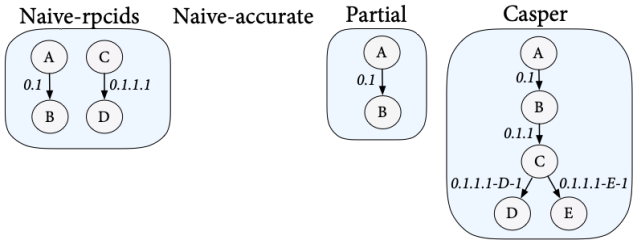
5.1 Comparing construction methods

5.1.1 Methodology. We consider three alternative approaches to Casper for constructing traces without deep knowledge of the inconsistencies in the datasets: naive-rpcid, naive-accurate, and partial. Figure 3 illustrates how the four approaches differ when reconstructing one example trace that has inconsistencies. We describe the alternative rebuild modes below.

naive-rpcid: keeps the first occurrence of a unique rpcid in the table and neither detects nor recovers any inconsistencies (similar to construction process in §2). Figure 3 shows the first row for each rpcid represented as a graph. The trace is disconnected since rpcid

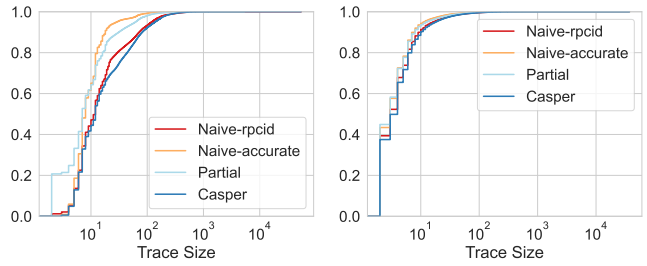
rpcid	UM	DM	rpctype	rt
0.1	A	B	rpc	+/-
0.1.1.1	C	D	rpc	+/-
0.1.1.1	C	E	db	+

(a) Trace Example



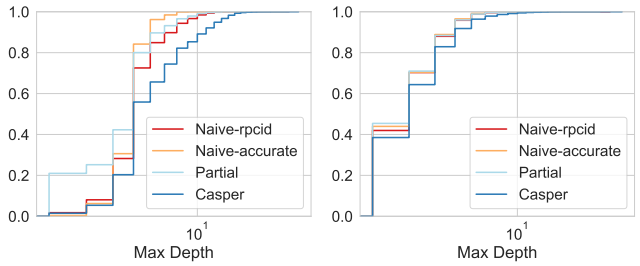
(b) Modes

Figure 3: Trace building modes. Four different modes for building trace graphs, each has a different method of handling errors and inconsistencies.



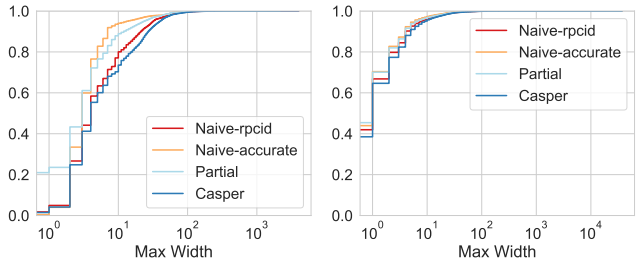
(a) 2021 (b) 2022

Figure 4: Trace sizes for different modes



(a) 2021 (b) 2022

Figure 5: Maximum trace depth for different modes



(a) 2021 (b) 2022

Figure 6: Maximum trace width for different modes

0.1.1 is missing. Additionally, we are missing node E since it was not captured in the first row for its `rpcid`.

`naive-accurate`: detects inconsistencies and ignores the entire trace if an inconsistency is identified. Figure 3 shows once the missing `rpcid` 0.1.1 is detected, all rows for the trace are deleted.

`partial`: keeps calls in the traces that are not affected by an inconsistency. When an inconsistency is identified, the entire downstream call path is removed from the trace. `partial` traces preserve the accurate portions of traces. Figure 3 shows once the missing `rpcid` 0.1.1 is detected, all downstream rows are deleted.

`Casper`: as described in section 4. Figure 3 shows how Casper fixes the missing call 0.1.1 and the CPE at 0.1.1.1, updating the repeated `rpcid` to be unique for each call.

For `naive-accurate` and `partial`, we allow inconsistencies that are trivial to fix (e.g. missing values and missing duplicate rows) since they do not affect the trace topology.

To compare the four approaches, we analyze the following topological characteristics of the reconstructed traces: trace size, call depth, and width. Trace size represents the total number of microservices in the trace. A microservice can be called many times within a trace and each call is included in the trace size. Call depth is the maximum depth of the call paths in the traces. Width is the maximum number of calls made by a single microservices. In graph form, this is the largest fan-out. We compare the cumulative distribution functions (CDFs) for all metrics.

5.1.2 Results. Overall, we find that Casper traces are larger, wider, and deeper than all other methods of constructing traces for both the 2021 and 2022 datasets. The 2022 traces are smaller (size, depth, and width) than the 2021 traces. The 2022 traces have less inconsistencies, so Casper's impact on the trace topology compared to `naive-rpcid` is less significant.

Trace size: For both the 2021 and 2022, Casper builds larger traces than all other approaches by correcting data loss and CPEs. Figure 4 shows the CDF of trace sizes for both years. For 2021, at the 50th percentile (P50), a Casper trace is the same size as `naive-rpcid` traces. However, Casper produces larger traces at P75 than all other modes. On average, Casper traces have size 33.08 whereas the size is 12.22, 15.50, and 29.11 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, Casper traces have similar, the same or slightly bigger, size compared to all other modes at all percentiles. Traces from 2022 are overall smaller than those from 2021. The average trace size `naive-accurate` is 12.22 in 2021, but is decreased to 4.98 in 2022.

Call depth: For both the 2021 and 2022, Casper builds deeper traces than all other approaches by reconnecting call paths that have missing `rpcids`. Figure 5 shows the CDF of the maximum call depth of a trace for both years. For 2021, at P50, Casper traces have the same depth as the other modes. However, Casper produces deeper traces at P75 than all other modes. On average, Casper traces have depth 6.40 whereas the depth is 4.84, 4.56, and 5.26 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, Casper traces have the same depth or slightly deeper than all other modes at all percentiles. Traces from 2022 are overall shallower than those from 2021. The average depth for a `naive-accurate` trace is 4.84 in 2021, but is decreased to 3.01 in 2022.

Width: For both the 2021 and 2022, Casper builds wider traces than all other approaches by differentiating repeated `rpcids` generated by CPEs. Figure 6 shows the CDF of maximum width of a trace for both years. For 2021, P25 and P50, a Casper trace has similar width as the other modes. At P75, Casper traces are wider. On average, Casper traces have width 10.73 whereas the width is 5.30, 5.91, and 8.97 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, Casper traces are similarly wide or slightly wider than all other modes at all percentiles. Traces from 2022 are overall narrower than those from 2021. The average width for `naive-accurate` traces is 5.30 in 2021, but is decreased to 1.92 in 2022.

5.2 Impact of recovery mechanisms

5.2.1 Methodology. We break down Casper's recovery mechanisms (described in §4.3) into four parts and quantify their impact. 1) *Adds missing calls* counts the number of new calls added to a trace. 2) *Fills in missing values* counts the originally unknown microservice names that Casper recovers. 3) *Updates* `rpcids` *at a CPE source* measures the number of `rpcids` added when differentiating calls at the first occurrence of a CPE in a call path. 4) *Recovers* `rpcids` *downstream from a CPE source* counts the number of calls Casper identified as uniquely connected to the trace downstream from the first CPE in the call path.

Casper collects metrics for each of the recovery mechanisms when reconstructing the traces. We calculate the number of traces that are affected by each recovery and its impact within the trace.

5.2.2 Results. **Adds missing calls**: Casper recovers all missing internal calls in a trace. For 2021, 30.47% of traces have at least one missing internal `rpcid`, with the average number of calls 10.7 (std: 23.79, P99: 115). For 2022, 8.77% of traces have at least one missing call added by Casper, with average 3.57 (std: 13.12, P99: 30). Casper reconnects broken traces, resulting in longer call paths.

Fills in missing values: Casper fills in missing `DM` values using duplicate information stored in rows or call paths. For 2021 traces, we are able to recover at least one `DM` in 99.98% of traces. On average, traces have 2.84 recovered `DM` names (std: 5.61, P99: 27). For 2022 traces, we are able to recover at least one `DM` in all traces. On average, traces have 2.61 recovered `DM` names (std: 6.99, P99: 34).

Updates `rpcids` at a CPE source: Given a call path starting from the root call, the first occurrence of a CPE is a CPE source. Casper modifies `rpcids` to be unique to differentiate different calls that originally shared the same `rpcid`. This modification may preserve more branches and yield wider traces. For 2021 traces, Casper modifies on average 2.77 `rpcids` per CPE source (std: 2.65 and P99: 11). For 2022 traces, Casper modifies on average 2.02 `rpcids` at a CPE source (std: 0.18 and P99: 3).

Recovers `rpcids` downstream from a CPE source: Casper connects unique call paths downstream from a CPE source, updating their `rpcids` to be unique which may preserve longer call path and yields deeper traces. We measure this impact by counting the number of such updated `rpcids` per CPE source. For 2021 traces, Casper modified on average 3.08 downstream `rpcids` (std: 11.22 and P99: 48). For 2022 traces, Casper modified on average 1.11 downstream `rpcids` (std: 32.97 and P99: 19). Note that additional CPEs can occur downstream, but they are rare. For 2021, the average number downstream CPEs is 0.23, (std: 1.5 and P99: 6). For 2022, the average number downstream CPEs is 0.14, (std: 2.71 and P99: 3).

