Lukewarm Serverless Functions: Characterization and Optimization

Citation for published version:
Schall, D, Margaritov, A, Ustiugov, D, Sandberg, A & Grot, B 2022, Lukewarm Serverless Functions: Characterization and Optimization. in *Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA) 2022.* ACM, pp. 757-770, The 49th International Symposium on Computer Architecture (ISCA), New York City, United States, 18/06/22. https://doi.org/10.1145/3470496.3527390

Digital Object Identifier (DOI):
10.1145/3470496.3527390

Link:
Link to publication record in Edinburgh Research Explorer

Published In:
Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA) 2022

Lukewarm Serverless Functions: Characterization and Optimization

David Schall
d.h.schall@sms.ed.ac.uk
University of Edinburgh
Edinburgh, United Kingdom

Artemiy Margaritov*
artemiy.margaritov@huawei.com
Turing Core, Huawei 2012 Labs
Edinburgh, United Kingdom

Dmitrii Ustiugov*
dmitrii.ustiugov@ed.ac.uk
ETH Zurich
Zurich, Switzerland

Andreas Sandberg
andreas.sandberg@arm.com
Arm Research
Cambridge, United Kingdom

Boris Grot
boris.grot@ed.ac.uk
University of Edinburgh
Edinburgh, United Kingdom

ABSTRACT

Serverless computing has emerged as a widely-used paradigm for running services in the cloud. In serverless, developers organize their applications as a set of functions, which are invoked on-demand in response to events, such as an HTTP request. To avoid long start-up delays of launching a new function instance, cloud providers tend to keep recently-triggered instances idle (or *warm*) for some time after the most recent invocation in anticipation of future invocations. Thus, at any given moment on a server, there may be thousands of warm instances of various functions whose executions are interleaved in time based on incoming invocations.

This paper observes that (1) there is a high degree of interleaving among warm instances on a given server; (2) the individual warm functions are invoked relatively infrequently, often at the granularity of seconds or minutes; and (3) many function invocations complete within a few milliseconds. Interleaved execution of rarely invoked functions on a server leads to thrashing of each function's microarchitectural state between invocations. Meanwhile, the short execution time of a function impedes amortization of the warm-up latency of the cache hierarchy, causing a 31-114% increase in CPI compared to execution with warm microarchitectural state. We identify on-chip misses for instructions as a major contributor to the performance loss. In response we propose Jukebox, a record-and-replay instruction prefetcher specifically designed for reducing the start-up latency of warm function instances. Jukebox requires just 32KB of metadata per function instance and boosts performance by an average of 18.7% for a wide range of functions, which translates into a corresponding throughput improvement.

CCS CONCEPTS

• **Computer systems organization → Architectures**; **Cloud computing**; *Processors and memory architectures*; **Architectures**; **Cloud computing**; *Processors and memory architectures*; • **Information systems → Computing platforms**; **Computing platforms**.

KEYWORDS

Serverless, characterization, microarchitecture, instruction prefetching

ACM Reference Format:
David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. 2022. Lukewarm Serverless Functions: Characterization and Optimization. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527390

1 INTRODUCTION

Serverless computing, also known as Function-as-a-Service (FaaS), has emerged as a popular cloud computing paradigm. In serverless, developers organize their applications as a set of functions, which are invoked on demand in response to external requests (e.g., clicks) or by other functions. Developers fully cede the management of resources used by their functions to the cloud provider, who spawns and shuts down function instances based on observed load. For developers, the serverless model offers two significant advantages: first, they do not need to worry about scalability of their applications, which are taken care of by the cloud provider; and secondly, they pay only for the time that their function is executing (i.e., per invocation). The latter contrasts sharply with traditional cloud deployments, including microservices, where a running instance incurs a cost to the developer regardless of whether it is processing requests or is idle.

Often, serverless functions execute fine-grained, short-running tasks [11, 12, 25]. Indeed, developers are incentivized to break down their applications into a collection of fine-grained functions to maximize elasticity, thus allowing different parts of business logic of the application to scale independently. Moreover, serverless providers, such as AWS Lambda and Google Cloud Functions, charge users for the maximum amount of memory that each of their functions consumes [5, 19], which further pushes the developers toward leaner,

*This work was done while the authors were at University of Edinburgh

ISCA '22, June 18–22, 2022, New York, NY, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527390

finer-grained functions. For instance, 70% of AWS Lambda functions are deployed with a 128–256MB memory limit [11]. In the case of interactive services, for which serverless is a common implementation model [15], the *end-to-end latency* must often meet an SLO target in the order of a few tens of milliseconds [20]; to do so, the individual functions may be expected to complete in a millisecond or less [25, 45, 54].

To avoid the long delays of booting a new function instance, cloud providers tend to keep recently-invoked instances alive (or *warm*, in serverless parlance) instead of shutting them down, in anticipation of additional invocations to that instance. Recent work has shown that Amazon, Google and Microsoft all keep recently-invoked function instances warm for at least several minutes and up to an hour [49]. The combination of small memory footprints for many functions, long keep-alive intervals enforced by cloud providers [49], and hundreds of gigabytes of memory in a representative server results in *thousands* of warm function instances residing on a typical cloud server [2]. The execution of these functions is interleaved in time based on invocation traffic, with many warm functions experiencing invocation inter-arrival rates measured in seconds or minutes [43] – an invocation rate that is relatively infrequent compared to their run time.

The high degree of co-residency and interleaving of serverless functions on a server, combined with short execution times (milliseconds or less) and relatively long inter-invocation intervals (seconds or minutes), mean that when a given warm function is invoked, it often finds none of its microarchitectural state on the core or in the cache hierarchy. Thus, while the function itself is memory-resident, the actual execution of the function is cold from the CPU perspective. We refer to this phenomenon as a *lukewarm* execution.

Our analysis reveals that lukewarm executions of functions result in a 31-114% performance degradation compared to executions with fully warmed up microarchitectural state (i.e., back-to-back executions of the same function on the same core). The reason for such a high performance degradation is that the short running time of the functions (typically on the order of a few milliseconds or less) is insufficient to amortize the warm-up time of microarchitectural structures. Using the Intel Top-Down performance analysis [52], we show that for 20 functions (including two distributed applications implemented as serverless workflows) the single largest source of performance loss (56% on average) is in the CPU front-end, specifically the on-chip misses for instructions.

Based on this finding, we examine the instruction footprints across multiple invocations of the same function and find significant commonality across invocations: 90% of all instruction cache blocks accessed by one invocation are also accessed by a subsequent invocation. We further find that the instruction footprints of individual invocations for the studied functions ranges from 300KB to over 800KB. With hundreds or thousands of co-running warm function instances on a serverless host, it is infeasible to keep the combined instruction footprints of all functions in processor caches.

Spurred by the observations above, we propose Jukebox, a record-and-replay instruction prefetcher for accelerating lukewarm serverless function executions. The idea of a record-and-replay instruction prefetcher is not new; indeed, prior works have proposed them for long-running server workloads by recording entire streams of instruction cache accesses or misses [16, 28, 29], requiring over 100KB of on-chip storage for metadata. In contrast to these works, Jukebox solves a different problem: how to accelerate short-running tasks that have no microarchitectural state or metadata on-chip. To accommodate thousands of warm functions, Jukebox stores its metadata in main memory using simple spatio-temporal compression designed for high coverage and low metadata redundancy. Our evaluation shows that 32KB of metadata (i.e., eight OS pages) is sufficient for high efficacy; for a thousand warm function instances, the required metadata cost is a mere 32MB. Jukebox requires simple hardware support and a negligible amount of on-chip state with no modifications to the processor caches. Our full-system simulation of Jukebox reveals that it speeds-up execution of lukewarm functions by 18.7%, on average.

To summarize, our contributions are as follows:

- We show that a high degree of interleaving in the execution of warm serverless functions with short running times leads to obliteration of their on-chip microarchitectural working sets between invocations, resulting in 31-114% performance degradation relative to an execution with warm microarchitectural state.
- We perform a detailed Top-Down analysis of the causes of the performance loss in the interleaved setup and show that the largest fraction (56%) of extra execution cycles is attributed to fetch latency indicating a bottleneck in instruction delivery.
- We propose Jukebox, a record-and-replay instruction prefetcher that accelerates lukewarm function executions. Jukebox requires a small amount (32KB) of metadata in main memory per function instances (32MB for a thousand functions) and provides 18.7% performance improvement on lukewarm invocations, on average.