5.3 Additional complete traces

5.3.1 Methodology. We evaluate Casper's effectiveness at rebuilding complete traces by measuring the number of additional complete traces output by Casper (when compared to `naive-accurate`). A trace is complete if there are no unrecoverable `rpcids` (explained in Section 4.3).

5.3.2 Results. For 2021, 58.32% of the traces have complete topology without needing to remedy any inconsistencies. Casper reconstructs an additional 25.5% of the traces, totaling to 83.82%.

For 2022, 86.42% of the traces have complete topology without needing to remedy any inconsistencies. Casper reconstructs an additional 12.18% of the traces, totaling to 98.6%.

6 DISCUSSION

Recommendation for consumers of the Alibaba datasets and related research papers: Users of the datasets should always specify their methodology for identifying and mitigating inconsistencies in their analyses. They should prefer the 2022 dataset, which contains fewer total inconsistencies. But, it is unclear if traces in the 2022 dataset were collected from the same applications or application versions as the 2021 dataset. The maximum trace sizes and maximum widths differ significantly between both years regardless of rebuild mode. As such, consumers may wish to use both datasets to test their work against a range of request-workflow characteristics.

We recommend caution when interpreting research that uses the 2021 dataset without specifying a methodology for handling inconsistencies. Readers should carefully consider if changes in trace characteristics or connectivity would affect the results. Of particular note is the Alibaba microservice analysis by Luo et al. [16]. This analysis uses a 7-day dataset of which the 2021 public release is a subset. But, does not specify whether the authors knew about the inconsistencies or whether they addressed them. As such, the presented graphs of trace characteristics, clustering results, and distributions suggest for the artificial trace generator may be suspect.

Exploring tradeoffs in capturing redundancies within trace data: Casper's functionality is possible because Alibaba's tracing infrastructure stores redundant data in caller/callee log messages. Namely, `rpcid` uniquely locates a call in the trace, allowing call-graph connectivity between services when intermediate calls are lost. But, `rpcids`' expressiveness results in larger context and larger network message sizes. Context-propagation errors can be circumvented by using chains of `UM` / `DM` fields. In contrast, the popular open-source model for distributed-tracing, OpenTelemetry [21], does not capture any redundancies. Spans (e.g., service executions) can be dropped if services' are too resource starved [22], leading to traces with (silent) missing nodes. Research is needed to explore how to encode redundancies in trace data or context and the overhead tradeoffs of doing so.

Tools to identify context-propagation errors: Capturing high-fidelity traces that represent their workflow relies on correct context propagation. Worse, context propagation errors can propagate downstream, making it difficult to identify the offending service. Tools, similar to lint, are needed that can detect whether services are propagating context correctly. These tools should be used prior to deploying new services or new versions.

7 SUMMARY

We systematized inconsistencies found in Alibaba's distributed tracing data and identified two root causes for these inconsistencies: data loss and context propagation errors. We built Casper, an toolkit which can remedy most inconsistencies in the trace data, building the largest accurate topologies. We evaluated Casper against other methods of constructing traces and show that our topologies are larger and more complex than other methods.

8 ACKNOWLEDGEMENTS

We thank the anonymous ACM/SPEC ICPE conference reviewers, Zhaoqi (Roy) Zhang, Chami Lamelas, Juraci Kröhling, Yuri Shkuro, and members of the D.O.C.C Lab at Tufts University for their insightful feedback and support. We thank the people at Alibaba who released such valuable data for our research community.

REFERENCES

[1] Alibaba's 2021 cluster trace of microservices 2021. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021.
[2] Alibaba's 2022 cluster trace of microservices 2022. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022.

[3] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. 2023. Blueprint: A Toolchain for Highly-Reconfigurable Microservice Applications. In *SOSP'23: Proceedings of the 25th Symposium on Operating Systems Principles*.

[4] BookInfo application 2024. https://istio.io/latest/docs/examples/bookinfo/

[5] Adrian Cockroft. 2016. The evolution of microservices. https://learning.acm.org/techtalks/microservices

[6] Confused of identifying services from MS_CallGraph_Table in MicroserviceTrace-v2021 2022. https://github.com/alibaba/clusterdata/issues/157.

[7] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. 2023. Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications. In *SOSP'23: Proceedings of the 25th Symposium on Operating Systems Principles*.

[8] Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html

[9] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*. 135–151. https://doi.org/10.1145/3445814.3446700

[10] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS'19: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[11] Github issue #157: Confused of identifying services from MS_CallGraph_Table in MicroserviceTrace-v2021 2021. https://github.com/alibaba/clusterdata/issues/157.

[12] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*. https://doi.org/10.1145/3472883.3486994

[13] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. 2023. Lifting the veil on Meta's microservice architecture: Aanalyses and topology and request workflows. In *ATC'23: Proceedings of the 2023 USENIX Annual Technical Conference*.

[14] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An end-to-end performance tracing and analysis system. In *SOSP'17: Proceedings of the 26th Symposium on Operating Systems Principles*.

[15] Chengzhi Lu, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2023. Understanding and Optimizing Workloads for Unified Resource Management in Large Cloud Platforms. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 416–432. https://doi.org/10.1145/3552326.3587437

[16] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*.

[17] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2022. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 62–77. https://doi.org/10.1145/3567955.3567964

[18] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The Power of Prediction: Microservice Auto Scaling via Workload Learning. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 355–369. https://doi.org/10.1145/3542929.3563477

[19] Microservice MSCallGraph: a call is recorded more than twice 2022. https://github.com/alibaba/clusterdata/issues/124

[20] Sam Newman. 2021. *Building microservices: designing fine-grained systems* (2nd ed.). O'Reilly Media, Inc.

[21] OpenTelemetry 2024. https://opentelemetry.io/

[22] OpenTelemetry Memory Limiter Processor 2024. https://github.com/open-telemetry/opentelemetry-collector/blob/main/processor/memorylimiterprocessor/README.md

[23] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *OSDI'20: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 805–825.

[24] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *SOSP'23: Proceedings of the 25th Symposium on Operating Systems Principles*.

[25] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report dapper-2010-1. Google.

[26] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. 2021. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*.

[27] Mert Toslali, Srinivasan Parthasarathy, Fabio Oliveira, Hai Huang, and Ayse K Coskun. 2021. Iter8: Online Experimentation in the Cloud. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*.

[28] Why are some call graph in cluster-trace-microservices-v2021 disconnected 2023. https://github.com/alibaba/clusterdata/issues/175.

[29] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*. 167–181. https://doi.org/10.1145/3445814.3446693

[30] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. POSTER: Benchmarking microservice Systems for software engineering research. In *ICSE'18 Companion: Proceedings of the 40th Companion to the International Conference on Software Engineering*.

[31] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2022. Dissecting Service Mesh Overheads. arXiv:2207.00592 [cs.DC]

Disambiguating Performance Anomalies from Workload Changes in Cloud-Native Applications

Alexandru Baluta
York University
Toronto, Ontario, Canada
balutaal@yorku.ca

Yar Rouf
York University
Toronto, Ontario, Canada
yarrouf@my.yorku.ca

Joydeep Mukherjee
California Polytechnic State University
San Luis Obispo, California, USA
jmukherj@calpoly.edu

Zhen Ming Jiang
York University
Toronto, Ontario, Canada
zmjiang@cse.yorku.ca

Marin Litoiu
York University
Toronto, Ontario, Canada
mlitoiu@yorku.ca

ABSTRACT

Modern cloud-native applications are adopting the microservice architecture in which applications are deployed in lightweight containers that run inside a virtual machine (VM). Containers running different services are often co-located inside the same virtual machine. While this enables better resource optimization, it can cause interference among applications. This can lead to performance degradation. Detecting the cause of performance degradation at runtime is crucial to decide the correct remediation action such as, but not limited to, scaling or migrating. We propose a non-intrusive detection technique that differentiates between degradation caused by load and by interference. First, we define an operational zone for the application. Then we define a disambiguation method that uses models to classify interference and normal load. In contrast to previous work, our proposed detection technique does not require intrusive application instrumentation and incurs minimal performance overhead. We demonstrate how we can design effective Machine Learning models that can be generalized to detect interference from different types of applications. We evaluate our technique using realistic microservice benchmarks on AWS EC2. The results show that our approach outperforms existing interference detection techniques in F_1 score by at least 2.75% and at most 53.86%.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **System administration**.

KEYWORDS

Microservice; Interference; Cloud Computing; Machine Learning

ACM Reference Format:
Alexandru Baluta, Yar Rouf, Joydeep Mukherjee, Zhen Ming Jiang, and Marin Litoiu. 2024. Disambiguating Performance Anomalies from Workload Changes in Cloud-Native Applications. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645046

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645046

1 INTRODUCTION

DevOps practice encourages the use of microservice architecture, where a traditional monolithic application is broken down into a collection of easier-to-manage smaller services. Benefits of microservice architecture include better scalability, easier continuous delivery support, data decentralization, and improved fault isolation. Due to these advantages, more application developers are adopting the microservice architecture recently. Microservices are deployed as cloud-native applications on public cloud platforms with each service encapsulated within a container running inside a virtual machine (VM). Containers have risen in popularity for their faster deployment speeds and their ability to allow applications to run in complete isolation from one another without incurring extra overhead. Container orchestration platforms such as Docker [25] and Kubernetes [7] are increasingly gaining popularity and are commonly used in public cloud platforms such as Amazon Web Services (AWS) [3] and Google Cloud Platform [4].