2 MOTIVATION

2.1 Serverless Workloads Characteristics

Serverless, also known as Function-as-a-Service (FaaS), has emerged as a new cloud programming paradigm in which the providers take complete responsibility of managing the cloud infrastructure leaving service developers to focus only on writing their business logic. In serverless, developers write their services as a set of stateless event-triggered tasks, called functions, which are invoked via HTTP requests. The providers spawn and tear down instances of each function on demand, following changes in the function invocation traffic.

Recent studies of AWS Lambda show that production deployments feature many short-running functions with a small memory footprint [11, 12]. For instance, 67% of Lambda@Edge functions complete within 20ms [12]. One reason for the prevalence of short-running functions is their frequent usage for implementation of interactive services. For example, Eismann et al. [15] found that the most common application domain for serverless functions is web services (33% out of 89 studied functions).

The demand for short functions continues to increase; for example, one of the AWS Lambda studies shows that the median function duration became 2× shorter in 2020 as compared to the median duration in 2019 [12]. In response to this trend, AWS Lambda decreased the billing granularity from 100ms to 1ms [5]. Lastly, both the AWS Lambda and Azure Functions demonstrate that >70% of functions have little memory footprint, allocating less than 300MB of memory [11, 43].

Despite the short running time of many function instances and their small memory footprints, cold-booting a function is a long-latency operation that can take hundreds of milliseconds in today's clouds [49, 50]. To avoid this latency in the critical path of function invocation, cloud providers tend to keep idle function instances alive (or *warm*) for 5-60 minutes [36–38, 49]. Although keeping function instances warm comes at a cost for the providers because users are billed only for the actual processing time of individual invocations, all major providers deploy this performance optimization.

With providers keeping function instances warm for 5-60 minutes, approximately 20-40% of all deployed functions have a warm instance when a request arrives, according to the study of Azure Functions [43]. The same study shows that fewer than 5% of all invocations have an inter-arrival time (IAT) of under a second. Thus, the IAT of a vast majority of invocations to warm instances lies in the range of 1 second to a few minutes.

2.2 Serverless Functions on a Cloud Server

As discussed above, many serverless functions have modest memory footprints and are kept warm for a number of minutes by the cloud provider to reduce the incidence of cold boots. With typical cloud servers today configured with hundreds of gigabytes of main memory [47], a thousand or more instances of warm functions may be resident on a server [4]. The warm instances tend to stay memory-resident as providers disable swapping to avoid the associated performance and security issues [48]. Meanwhile, many functions have short execution times of a few milliseconds or less, with invocations that are rare compared to their processing time.

The combination of these trends results in a high degree of function interleaving. Simplistically assuming a server, with instances of functions, whose invocation processing time is 1ms with a 1s inter-arrival time (IAT) for each instance, a thousand unrelated invocations will be interleaved between two invocations of the same function. In fact, function execution time may vary and their inter-arrival time distribution is not uniform, but with thousands of function instances kept warm on a host and typical inter-arrival times of seconds to minutes, a huge degree of interleaving is likely.

The problem that stems from such extensive interleaving is that a new invocation to a warm function instance is likely to find its on-chip microarchitectural state largely obliterated. Thus, a function instance that is warm from a runtime's perspective (i.e., has its state fully loaded in memory) faces a cold CPU, requiring the instance to fill all of the microarchitectural structures both in the core and throughout the cache hierarchy – a phenomenon we refer to as a *lukewarm* invocation. Lukewarm invocations pose a particularly acute problem for serverless functions with invocations times in the range of milliseconds, since the short execution times do not offer the opportunity to amortize the latency needed to warm up microarchitectural structures over a long execution period.

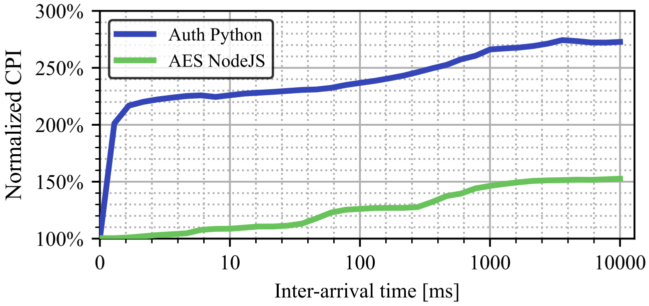


Figure 1: Effect of request inter-arrival time on the CPI of a given function on a high-occupancy server. CPI is normalized to back-to-back invocations.

To illustrate the problem, we study the performance of representative serverless functions, packaged as Docker containers, running on a modern server. Multiple instances of many function are kept warm on the server. The hardware setup and the functions are described in Sec. 4. Clients send invocation requests to the various instances maintaining a stable load on the server (around 50% of peak CPU load). In each experiment, one function instance is selected as a function-under-test (FUT). The invocation IAT for the FUT is fixed for the duration of the experiment. For each invocation, we sample a set of performance counters using *perf*. We then repeat the experiment with a different IAT for the FUT. Each experiment with IATs lower than 100ms was run for 3 minutes while the experiments with 100ms or longer IATs – for 10 minutes.

Figure 1 represents the cycles per instruction (CPI) for two representative FUTs: an authentication function written in Python and an AES encryption function written in NodeJS. For this experiment, we choose functions written in different languages to highlight the language-independent nature of the behavior. The figure clearly shows that increasing the invocation IAT tends to increase the CPI. The number of cycles spent per invocation of an authentication function increases by more than 2x and stabilizes at a 270% higher CPI with IAT of over 1 second. With the same IAT, the AES encryption function requires 150% more cycles per instruction compared to back-to-back execution. The reason why the CPI grows as the invocation IAT is increased is that the execution of numerous other instances between two invocations of the FUT thrashes all of the microarchitectural state on the CPU core where the FUT executes and throughout the cache hierarchy. Thus, when a new invocation to a FUT arrives after a long IAT, the FUT experiences a lukewarm execution, with poor performance.

2.3 Top-Down Analysis of Lukewarm Executions

To get a deeper understanding of the sources of performance loss in lukewarm executions, we study each of the 20 functions in our suite (Table 2) in two configurations. In the first configuration, the FUT is invoked repeatedly on the same core on an otherwise idle server – this yields the lowest possible execution time for the studied function as each invocation after the first one enjoys fully warmed up microarchitectural state and caches. We refer to this as

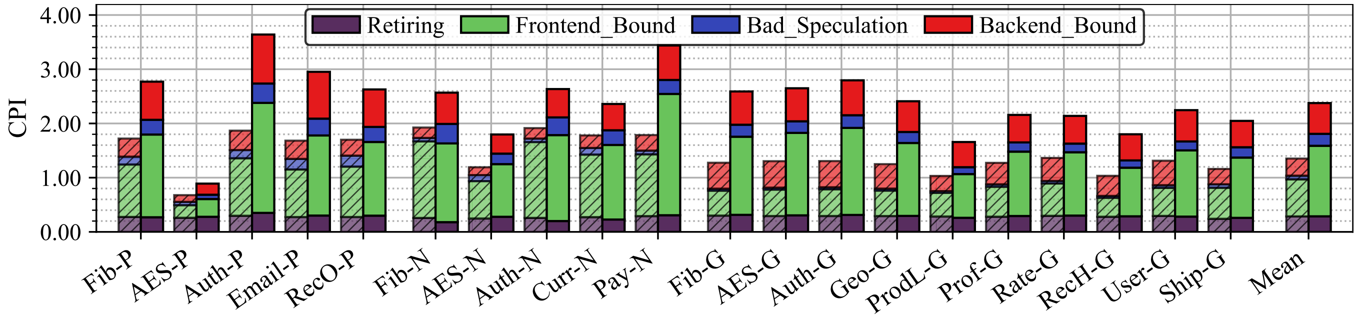


Figure 2: Top-Down CPI analysis of serverless functions. Striped bars: reference execution, solid bars: interleaved execution.