Multiple microservices are generally consolidated on a single VM. This is done to improve resource consumption levels inside a VM and thereby optimize cloud costs. When multiple microservices are deployed on the same VM, they can often compete with each other for shared host-level resources. Such shared resource contention can alter application behavior and make it deviate from the development time specifications and performance. We refer to application performance degradation due to shared resource contention as *performance interference*. Performance interference has been observed before in applications running on public cloud environments [17, 20, 21, 23, 28–30, 36].

Detecting and disambiguating performance interference anomalies from other causes of performance degradation is very important for application runtime self-management and for avoiding outages. For example, a performance degradation due to legitimate workload surge can be automatically mitigated by horizontal scaling whereas interference might be mitigated though application redeployment and relocation. Another use case where detection interference anomalies is important is in reducing the number of

alerts the IT operators have to deal with. It is estimated that the an IT operator faces *80-100* alerts per system [35]. We need to detect if there is interference, and disambiguate if the performance degradation is from interference or from normal behavior. Therefore, it is important that an application detects when its execution environment is perturbed significantly by other applications.

Automatically detecting runtime interference in cloud-native applications is a challenging task. Application workload variability and unpredictability can produce the same effects as interference, and hence it is difficult to distinguish between them. Researchers have looked, with limited success, into instrumenting applications [11], recording their mean request response times, and comparing them against their baseline response times to detect the presence of interference. Continuously instrumenting application code and monitoring the response times of running services [19] can significantly increase the cost of development and maintenance. In addition, continuous monitoring of service metrics can incur a prohibitive overhead when the service is facing a heavy workload [10, 18]. Furthermore, interference detection techniques that model the baseline behavior may not generalize well to scenarios where the interfering application changes at runtime.

To address some of the above challenges, we propose a lightweight method that uses a small number of deployment and runtime performance metrics to automatically disambiguate interference and workload effects on performance. The method involves building or training a model such as queuing, regression, or machine learning models prior to deployment. By using interference for training when using machine learning models, we are no longer dependent on costly response time instrumentation. At runtime, we can use the model and readily available performance data to detect the interference. The method is non-intrusive and does not require any application performance profiling. We use resource utilization metrics that can be easily collected at runtime from within each application container along with simple application level metrics such as request throughput or response time. The assumption is that an autonomic manager is application specific, has access only to managed application metrics and does not see the other application metrics, only their effect on the managed application metrics. The basis of this assumption is that the environment is dynamic and collocated applications might be deployed after the target application. This is in line with the current practice when we develop autoscalers for each application. We also make the assumption that the ML model can be trained as a DevOps process activity and then used at runtime within an AIOps (Artificial Intelligence for IT Operations) platform for interference detection and mitigation.

The paper addresses the following research questions:

RQ-1: How severe is the impact of performance interference in cloud-native applications deployed on public cloud? To address this, we performed experiments to characterize the impact of interference on a microservice benchmark, Acme [1], hosted on the AWS EC2 cloud platform. The results indicate that the response time of Acme can be severely impacted by interference by at least a factor of 39% and at most a factor of 6955% at moderate CPU utilization levels.

RQ-2: How well does a disambiguation method perform when the interfering workloads used for model training and deployment are similar? To answer RQ-2, we develop and train models to detect interference caused by applications with similar performance characteristics. Results show that ML models outperform state-of-the-art regression based and threshold based interference detection techniques by at least 0.67% and at most 23.29%. Furthermore, our method incurs minimal overhead of only 1% to 2% on the service response time at runtime.

RQ-3: How well does our disambiguation method perform when the interfering workloads used for model training and deployment are not similar? This question covers a common case when we do not know a priori what applications might share the infrastructure with our managed application. To address RQ-3, we show that we can develop and train models to detect interference caused by an arbitrary application that stresses the same resources as used by the target application. In addition, this model generalizes well for detecting interference from different containerized interfering applications. Lastly, we evaluate scalability by using our model to detect interference against a large multi-tired microservice application. As presented in our results, our technique outperforms state-of-the-art regression based and threshold based interference detection techniques by at least 2.75% and at most 53.86%.

The paper makes the following contributions:

(1) We introduce a formal definition of cloud-native application interference disambiguation, rooted in queuing theory.
(2) We introduce an interference detection method that generalizes across interference scenarios and outperforms the state-of-the-art.
(3) We evaluate the method using several model types, including queuing and machine learning models.
(4) We validate the method on four industrial strength applications and show that it scales to large deployments and works in public clouds.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 details the motivation and methodology. Section 4 describes experimental setup. Section 5 presents the results of RQ-1, RQ-2, and RQ-3. Section 6 details threats to validity. Section 7 is the conclusion and discussion of future work.

2 RELATED WORK

2.1 Interference Definitions

Koh et al. [23] investigate the impact of VM-level performance interference from co-located workloads and characterize interference impact on low level system metrics. They cluster workloads by interference type and construct performance prediction models for co-located workloads using weighted means and regression analysis. Paul et al. [30] similarly characterize performance interference in co-located VMs and further measure the application performance degradation and impact on low-level system metrics. The authors highlight types of workloads that perform well or not when co-located. In contrast to these work, we characterize the impact of interference among containerized microservices.

2.2 Interference Measurement and Classification

Jha et al. [20] measured intra-microservice and inter-microservice interference using four benchmark microservices. The results of their experiments showed significant container interference present in both intra-container and inter-container cases. Garg, and Lakshmi. [17] ran experiments using well-known containerized benchmarks to detect microservice interference and accordingly identified the shared resources subject to contention in these scenarios. Novaković et al. [29] propose DeepDive to mitigate VM-level performance interference. Their approach detects interference by comparing low-level system metrics against a workload's baseline values and migrates VMs to other physical machines as needed. Wang et al. [36] present Vmon, a system that detects and quantifies VM-level performance interference. Vmon profiles an application running on a VM to observe how Hardware Performance Counters correlate with application performance. DeepDive and Vmons detect VM-level interference through use of low-level system metrics whereas our work detects microservice interference using metrics that are readily available and do not pose prohibitive overhead in collecting. Additionally, our work is differentiated in that we evaluate the effectiveness of model re-use across different scenarios.

2.3 Interference Regression

Joshi et al. [21] propose Sherlock, a method for detecting long-lived performance interference in containerized services caused by co-located VMs. The authors employ a regression detection technique to model performance interference using VM and application-level metrics at runtime. Kang and Lama [22] predict microservice interference using Gaussian Process models trained at runtime using a sliding window technique. Baluta et al. [12] predict microservice interference using AutoML models trained at runtime using a sliding window technique. Our work differentiates from these as we explore the reuse of pre-trained interference detection models in varying environments.

3 DEFINITIONS, MODELS AND METHODOLOGY

Assume that at deployment time, we profile the performance of an application and the interference effects in the space (U, λ, R), where U is the utilization, R is the response time and λ is the workload (arrival rate of requests) of the application and infrastructure. That profile is captured in a model and the profile is defined over a large workload space and interference levels.

In Figure 1, we illustrate interference in a simplified two-dimensional space, U and R. The blue area is the baseline profile for one workload ([λ=λ_1]). The response time of application, with no interference, falls within 95 percentile Confidence interval, denoted [$minR_{base}$ $maxR_{base}$]. With the help of tunable slack variable, α, we define a *operational zone* around the operational area (gray area). This parameter is application-dependent and defined by the application owner and can be derived from Service Level Agreements. Interference zones are those areas in the space (U, λ, R) where response time is outside the interval [$minR_{base}$-α, $maxR_{base}$+α] but cannot be explained through the change of the workload. For example, red areas in the figure are interference anomalies, whereas

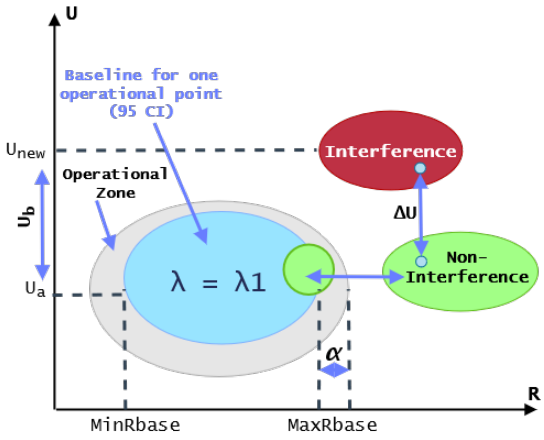


Figure 1: Interference Range

green areas depict normal behaviour, non-interference, caused by workload. Although the anomaly depicted in red can be explained by other causes (bugs, misconfigurations, etc.), in this paper we focus on interference anomalies. ΔU is the utilization step size between non-interference and what is classified as interference. Although both are outside the operational zone, it is important to know what causes the shift in the operation region. Different root causes of anomalies can be mitigated through different runtime actions.

In Section 3.1, we formulate interference as a Queuing Network problem. In Section 3.2, we motivate the use of Machine Learning to address interference. In Section 3.3, we describe our methodology including data collection and pre-processing, as well as model training and deployment.