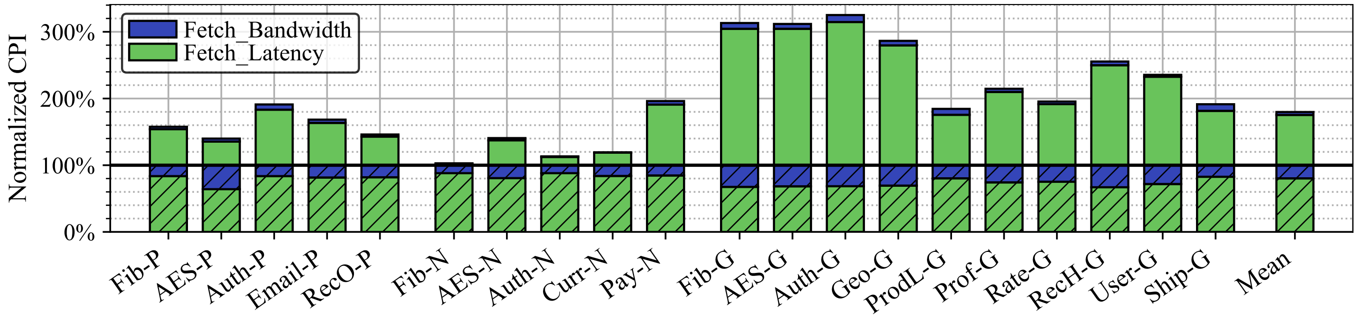


Figure 3: Top-Down CPI analysis of the front-end stall cycles. The striped portions show the reference execution, solid portions show the additional cycles due to interleaving. Normalized to the front-end portion of the CPI for the reference execution in Figure 2.

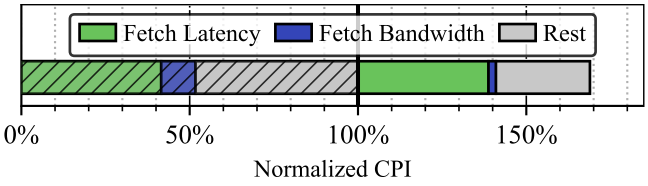


Figure 4: Average CPI in the interleaved setup normalized to the average CPI in the reference execution. The striped part represents CPI in the reference execution, the solid part – the extra CPI observed in the interleaved setup. ***Rest*** **includes all other cycle categories except the front-end stall.**

the *reference* execution. The second configuration runs a stressor after each invocation of the FUT.

To achieve the effect of interleaving with other functions, we use `stress-ng` [9] as a stressor and run it on the same core as the FUT in order to thrash caches and the core's microarchitectural state.

The performance degradation experienced by the functions in this configuration is similar to that of combining a high degree of interleaving with high IAT (Sec. 2.2).

Using the data collected from performance counters, we perform an analysis of each function's CPI stack using the Top-Down methodology [52]. Figure 2 shows the results of the analysis for the top-most level of the Top-Down tree, which classifies all pipeline slots into one of four categories: front-end bound (e.g., instruction cache and I-TLB misses), back-end bound (e.g., data cache misses, structural hazards for execution resources), bad speculation (e.g., branch mispredictions) and retiring. Note, the first three categories relate to microarchitectural bottlenecks, which should be minimized, and only the last one corresponds to useful work.

We make two observations based on the results presented in Figure 2. First, aggressive interleaving (modeled by the stressor in this experiment) has a detrimental effect on all functions, increasing their CPI by 31-114% (70% on average). Second, 51% and 55% of all cycles are classified as front-end stall cycles[1] in reference and interleaved execution, respectively. Moreover, on average, the front-end is responsible for 62% of all stall cycles in reference execution (65% in the interleaved execution). On 15 out of 20 studied functions, the front-end contributes to more than 50% of the extra stall cycles observed in the interleaved execution compared to reference execution. As a result, the front-end is the main source of stalls for the majority of the studied functions.

We next focus on the front-end stall cycles to understand the source(s) of the bottleneck. Following the Top-Down methodology, we classify the front-end stall cycles into two categories: fetch latency and fetch bandwidth. As shown in Figure 3, the vast majority

[1] In Top-Down, a stall cycle is defined as a CPU cycle in which the pipeline cannot make progress because at least one architectural component is fully utilized and cannot take additional work. However, we note that in an out-of-order architecture, other components can often make progress in the shadow of a pipeline stall. Furthermore, stalls can overlap with each other as well as with retiring.

of front-end stall cycles in both reference and interleaved executions are due to fetch latency. In the interleaved execution, fetch latency stall cycles increase by an average of 94% over the reference while fetch bandwidth stalls grow only by 22% on average.