3.1 Queuing Network Models and Interference Modeling

In this section, we explain how interference can be formulated as Queuing Network Models (QNM). To begin, we consider a containerized application a running inside a VM without interference, as seen in Figure 2. We assume that the application a stresses a single resource k in the VM so as to incur an utilization of $U_{a,k}$ on the resource. We assume a request arrival rate of λ_a to application a. If, at runtime, we had a way to measure service demand $D_{a,k}$ of application a at resource k, we could then predict the no-interference mean request response time R_a of application a as:

$$R_a = \frac{D_{a,k}}{1 - U_k} \tag{1}$$

where U_k is the total utilization incurred at resource k in the VM. Since in a no-interference scenario, the only application stressing resource k is our monitored application a, $U_k = U_{a,k}$. Finally, we substitute U_k with $U_{a,k}$ in eqn. 1 to obtain the response mean time R_a of application a in a no-interference environment as given by:

$$R_a = \frac{D_{a,k}}{1 - U_{a,k}} \tag{2}$$

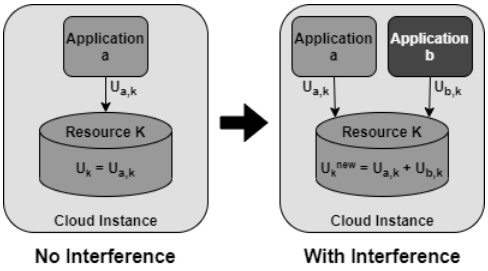


Figure 2: Effect of Interference on Resource Utilization

By considering the forced and utilization laws as well as steady state, we have

$$R_a = \frac{D_{a,k}}{1 - D_{a,k} * \lambda} \tag{3}$$

and this allows us to extrapolate the response time outside of an operational point and for any workload λ (cf. green areas in Figure 1).

To illustrate interference, consider another application b running inside the same VM, as seen in Figure 2. Application b incurs an utilization of $U_{b,k}$ on resource k inside the VM. Accordingly, from eqn. 1, the new response time R_a^{new} of application a when it faces interference from application b is:

$$R_a^{new} = \frac{D_{a,k}}{1 - U_k^{new}} \tag{4}$$

where U_k^{new} represents the total utilization of resource k, i.e. the sum total of the utilization incurred by both applications a and b at resource k given by:

$$U_k^{new} = U_{a,k} + U_{b,k} \tag{5}$$

Equations 2 and 5 explain interference (cf. Figure 1, red zones).

3.1.1 QNM Methodology. We tune QNM following the approaches in [14, 38]. Since we work under the assumption that we do not have access to the metrics of other applications and we consider the system is a product form network [13], we can use Eq. 3 to estimate the response time R and compare it with the measured response time. If the measured response time is outside the bounds [R-α, R+α] then we label it as affected by interference, otherwise we consider the deviation as caused by application load.

3.2 Machine Learning Classification and Interference Modeling

Based on eqn. 2 and eqn. 4, we observe that a point $[\lambda_a, R_a, U_k]$ is transposed to $[\lambda_a, R_a^{new}, U_k^{new}]$ in the same tri-dimensional space under the impact of interference. Based on this observation, we aim to use resource and application metrics to train ML models that can classify two distinct classes of response time values, one class denoting the no-interference scenario represented by R_a, and the other denoting the interference scenario represented by R_a^{new} (corresponding to red and green zones in Figure 1). We note that it appear more practical to use ML models instead of QNMs for interference detection since it is infeasible to accurately measure resource service demands for all containers belonging to a monitored microservice application serving different kinds of workloads. Furthermore, multiple levels of virtualization involved in a cloud-based microservice container can involve estimating service demands for unknown virtualized resources [27], which can be difficult. In contrast, ML models use resource utilization and throughput metrics that are easy to collect at the container and VM levels.

We use our understanding of QNMs as motivation for applying machine learning to detect interference. From eqn. 4, since the new response time R_a^{new} of application a is only impacted by the total utilization U_k incurred at resource k, it follows that R_a^{new} remains unchanged even if U_k is incurred by different types of applications as long as the levels of total utilization incurred at resource k are similar. Consequently, we aim to use this logic as motivation for training our ML models to predict interference classes with one type of benchmark application which can then be used at runtime to classify similar interference from other types of applications.

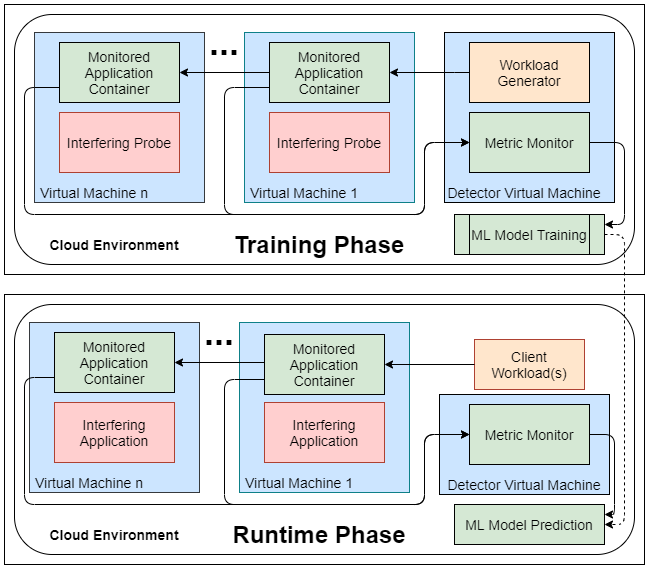


Figure 3: Overview of Interference Detection Technique

3.3 Machine Learning Methodology

In this section, we present our methodology focused on ML models since they are central to our work. Section 3.3.1 defines the assumptions of our technique. Section 3.3.2 details data generation and pre-processing. Section 3.3.3 describes the methodology to train an ML model for interference detection. Section 3.3.4 details the use of an ML model for interference detection at runtime.

3.3.1 General Methodology. We refer to the in-production application as the *target application*. The application owner deploys their target application on managed cloud services. In doing so, the application owner delegates management of physical servers or even VMs to the cloud provider. Consequently, the application owner only has guaranteed visibility into the target application's containers and access to VM level metrics. Our target application consists of multi-tiered microservices, each bundled inside a container. It is typical for microservice applications to be distributed over several VMs, with each VM hosting one or more microservice containers.

In our study, we consider an *interfering application* as one that runs on the same VMs as the target application and competes for shared VM resources.

Our detection technique monitors resource utilization data from inside the VM and the application containers metrics that are easy to collect. We record container utilization metrics at each target application container along with VM utilization metrics collected at a *sampling interval* of t seconds. We also record the overall application throughput and the response time, at each sampling interval. The sampling interval needs to be tuned for each target application so as to incur minimal levels of performance overhead on the system.

3.3.2 Data Collection and Pre-Processing for ML. Figure 3 presents the high level overview of our interference detection technique. As seen in the figure, our technique runs in two phases. The first phase is the *training phase* where we run controlled experiments to train our ML model in an offline model training environment. In the training phase, we run our target microservice application in a controlled cloud environment where its response time is not impacted by performance interference. This is done by running the target application in isolation on the n VMs hosting the microservice without any interfering application present. The objective is to obtain baseline metrics when no interference is present. To this end, we increase the arrival rate of the application workload to the target application in steps to incur a wide range of resource utilization observed at each application container. This is done through a *Workload Generator* tool running inside a Detector VM which resides alongside the application VMs, as seen in Figure 3. We ensure that our workload generation setup is free from internal software bottlenecks and network latency issues by following the approach detailed in past work [28]. Since application owners can only access metrics from the VM and their own application containers, we omit including metrics of any other kind in our data set for ML model training. To this end, a *Metric Monitor* tool from inside our Detector VM, as shown in Figure 3, collects measurable metrics from the target application and its environment such as application response time and throughput, and container and VM resource usage. Although we choose the average request response time of a target application as our performance metric, other metrics such as the mean throughput values can also be used. Using only the application throughput and response time along with VM and application resource utilization metrics as input allows for a smaller feature set that is easy to monitor, collect and does not incur prohibitive monitoring overhead at runtime.

We repeat each step of workload generation N times to obtain N estimates of the average application request response time R_{base} at each step. We refer to N as the *number of workload repetitions*. We use this data to construct the 95% Confidence Interval (CI) of the baseline application response time at each step. The upper and lower limits of this CI are denoted as $maxR_{base}$ and $minR_{base}$, respectively. To mitigate against performance variability inherent in public cloud platforms, we repeat each step, i.e. set the value of N in our training phase such that the width of our CI is within 5% of its sample mean at each step. Alternatively, if the cloud platform has significant performance variability, the application owner may consider deploying the application containers inside *dedicated VMs* which have same specification as general VMs and are offered by public cloud platforms to provide stable performance. Although dedicated VMs are more expensive than general VMs, the application owners only need to use them for a short period of time to collect training data for ML models.

Once we obtain the 95% CI of the application's baseline response time, we next introduce interference into the system. For this purpose, we run a controllable *interfering probe* along with the target application on the n VMs hosting the target application as shown in Figure 3. Doing so imposes additional stress on shared resources which is expected to increase response time as depicted in Equation 4. Similar to before, we send the same step-wise increasing workload to our target application from our workload generator tool from inside the Detector VM. We also simultaneously vary load on the interfering probe to diversify its degree of shared resource contention and introduce different levels of performance interference on our target application at each step. We record the average request response time R_t of the target application along with VM and container utilization metrics collected by the Metric Monitor tool when the interfering probe is also running. This is done at our sampling interval of t seconds continuously for a duration of x seconds to obtain the training data set to be used in the ML model. The application throughput, response times, and container and VM utilization metrics for each sampling interval represent a single record in the training data set of our ML model.

We use the baseline response time data obtained in the training phase to label the ML data set in our offline model training. As mentioned before, our ML model outputs binary classification with 2 labels, where label 1 indicates interference and label 0 indicates no-interference. To this end, we compare the value of R_t at each record in this set with its corresponding value of $maxR_{base}$ obtained at the same step of workload. As depicted in Figure 1, if R_t exceeds $maxR_{base}$, we infer that the application response time suffers from performance interference and accordingly assign the label 1 to the record. Otherwise, the record is assigned with label 0. Once the training data set is labeled, we remove the associated response time pairing from each record to construct the final input data set to the ML model.