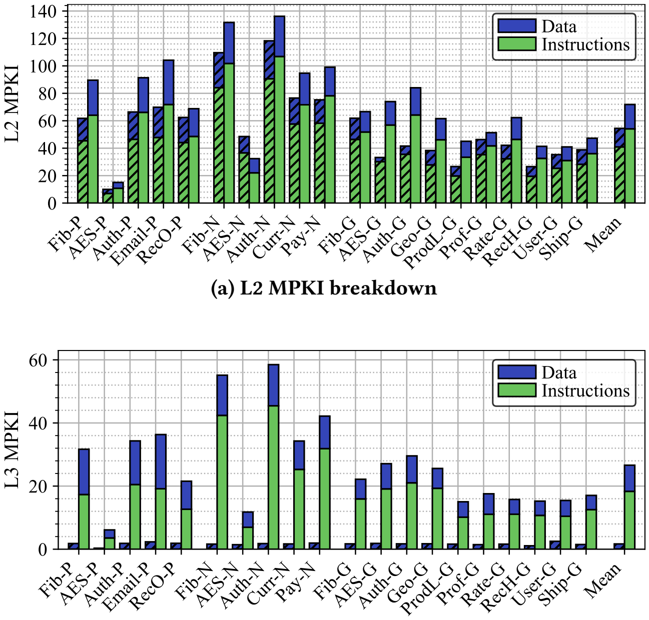
Figure 3 puts the front-end performance problem in focus by isolating the portion of the CPI due to front-end stalls from Figure 2 and breaking down the stall cycles into fetch latency and fetch bandwidth. Figure 4 focuses on the front-end related stalls portion of the *Mean* bar for the interleaved execution from Figure 2. To identify the source of the extra stall cycles in the interleaved setup, the CPI stack in Figure 4 is normalized to the reference execution. Our key observation is that with function interleaving, most of the extra stall cycles occur due to front-end inefficiencies. More specifically, the figures clearly point out fetch latency as a key performance bottleneck in the execution of serverless functions, responsible for 56% of all extra stall cycles in the interleaved setup, on average.

2.4 The Story of Cache Misses

To understand the source of fetch latency stalls, we examine instruction misses throughout the cache hierarchy and compare them to data misses. Noting that L1-I misses are consistently high in both reference and interleaved executions, which is expected in light of the findings above, we focus our study on L2 and L3 caches.

Figure 5a shows the L2 MPKI for both instruction and data references. We make several observations. First, miss rates are high for both reference and interleaved executions, with an average MPKI of 54 for the former and 72 for the latter. Second, we note that misses for instructions are more frequent than misses for data, which suggests that the instruction working set is larger than the data working set. Given that the in-order front-end can not overlap processing of instruction cache misses while the out-of-order back-end often can hide some of the latency of data cache misses, it is not surprising that the front-end is a more significant contributor to total stall cycles than the back-end (Figure 2). Lastly, we note that the high L2 miss rates for the reference setup can, at least partially, be attributed to a relatively small L2 of 256KB in the evaluated server as compared to the large instruction footprints of the studied functions, as discussed in Sec. 2.5. Meanwhile, in the interleaved setup, L2 miss rates are expected to be high due to the cold cache in the wake of interleaving.

We next shift our attention to the LLC (i.e., L3 cache), whose MPKI is shown in Figure 5b. The striking trend in the figure is that reference executions have no LLC misses for instructions and very few misses for data, which is explained by the fact the working sets of the studied functions easily fit in the 25MB LLC of the evaluated server and that the back-to-back invocation pattern facilitates LLC residency. Meanwhile, the LLC misses for interleaved executions exceeds 10 MPKI, with several functions experiencing MPKIs in excess of 40. The majority of the misses are for instructions, which explains the high fraction of front-end related stall cycles in in the interleaved setup: each L1-I miss to the main memory leaves the core front-end starved of instructions for an extended period of time.



(a) L2 MPKI breakdown

(b) L3 MPKI breakdown

Figure 5: MPKI breakdowns of level 2 (a) and level 3 (b) caches. The striped bars (on the left in each pair of bars): reference execution, solid bars: interleaved execution.