3.3.3 ML Model Training. AutoML [15, 24, 32] has gained popularity as an effective solution for training well-performing ML models. The AutoML framework evaluates several ML models trained on the same data set against one another and outputs the best performing model. We leverage the H2O AutoML framework [24] in our ML model construction. This framework can be easily integrated with the DevOps feedback loop to automate runtime interference detection.

The framework trains multiple models including but not limited to XGBoost and Stacked Ensemble Models. An ensemble model is composition of several ML models and their predictions. The framework automatically tunes ML algorithm hyper-parameters using random grid search over a predefined range of possible hyper-parameter values. H2O AutoML iteratively evaluates models under these varying hyper-parameter configurations and outputs the best performing model. In addition, the H2O AutoML framework employs 5-Fold cross validation by default to promote models generalizing well to unseen data. We employ an 80-20 stratified train-test

set split while preserving class ratios. We evaluate our models by *Precision*, *Recall*, and F_1 *score*.

When a model makes positive predictions indicating interference as present, the precision score measures how often interference is truly present in the environment. Recall on the other hand measures how well a model identifies true instances of interference relative to all positive predictions made by the model. F_1 score is the harmonic mean of precision and recall and summarizes their joint behavior in a single metric.

3.3.4 ML Model Runtime Deployment. Once ML model construction is complete, we move on to the second phase, i.e., the *runtime phase*, where we deploy the ML model to detect container interference at runtime as seen in Figure 3. In this phase, we continuously monitor throughput, VM and container resource utilization at runtime as the microservice serves client traffic. Subsequently, this data is fed as input to the ML model, which in turn classifies the current runtime state of the target application as either 'interference' or 'no-interference'.

4 EXPERIMENTAL VALIDATION

The experiments are designed to answer research questions RQ-1 to RQ-3 for a range of deployments and interference types. We consider: a) multiple target and interference applications; b) different deployments for the target and interference applications; c) different types of interference. In this section, we show the design of the experiments and the implementation setup. Section 5 details the results of these experiments.

4.1 Target and Interference applications

In the experiments, we use 'in-production' applications such as ACME Air and Boutique as our Target application. Acme Air emulates transactions for an airline website and consists of 2 microservices, a NodeJS Web server, and a MongoDB database. Acme Air is a well known and frequently used benchmark [33, 34]. We run these 2 microservices in their own Docker containers. We refer to these containers as the *Acme-Web* and *Acme-Db* containers respectively. Online Boutique [6] is a popular open-source benchmark application developed by Google. The e-commerce application is composed of multiple microservices deployed on Kubernetes, an open-source container orchestration system.

As interference applications, we use stress-ng and Air Quality Monitor. Stress-ng [9] is a linux stress tool that allows the user to stress the CPU, memory, disk and other resources. We use stress-ng as a configurable artificial application benchmark which can be used to generate a wide range of resource utilization levels. Air Quality Monitor [2] is an Internet of Things (IoT) application that processes air quality sensor data.

4.2 Interference Types

We consider the following interference types:

(1) *Correlated and similar workloads (CSI).* In this use case, we consider that the interference is generated by applications in which the load at different VMs is *correlated* with the load of an ingress service/gateway). The interfering applications belong to the same class as the in-production application, therefore the workloads of the in-production and interfering application are *similar*. We consider e-commerce applications, like ACME, for both training and inference.
(2) *Correlated but dissimilar workloads (CDI).* Here, we consider that the interfering applications belongs to a different class than the in-production target application. We use ACME as the in-production application and the Air Quality Monitor application, [2] which is an Internet of Things (IoT) application, as the interference.
(3) *Non-correlated and dissimilar workloads (NDI).* We consider that interference can happen at any VM, independently (*uncorrelated*). Also, the interference is not similar to the load of the in-production application. We use stress-ng [9] for training and inference. Prior work [16] has used the stress-ng benchmark to incur varying level of resource utilization by stressing the system under study.

4.3 Deployment types

We want to show that we can distinguish the interference from normal load in multiple application configurations. For that we consider several deployment types for the in-production application.

4.3.1 Single Virtual Machine (1VM). In this deployment, we consolidate our in-production microservice application and the interfering load on the same VM. We run experiments where the interfering application is either 1.) stress-ng (NDI scenario), 2.) another copy of Acme Air (CSI scenario), or 3.) Air Quality Monitor application (CDI scenario). We use our workload generator tool to send a workload to Acme Air while also running our interfering application to incur a wide range of resource utilization and interference levels on our target application.

4.3.2 Dual Virtual Machine (2VM). Here, Acme Air application is distributed across two VMs. This is done to motivate distributed use cases where a microservice application like Acme Air is hosted in containers running across different VMs. In this scenario, we run the Acme-Web and Acme-Db containers on 2 separate VMs respectively. We run NDI, CSI, and CDI scenarios, similar to the single VM experiments. In each of these scenarios we run 1 interfering application container on each of the 2 VMs hosting the Acme-Web and Acme-Db containers respectively. Workloads to these two copies are varied to incur a wide range of resource utilization.

4.3.3 At-scale Deployments. To demonstrate the scalability of our methodology, we selected Online Boutique, a large 11-tier microservice e-commerce application. In our experimentation, we deploy Online Boutique on an Amazon EKS (Elastic Kubernetes Service) Cluster with 4 cluster nodes using an EC2 node group composed of EC2 m5.large VMs. Our Amazon EKS uses Amazon Linux 2 and Docker as the container runtime. We have a total of 14 Kubernetes pods of Online Boutique distributed between the 4 Cluster Nodes. For our interference application, we selected Acme-Air and stress-ng from our previous experiments. For the Acme-Air deployment, we distributed Acme-Air and MongoDB pods on two of our EKS nodes where the Boutique front-end pods reside. This deployment pattern is also repeated with two stress-ng pods. We have in total 9 pods responsible for monitoring our cluster at the container, node and application level.

4.4 Experimental Setup and Methodology

Our experiments were run in the AWS EC2 Cloud. We use multiple m5.large EC2 VMs residing in the same availability zone on the EC2 cloud platform. These VMs run Ubuntu 16.04 and each have 2 VCPUs, 8GiB of RAM, and 64GiB of Elastic Block Storage. We used Docker version 19.03.13 as the containerization platform on our EC2 VMs.

4.4.1 Workload Characteristics. We use *httperf* [26] as the Workload Generator tool to send workload to our target Acme Air application. The httperf client runs inside a separate VM and is not containerized. We use the *httperf* tool to submit the default workload obtained from the official Acme Air project [1] as the workload transaction mix submitted to Acme Air.

We use the number of concurrent connections and inter-request arrival time settings in *httperf* to drive a wide range of application resource utilization. Acme Air workload has a step size increase of approximately 12.5% CPU utilization. stress-ng has a step size increase of 20% CPU utilization. The Air Quality Monitor has a step size increase of 20% CPU utilization.

The workloads were chosen such that comparable Acme Air utilization levels are generated with and without interference present. Otherwise, lacking overlap between interference and non-interference scenarios, a simple utilization threshold algorithm could detect interference with ease. Next, we set the number of workload repetitions $N = 40$ to capture variance. Finally, for the purpose of this study, we set the workload duration $x = 120$ seconds.

For our at-scale deployment, we use Locust, a python-based workload generator [5], to generate the workload on Online Boutique. The workload generator increases the utilization on each of the nodes by 15% increments, ranging from 15% to 80% utilization on the Kubernetes Cluster. We use the same workloads of Acme Air and stress-ng in the previous experiments as the interference workload while the Boutique Workload is running for our experiments.

4.4.2 Metrics Monitoring. To monitor the applications and their environment, we leveraged prominent industry solutions for the *Metric Monitor*. Prometheus is an open-source monitoring solution that integrates with multiple metrics exporters [8]. Additionally, application metrics averages were output in httperf logs. The application metrics are joined with the container and VM metrics from Prometheus by timestamp. These metrics are used in analysis, data labelling, and ML model construction.

4.4.3 RQ-2 Setup. We conduct experiments to compare our interference detection technique with baseline and state-of-the-art techniques. To this end, we first set up experiments where we compare our technique against a simple utilization-threshold detection technique used in previous work [31]. This serves as a baseline technique where we monitor the resource utilization metrics from the Acme-Web and Acme-Db containers at runtime, and indicate the presence of interference if these metrics exceed pre-defined threshold values. We set the pre-defined thresholds of CPU and memory utilization in Acme-Web to 38% and 1.5%, and in Acme-Db to 18% and 38% respectively. We chose these pre-defined threshold values since we observed in earlier experimentation that at these utilization levels, the baseline response time of Acme Air without interference is 3 ms, which is the average baseline response time recorded over all utilization levels incurred by Acme Air. We compare the container utilization metrics of Acme to see if they exceed the pre-defined utilization thresholds. If so, interference is inferred. In addition to the threshold model, we evaluate Queuing Network Models to provide another baseline.

We also conduct another set of experiments to compare our interference detection technique with a state-of-the-art detection technique used in current research [21]. We chose this technique as a regression based approach commonly used in interference and anomaly detection. We adopt the detection method outlined in [21] to construct logistic regression models for interference detection at runtime. Logistical regression predicts the probability of occurrence of a binary classification by using a logistic function. The regression models are fit on the same datasets that were constructed and used for our ML approach as described in section 3.3.2. At runtime, we use the regression model in our single and dual VM experiment setups to predict whether interference is present or not in our Acme Air application. We compare the performance of our QNM and ML approach with the logistical regression model. We evaluate the Precision, Recall, and F_1 score when these baseline technique are applied in our experiments.

4.4.4 RQ-3 Setup. We run experiments to evaluate the effectiveness of ML models in detecting performance interference when the interfering application class at runtime is different from training time. State-of-the-art techniques considered are the same Regression and Threshold based techniques as introduced in the RQ-2 Setup. QNM models are omitted as they explicitly define interfering applications whereas in these experiments we assume no knowledge of the interfering application class at runtime. To train our ML models, we use the same methodology as described in section 3.3.3. For these environments, we choose one of our three applications to serve as the interfering probe benchmark application in the ML model training. These three applications correspond to the NDI, CSI, and CDI scenarios. Next, we deploy these trained ML models in the runtime phase as described in section 3.3.4. However, at runtime, we evaluate the resultant ML models where a different interfering application is present in the environment than as seen at training time. Accordingly, a different interference scenario is encountered at runtime than training time. The goal is to evaluate the model performance against an interfering application class not yet seen by the ML model. To measure the effectiveness of our generalized ML model, we evaluate our method against state-of-the-art models of the logistic regression and simple threshold techniques described in the RQ-2 Setup. The logistic regression model follows a similar methodology as our ML model in that the interference scenario encountered at runtime is different than that used at training time.

5 RESULTS AND DISCUSSION

In this section, we evaluate the interference scenarios for our research questions and present our findings.

5.1 RQ-1 Results: How Severe is the Impact of Interference?

5.1.1 Single VM Experiments Results Analysis. Tables 1, 2, and 3 shows the target Acme Air application's utilization values along

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	14.36%	1.31%	7.56%	41.18%	1.33	1.34	0.75%
469.4	28.66%	1.35%	13.65%	41.83%	1.91	1.93	1.05%
631.49	37.68%	1.39%	18.09%	41.99%	2.58	2.87	11.24%
937.03	53.66%	1.51%	27.88%	42.00%	9	12.53	39.22%

Table 1: NDI Interference in Single VM

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	17.93%	1.25%	9.19%	26.65%	1.22	2.66	118.03%
469.29	34.15%	1.41%	16.90%	27.35%	1.78	24.74	1289.89%
631.48	41.80%	1.44%	21.99%	27.66%	2.4	104.13	4238.75%
673.49	45.95%	1.49%	23.18%	28.01%	7.35	518.53	6954.83%

Table 2: CSI Interference in Single VM

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	15.38%	1.43%	10.04%	20.56%	2.38	8.1	240.34%
469.39	28.08%	1.32%	17.25%	20.64%	2.98	31.83	968.12%
631.4	37.19%	1.40%	23.28%	20.86%	4.16	135.66	3161.06%
938.97	54.04%	1.50%	36.50%	20.78%	16.48	590.88	3485.44%

Table 3: CDI Interference in Single VM

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
469.39	26.03%	1.56%	11.82%	45.20%	1.11	1.81	63.06%
631.48	34.83%	1.61%	15.56%	45.39%	1.57	2.80	78.34%
937	48.88%	1.66%	21.90%	45.39%	3.02	12.23	304.97%
1099.85	54.63%	1.66%	24.29%	45.78%	5.66	24.10	325.80%

Table 4: NDI Interference in Dual VM

the impact of interference on the average request response time of Acme Air. Accordingly, Web CPU and Web Mem refer to the target Acme Air's Web server. Similarly, DB CPU and DB Mem refer to the target Acme Air's Database server. R_{base} and R_t refer to the baseline response time of our target Acme Air application without interference and the runtime response time of Acme Air with interference respectively. As seen in the tables, depending on the resource utilization and workload levels, the average response time of Acme Air is heavily impacted by different levels of performance interference. Even at comparatively light interference levels, the response time of Acme Air more than doubles, as seen in the first row in Table 2.

5.1.2 Dual VM Experiments Results Analysis. Tables 4, 5 and 6 shows the impact of interference on the average request response time of Acme Air in dual VM scenarios. For the dual VM NDI Scenario, the impact of interference on the average response time of Acme Air can be significant. In the worst case, the average Acme Air response time increases from 5.66 ms to 24.1 ms, a 325.8% increase. For the dual VM CSI Scenario, the response time of Acme degrades significantly when running alongside a second copy of the Acme application. In the worst case scenario, the response time of Acme increases from 3.46 ms to 202.47 ms. For the dual VM CDI scenario, Acme's response time degrades significantly in this scenario as well.

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	17.52%	1.31%	9.81%	33.70%	1.11	2.46	121.62%
469.34	33.16%	1.44%	17.60%	33.94%	1.33	12.91	870.68%
631.5	41.92%	1.49%	22.22%	34.28%	1.55	28.47	1736.77%
907.29	57.11%	1.56%	30.12%	34.67%	3.46	202.47	5751.73%

Table 5: CSI Interference in Dual VM

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	14.18%	1.42%	9.35%	26.19%	1.24	5.46	340.32%
469.4	25.98%	1.33%	15.22%	26.44%	1.23	13.67	1011.38%
631.5	34.42%	1.38%	20.74%	26.52%	1.45	26.68	1740.00%
939.21	49.55%	1.49%	30.97%	26.54%	2.32	90.22	3788.79%

Table 6: CDI Interference in Dual VM

In the worst case, the response time of Acme increases from 2.32 ms to 90.22 ms.

RQ-1: Through extensive experimentation, we characterize interference through its impact on container utilization metrics. Results show significant impact of interference on the response time of in-production applications for all interference types considered. For both single and dual VM deployments, response time degrades by at least a factor of 39% and at most a factor of 6955% when the normal CPU utilization levels are between 40% and 60%. We also conclude that the higher the utilization, the higher the impact of interference.

5.2 RQ-2 Results: When Interfering Workloads for Model Training and Deployment are Similar

In this section, we evaluate the performance of ML, QNM and state-of-the-art models to detect interference when the interfering application load is known in the training phase. The applications used for the interfering workloads are the same for both model training and deployment for the RQ-2 results. This is indeed an ideal case, however, it is important in defining a baseline. We compare the effectiveness of three different approaches for all our application scenarios; Regression, QNM and ML.

Figure 4 presents a F_1 score box-plot for ML approach compared to the other state-of-the-art approaches described in Section 4.4.3. On the box-plot, the horizontal bar represents the median F_1 scores across all scenario types and VM deployments. The higher the median, the shorter the box, the better the performance. The Threshold model has the lowest performance and the Regression model has high variability of F_1 scores, showing that the Regression has difficulty consistently detecting interference across different scenarios and deployment types. We performed the ANOVA test on our data to to confirm the significant differences between the approaches (f-value = 16.343801, p-value = 0.000013). Overall, we observe from the box-plot that our ML approach outperforms all state-of-the-art methods with the highest F_1 scores and lowest variability.

Table 7 captures a closer look on the details of the performance of ML approach and state-of-the-art techniques described in Section 4.4.3. The Threshold and QNM approaches performs the worst in all

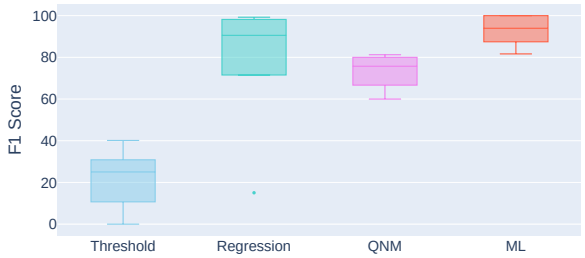


Figure 4: Performance of ML versus State-of-the-Art Models

Scenario	Approach	F_1 Score	Precision	Recall
NDI in 1VM	Threshold	30.75%	22.69%	47.69%
NDI in 1VM	Regression	15.04%	40.0%	9.26%
NDI in 1VM	QNM	60.0%	46.15%	85.71%
NDI in 1VM	**ML**	**81.64%**	**100.0%**	**68.98%**
CSI in 1VM	Threshold	16.67%	87.5%	9.21%
CSI in 1VM	Regression	97.99%	100.0%	96.05%
CSI in 1VM	QNM	80.0%	80.0%	80.0%
CSI in 1VM	**ML**	**98.67%**	**100.0%**	**97.37%**
CDI in 1VM	Threshold	0.00%	0.00%	0.00%
CDI in 1VM	**Regression**	**89.59%**	**92.79%**	**86.6%**
CDI in 1VM	QNM	81.25%	100.0%	68.42%
CDI in 1VM	ML	87.46%	100.0%	77.71%
NDI in 2VM	Threshold	41.94%	46.43%	38.24%
NDI in 2VM	Regression	73.85%	77.42%	70.59%
NDI in 2VM	QNM	66.67%	66.67%	66.67%
NDI in 2VM	**ML**	**97.14%**	**94.44%**	**100.0%**
CSI in 2VM	Threshold	40.13%	89.85%	25.84%
CSI in 2VM	Regression	99.24%	99.27%	99.21%
CSI in 2VM	QNM	77.78%	77.78%	77.78%
CSI in 2VM	**ML**	**99.91%**	**99.82%**	**100.0%**
CDI in 2VM	Threshold	19.27%	86.45%	10.84%
CDI in 2VM	**Regression**	**91.5%**	**95.44%**	**87.87%**
CDI in 2VM	QNM	73.68%	87.5%	63.64%
CDI in 2VM	ML	87.97%	100.0%	78.52%

Table 7: Performance of ML vs. State-of-the-Art Models

the scenarios. The Regression approach overall has a significantly better performance compared to the Threshold and QNM in the CSI and CDI scenarios. However, the Regression model showed significant performance degradation in the NDI scenario with a F_1 score of 15.04 %, being on par with the Threshold model. Compared to the Regression model F_1 score in all the other scenarios in both VM deployments, this can be considered an outlier. The poor performance of the QNM model when detecting performance interference might be impacted by the lack of precision in estimating alpha and assuming ideal arrival rates and service time distributions. The ML approach outperforms all the models in each of the scenarios. The Regression model does outperform out ML approach in the CDI scenario by 2.13 % and 3.53 %. However, the ML approach is more consistent in performance with a high F1-score, Precision and Recall across all the scenarios, unlike the Regression approach. This demonstrates that the ML approach can consistently classify

Scenario	Approach	F_1 Score	Precision	Recall
NDI in 4VM	QNM	40.45%	34.05%	49.81%
NDI in 4VM	Regression	85.71%	81.03%	90.98%
NDI in 4VM	**ML**	**99.12%**	**99.25%**	**99.00%**
CSI in 4VM	QNM	42.5%	36.31%	51.24%
CSI in 4VM	Regression	90.14%	87.48%	92.98%
CSI in 4VM	**ML**	**99.17%**	**99.0%**	**99.26%**
CDI in 4VM	QNM	0%	0%	0%
CDI in 4VM	Regression	97.57%	95.82%	99.38%
CDI in 4VM	**ML**	**99.79%**	**99.79%**	**99.79%**

Table 8: Boutique subject to Interference

interference with high accuracy on a variety of different types of applications and deployments.