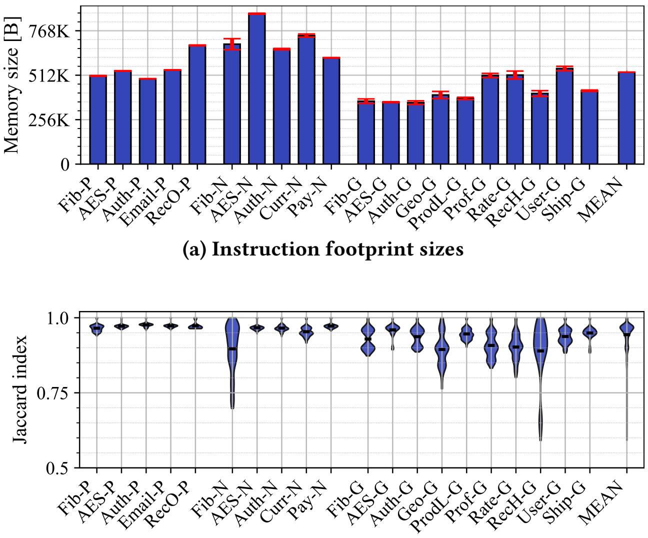
2.5 Instructions in Focus

Having identified long-latency misses for instructions as a key performance bottleneck in the execution of warm serverless functions, we next study the instruction footprints of individual invocations of functions from our suite. For this analysis, we use the gem5 full-system simulator and run the same containerized function instances as we do on real hardware. Sec. 4.2 details our simulation setup. We load the warmed-up system state from a checkpoint and execute each function 25 times. For each execution, we trace L1-I accesses, at cache block granularity, eliminating any repeated cache block addresses from the trace to get the set of unique instruction blocks accessed per invocation. We record both user and kernel instruction accesses.

Figure 6a shows the instruction footprint sizes of individual invocations for the studied functions in blue. Error bars indicate the range of recorded values for a given function. We observe that with only a few exceptions, the instruction footprints of individual invocations ranges from just over 300KB to around 800KB with notably low variance for the vast majority of functions.

Lastly, we study the *commonality* in the instruction footprints across invocations. For this study, we compare the footprints (in terms of cache block addresses) of each invocation with that of each other 24 invocations, for a total of 300 pair comparisons. For each pair of invocations, we compute *commonality* as the Jaccard index [23] which is defined as the ratio between the intersection and the union of unique cache block addresses that belong to the instruction footprint of the pair of invocations.

Results of the commonality study are shown in Figure 6b. For all but three functions the mean of commonality among the 300



(a) Instruction footprint sizes

(b) Instruction commonality

Figure 6: (a) Instruction footprint sizes of individual invocations. (b) The distribution of Jaccard indices calculated from the 25 function invocations. The Jaccard index ranges from 0 (nothing in common) to 1 (identical), a higher value suggests that record and replay to be more likely to prefetch that function.

compared pairs of invocations exceeds 90%. Their commonality distributions ranges above 80% and half of them do not even subceed 90%. Only two functions show outliers in their 300 compared pairs that have footprints with a commonality less than 75%. We thus conclude that multiple invocations of the same function have high commonality in their individual instruction footprints.

2.6 Summary

Many serverless functions have small memory footprints, short execution times and comparatively long IATs. Thousands of such function instances may run simultaneously on a cloud server, resulting in a very high degree of interleaving between two invocations of the same function. The interleaving obliterates on-chip microarchitectural state of the functions, resulting in a *lukewarm* execution with cold caches and a cold core. Lukewarm executions carry an average performance penalty of 70% compared to an execution with fully warmed microarchitectural state. The single biggest source of performance overhead in a lukewarm execution is the core front-end, particularly fetch latency, which constitutes 56% of all additional stall cycles, on average. Frequent L2 and LLC misses for instructions are a key contributor to the high fetch latency. The high on-chip miss rates for instructions in interleaved invocations of serverless functions can be explained by their large instruction footprints of individual invocations, commonly in the range of 300KB to over 800KB. At the same time, there exists high commonality in the instruction footprints of different invocations of the same function.

3 DESIGN

3.1 Design Overview

Based on the insights of Sec. 2, we introduce Jukebox, an instruction prefetcher specifically designed to accelerate *lukewarm* executions of serverless functions. Jukebox exploits instruction commonality the high commonality of instruction blocks across invocations by recording the working set of one invocation and replaying it whenever a new invocation to the same instance arrives.