Table 8 show the F_1 score, Precision and Recall for the Boutique deployment experiments. Since the Threshold model performance was the lowest of the models, we omitted it from our experiments. The F_1 score for the ML approach has less variability than the regression method which shows that the ML approach is more consistently able to detect performance interference for large at-scale deployments. Our QNM model performed the worst when detecting interference for at-scale deployments. Specifically, the CDI scenario demonstrates the limitation of the QNM approach and it's inflexibility in handing variability and incomplete metrics which caused the QNM approach unable to classify interference. Overall our ML approach outperforms the other approaches from 2.22% up to 13.41%.

Scenario	Best ML Model	F_1 Score	Precision	Recall
NDI in 1VM	GBM	81.64%	100.0%	68.98%
NDI in 2VM	Ensemble	97.14%	94.44%	100.0%
CSI in 1VM	Ensemble	98.67%	100.0%	97.37%
CSI in 2VM	Ensemble	99.91%	99.82%	100.0%
CDI in 1VM	GBM	87.46%	100.0%	77.71%
CDI in 2VM	GBM	87.97%	100.0%	78.52%

Table 9: Top Performing ML Model Per Scenario

Interference Base Utilization	Interference Step Size	F_1 Score	Precision	Recall
20 %	10 %	97.95%	100%	96%
10 %	6 %	91.30%	100%	84%
6 %	2 %	89.36%	91.30%	87.50%

Table 10: ML Accuracy with Different Interference Step Size

As the ML approach has the best overall performance, we then investigated the step size of the workload and it's effect on the performance of the ML approach. Since the same workload is used across all the experiments, we investigate how the F_1 score of the ML approach are affected with different workload step sizes. Table 10 shows differing workloads of the interfering application with the Base Utilization as the starting point of the interfering application and its respective step size. A higher base utilization and step size increases causes the ML model to classify the interference with more ease. As the step size of the workload is reduced to smaller

Env	Training / Runtime Scenarios	Approach	F_1 Score	Precision	Recall
1VM	NDI / CSI	Threshold	10.3%	94.15%	5.45%
1VM	NDI / CSI	Regression	21.91%	1.0%	12.31%
1VM	**NDI / CSI**	**ML**	**75.77%**	**99.26%**	**61.27%**
1VM	NDI / CDI	Threshold	0.0%	0.0%	0.0%
1VM	NDI / CDI	Regression	53.8%	1.0%	36.8%
1VM	**NDI / CDI**	**ML**	**82.77%**	**78.18%**	**87.93%**
1VM	CSI / CDI	Threshold	0.0%	0.0%	0.0%
1VM	CSI / CDI	Regression	87.46%	83.68%	91.61%
1VM	**CSI / CDI**	**ML**	**90.21%**	**82.17%**	**100.0%**
2VM	NDI / CSI	Threshold	39.88%	88.04%	25.78%
2VM	**NDI / CSI**	**Regression**	**91.81%**	**93.91%**	**89.8%**
2VM	NDI / CSI	ML	89.53%	98.5%	82.1%
2VM	NDI / CDI	Threshold	17.8%	85.43%	9.93%
2VM	NDI / CDI	Regression	78.63%	93.3%	67.95%
2VM	**NDI / CDI**	**ML**	**82.62%**	**93.86%**	**73.79%**
2VM	CSI / CDI	Threshold	17.8%	85.43%	9.93%
2VM	CSI / CDI	Regression	90.98%	84.41%	98.67%
2VM	**CSI / CDI**	**ML**	**94.37%**	**91.68%**	**97.22%**

Table 11: Evaluation of Training/Runtime Interference

values, the F1-score decreases. However even at very minimum step size increases of 1.5 - 3 %, the ML approach has an 89.36 % F1-Score with relatively high precision and recall when classifying interference.

Finally, we report the ML models with highest F_1 Score for both the single and dual VM experiments in Table 9. Scenarios represented in this table are described by the number of VMs in the environment as well as the characteristics of interfering application. As seen in the table, GBM models performs best for interference detection in all single and dual VM CSI scenarios. In the dual VM NDI scenario, a stacked ensemble model performs best. In all CDI scenarios, a stacked ensemble model also performed the best.

RQ-2: We evaluated interference detection techniques on different deployment and interference types where the interfering load is the same in the training and runtime phases. ML models outperform logistic regression, QNM and simple threshold techniques in each of our evaluation experiments in F_1 score by at least 0.67% and at most 23.29%. The high ML performance is maintained at-scale deployments and we can conclude ML models can better handle the variability of the cloud.

5.3 RQ-3 Results: When Interfering Workloads for Model Training and Deployment are Different

Cloud environments of scale are subject to frequent change. Containerized microservices may be co-located and scaling actions may introduce additional containers that in turn stress the underlying VMs. It is impractical to train an interference model for every possible deployment combination. Accordingly, we attempt to construct a ML model that performs well when our target application is subject to a different interference scenario in the runtime phase as opposed to its training phase.

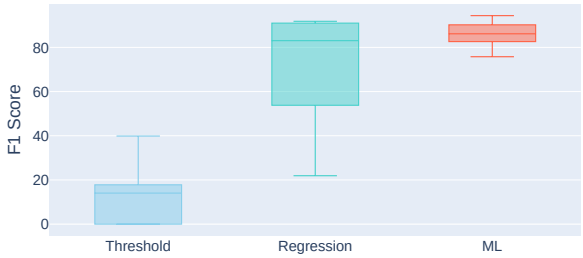


Figure 5: Evaluation for Unknown Interference

Training / Runtime Scenarios	Approach	F_1 Score	Precision	Recall
NDI / CSI	Regression	86.19%	88.54%	83.96%
NDI / CSI	**ML**	**98.44%**	**98.69%**	**98.18%**
NDI / CDI	Regression	86.05%	99.19%	75.99%
NDI / CDI	**ML**	**99.10%**	**99.50%**	**98.72%**
CSI / CDI	Regression	80.18%	99.57%	67.12%
CSI / CDI	**ML**	**97.64%**	**99.48%**	**95.87%**
NDI + CDI / CSI	Regression	83.20%	94.25%	74.46%
NDI + CDI / CSI	**ML**	**97.11%**	**99.19%**	**95.12%**
NDI + CSI / CDI	Regression	80.72%	99.58%	67.86%
NDI + CSI / CDI	**ML**	**99.65%**	**99.47%**	**99.83%**

Table 12: Boutique Training/Runtime Interference

Table 11 and Figure 5 shows our results from the experiments conducted in Section 4.4.4. Table 11 denotes the interference scenario encountered at training time as well as the interfering scenario present at runtime. In this way the ML models were evaluated in scenarios where the interference scenario is unknown and previously unseen. Notably, ML outperformed state-of-the-art methods in both single VM and dual VM environments. In each scenario, ML obtained the highest F_1 score when compared to state-of-the-art techniques. As seen in Figure 5, the ML approach consistently able to classify interference and significantly outperformed the ML and Regression. We performed the ANOVA test on the data to to confirm the significant differences between the approaches (f-value = 24.650731, p-value = 0.000018).

It's also notable that the models trained in CSI scenarios and tested at runtime with CDI interference resulted in a better F_1 score than the models with NDI interference at training time. Models where the interference scenario is the same at training and runtime perform substantially better over their counterparts where interference scenarios are different. However, using a model trained for use with different interference scenarios does not suffer from long pre-deployment or training phases.

For the large at-scale deployment, we compared the highest performing approaches in the previous experiments. When the interfering application is unknown with large at-scale deployments, Table 12 shows the Regression approaches begin to struggle in detecting the performance interference. Our ML approach outperforms the F_1 score of the regression up to 20.57%.

RQ-3: We evaluated ML versus the state-of-the-art interference detection techniques where the interference scenario is different in the training and runtime phases. The ML models outperform state-of-the-art techniques in F_1 score by at least 2.75 % and at most 53.86 % for all but one interference types and different deployments. We can conclude that training with interfering applications yields better performance than if we train with random interference (NDI with stress-ng). The performance of the ML approach (F_1 score), although lower than in the case of RQ-2, is high enough and scales well so it can be applied in production.

6 THREATS TO VALIDITY

We identify threats to validity of our work as per Wohlin et al [37]. We note as an external threat that our ML model training approach might not detect interference from unknown interfering applications well if the resource utilization signature of the benchmark interfering application is significantly different from the interfering application used at runtime. If the interfering application used in the training phase and the interfering application at runtime stresses different VM-level resources, our technique may be unable to classify interference. Furthermore, as another external threat, in multi-VM deployment strategies frequently used for microservice deployment, if the VM characteristics at runtime change relative to what was used in the training phase, our ML models may not be successful and will need to be re-trained.

7 CONCLUSIONS AND FUTURE WORK

In this work, we propose a runtime performance interference detection technique that leverages Machine Learning. Our ML models are trained on microservice resource utilization metrics subject to varying environments and different interfering applications. Our approach does not require expensive service instrumentation and does not pose prohibitive monitoring overhead. Our proposed technique is effective in detecting performance interference for a realistic microservice benchmark running on the EC2 cloud platform and also outperforms baseline and state-of-the-art interference detecting techniques. Our ML models are also effectively used in varying cloud environments where the interference characteristics change at runtime. In the future, we plan to investigate model maintenance over time with respect to an application's behavioural drift through continuous learning or time series techniques. Furthermore, we will integrate our ML Models into an AIOps platform that takes deployment actions to mitigate performance interference.

REFERENCES

[1] [Online]. Acme Air. https://github.com/acmeair
[2] [Online]. Air Quality Monitor. https://github.com/jlofw/air-quality-monitor
[3] [Online]. *Amazon Web Services.* https://aws.amazon.com/
[4] [Online]. *Google Cloud.* https://cloud.google.com/
[5] [Online]. *Locust.* https://locust.io/
[6] [Online]. *Online Boutique.* https://github.com/GoogleCloudPlatform/microservices-demo
[7] [Online]. *Production-Grade Container Orchestration.* https://kubernetes.io/
[8] [Online]. *Prometheus.* https://prometheus.io/
[9] [Online]. *Stress-ng.* https://kernel.ubuntu.com/~cking/stress-ng/
[10] Sandip Agarwala, Yuan Chen, Dejan Milojicic, and Karsten Schwan. 2006. QMON: QoS-and utility-aware monitoring in enterprise systems. In *2006 IEEE International Conference on Autonomic Computing*. IEEE, 124–133.
[11] Yasaman Amannejad, Diwakar Krishnamurthy, and Behrouz Far. 2015. Detecting performance interference in cloud-based web services. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 423–431.
[12] Alexandru Baluta, Joydeep Mukherjee, and Marin Litoiu. 2022. Machine Learning based Interference Modelling in Cloud-Native Applications. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 125–132.
[13] R.J. Boucherie and N.M. van Dijk. 2011. *Queueing Networks: A Fundamental Approach*. Springer US. https://books.google.ca/books?id=C98YswEACAAJ
[14] Eli Brookner. 1998. *Tracking and Kalman filtering made easy*. Wiley New York.
[15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 2755–2763.
[16] Ruoyu Gao and Zhen Ming Jiang. 2017. An exploratory study on assessing the impact of environment variations on the results of load tests. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 379–390.
[17] Surya Kant Garg and J Lakshmi. 2017. Workload performance and interference on containers. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 1–6.
[18] Vojtěch Horký, Jaroslav Kotrč, Peter Libič, and Petr Tůma. 2016. Analysis of Overhead in Dynamic Java Performance Monitoring. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2851553.2851569
[19] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-Hosted Web Applications. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 469–478. https://doi.org/10.1145/3038912.3052649
[20] Devki Nandan Jha, Saurabh Garg, Prem Prakash Jayaraman, Rajkumar Buyya, Zheng Li, and Rajiv Ranjan. 2018. A holistic evaluation of docker containers for interfering microservices. In *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 33–40.
[21] Kartik Joshi, Arun Raj, and Dharanipragada Janakiram. 2017. Sherlock: Lightweight detection of performance interference in containerized cloud services. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 522–530.
[22] Peng Kang and Palden Lama. 2020. Robust Resource Scaling of Containerized Microservices with Probabilistic Machine learning. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 122–131.
[23] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. 2007. An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 200–209.
[24] Erin LeDell and Sebastien Poirier. 2020. H2O AutoML: Scalable Automatic Machine Learning. *7th ICML Workshop on Automated Machine Learning (AutoML)* (July 2020). https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf
[25] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014).
[26] David Mosberger and Tai Jin. 1998. Httperf—a Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev.* 26, 3 (Dec. 1998), 31–37. https://doi.org/10.1145/306225.306235
[27] Joydeep Mukherjee, Alexandru Baluta, Marin Litoiu, and Diwakar Krishnamurthy. 2020. RAD: Detecting Performance Anomalies in Cloud-based Web Services. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE,

493–501.

[28] Joydeep Mukherjee and Diwakar Krishnamurthy. 2018. Subscriber-driven cloud interference mitigation for network services. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–6.

[29] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA) *(USENIX ATC'13)*. USENIX Association, USA, 219–230.

[30] Indrani Paul, Sudhakar Yalamanchili, and Lizy K John. 2012. Performance impact of virtual machine placement in a datacenter. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*. IEEE, 424–431.

[31] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. 2012. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*. 285–294. https://doi.org/10.1109/ICDCS.2012.65

[32] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 847–855. https://doi.org/10.1145/2487575.2487629

[33] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE, 1–10.

[34] Yohei Ueda and Moriyoshi Ohara. 2017. Performance competitiveness of a statically compiled language for server-side Web applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–22.

[35] Fotios Voutsas, John Violos, and Aris Leivadeas. 2023. Filtering alerts on cloud monitoring systems. In *2023 IEEE International Conference on Joint Cloud Computing (JCC)*. IEEE, 34–37.

[36] Sa Wang, Wenbo Zhang, Tao Wang, Chunyang Ye, and Tao Huang. 2015. Vmon: Monitoring and quantifying virtual machine interference via hardware performance counter. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 2. IEEE, 399–408.

[37] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen. 2000. *Experimentation in Software Engineering*. Kluwer Academic Publishers.

[38] Tao Zheng, C. Murray Woodside, and Marin Litoiu. 2008. Performance Model Estimation and Tracking Using Optimal Filters. *IEEE Transactions on Software Engineering* 34, 3 (2008), 391–406. https://doi.org/10.1109/TSE.2008.30

Author Index

Alt, Lukas 71
Alves, Ruan 258
Babar, Mohammad Fakhruddin ..211
Baluta, Alexandru 286
Bandamudi, Likhith 201
Basso, Matteo 14
Binder, Walter 14
Casale, Giuliano 223
Challa, Prajwal 107, 267
Cortellessa, Vittorio 48
Das, Saikat Kumar..... 25
De Matteis, Tiziano 166
Di Menna, Federico 48
Doekemeijer, Krijn 154
Domaschka, Jörg 60
Ertl, Otmar 2
Ezzati-Jivan, Naser 37
Foster, Ian 190
Funda, Christoph 224
Ganu, Hrishikesh V. 25
Geldenhuys, Morgan K..... 130, 142
German, Reinhard 224
Gupta, Pranjal 119
Hackenberg, Daniel 95
Hanindhito, Bagus 178
Hasan, Monowar 211
Hassan, Md Mahadi 211
Henning, Sören 2
Herpel, Thomas 224
Hielscher, Kai-Steffen Jens 224
Hillston, Jane 118
Huye, Darby 276
Ilsche, Thomas 71, 95
Iosup, Alexandru 166
Jiang, Song 107, 267
Jiang, Zhen Ming 286
John, Lizy K. 178
Kao, Odej 130, 142
Karmaker Santu, Shubhra Kanti 211
Kettimuthu, Rajkumar 190
Khazaei, Hamzeh 236
Khosravi Tabrizi, Amirmahdi 37
Kozhokanova, Anara 71
Krasznahorkay, Attila 83
Kunde, Shruti 201
Leichtfried, Michael 2
Lira, Anna 258
Litoiu, Marin 119, 236, 286
Liu, Lan 276
Liu, Zhengchun 190
Löff, Júnior 14
Ma, Xiaolong 190
Manterola Lasa, Aratz 166
Mendes, Mariana 258
Mishra, Mayank 201
Mohammed, Manzoor 1
Mohapatra, Prateeti 119
Morais, Fabio 258
Mueller, Matthias S. 71
Mukherjee, Joydeep 286
Nagar, Seema 119
Nardelli, Matteo 247
Pandey, Shrey 25
Papka, Michael E. 190
Pereira, Thiago Emmanuel 258
Pfister, Benjamin J. J..... 130
Pimentel, Andy D..... 83
Rabiser, Rick 2
Ramalho, João 258
Rasolroveicy, Mohammadreza ... 119
Ren, Zebin 154
Rosà, Andrea 14
Rouf, Raphael 119
Rouf, Yar 286
Russo Russo, Gabriele 247
Salzburger, Andreas 83
Sambasivan, Raja R. 276
Sarkar, Souvika 211
Scheinert, Dominik 130, 142
Schiavio, Filippo 14
Schöne, Robert 95
Singh, Ravi Kumar..... 201
Singh, Satyajeet 25
Singhal, Rekha 201
Song, Baohua 267
Swatman, Stephen Nicholas..... 83
Talluri, Sacheendra 166
Tehrany, Nick 154
Terboven, Christian 71
Tetreault, Francois 37
Thamsen, Lauritz 142
Torabi, Hoda 236
Traini, Luca 48
Trivedi, Animesh 154
Varbanescu, Ana-Lucia 83
Velten, Markus 95
Vogel, Adriano 2
Volpert, Simon 60
Watts, Ian 119
Wesner, Stefan 60
Winkelhofer, Sascha 60
Yan, Feng 190
Yang, Lei 190
Zhao, Xingsheng 107
Zhong, Chen 107