Compared to state-of-the-art instruction prefetchers [7, 16, 28, 33] which target the L1-I, a unique feature of Jukebox is that it prefetches into the L2. This choice is motivated by two observations. First, the instruction footprints of individual invocations of the containerized functions generally stay within 800KB, a value much higher than a typical L1-I capacity of 32–64KB. However, such instruction footprints fit comfortably within the L2 capacities of today's server processors, including Intel Skylake and later [1], Amazon Graviton 2 [46], and the upcoming AMD Zen 4 [40], all of which have L2 caches of 1MB. Secondly, prefetching into the large L2 significantly simplifies the prefetcher's design, since it avoids the need to place prefetches into the small L1-I. With a small cache as a prefetch target, it is essential that prefetches arrive just in time to avoid being evicted (if they arrive too early) or not being useful (if late). With L2 as the prefetch target, an aggressive prefetcher can simply fill it at the start of execution and expect instructions to not be evicted in the duration of a short-running function. Prefetching into the L2 does sacrifice some performance compared to prefetching into the L1-I; however, since the latency of an L2 hit is approximately 10 cycles, while an LLC hit is typically over 30 cycles and an LLC miss is hundreds of cycles, the bulk of the opportunity in reducing stalls in a serverless environment comes from avoiding L2 misses. While Jukebox replays prefetches into the L2, its record logic sits at the L1-I, which enables recording of *virtual* addresses of instruction cache misses. Operating on virtual addresses is essential for the prefetcher to work well with the virtual memory subsystem and not be impacted by, for instance, page migrations due to memory compaction [26].

We next discuss details of Jukebox, whose operation consists of two distinct phases: *record* and *replay*. The record phase begins as soon as the container running the function has been launched. The replay phase is triggered when the OS resumes the process of a function that has been suspended waiting for new invocations. Both record and replay phases are initiated by the OS initializing a pair of dedicated registers with a pointer to the memory region for Jukebox's metadata, similar to how the OS initializes the CR3 register holding the pointer to the root of the page table of a process.

3.2 Record

Jukebox records the stream of L2 misses for instructions using a spatio-temporal encoding that provides for a compact metadata footprint and facilitates timely prefetching. The main component tracking addresses to record is the *Code Region Reference Buffer (CRRB)*, a small fully-associative FIFO structure that is accessed using the virtual address of a code region. Each CRRB entry contains a pointer to a memory region *(region pointer)* and an *access vector* holding one bit per cache line within that region. The least significant bits of the pointer used to address individual bytes within the region are not included in a CRRB entry. The *region pointer* can address fixed-sized regions with size chosen at design time. We study a Jukebox configuration with a CRRB entry comprising of a 38-bit *region pointer* and a 16-bit *access vector* (assuming 48-bit virtual addresses and 64B cache lines), for a total of 54 bits per entry (see Sec. 5.1 for an analysis of preferred code region size).

Recording logic is outlined in Figure 7a. Upon an L1-I miss, the request is forwarded to the L2 as usual. On an L2 hit, the Jukebox record mechanism takes no action, effectively filtering all L2 hits. If there was a miss in the L2, when the miss finally returns to the L1-I, it is recorded by Jukebox. This is done by generating a lookup into the CRRB, where the virtual address of the corresponding code region is checked against the existing entries ①. The code region virtual address is generated by taking the most significant bits of the missed block's virtual address corresponding to a CRRB pointer (38 bits for the studied Jukebox design). If a matching entry is found, the prefetcher sets the n^{th} bit in the access vector of the found entry where n is the offset of the cache line within the code region. Otherwise, the oldest entry in the CRRB is evicted ②, and a new entry is allocated ③. The evicted entry is written to memory, optionally bypassing the cache hierarchy since on-chip reuse of the metadata is not expected.

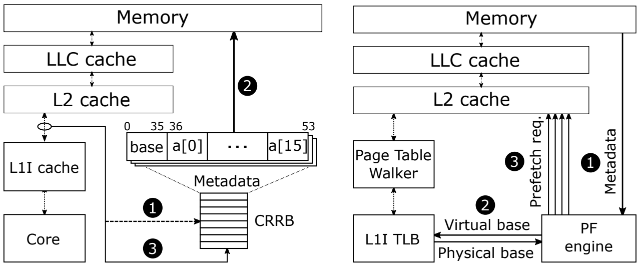
An entry evicted from the CRRB cannot be modified; thus, if an L2 instruction miss occurs to a region whose corresponding entry has already been pushed to memory, a new entry for the same region is created in the CRRB. As a result, a given code region might appear multiple times in the trace recorded by Jukebox. The effect of this design choice is that it increases the metadata footprint of Jukebox but simplifies the design, since evicted entries do not need to be brought back from memory.

There are two possible options for determining whether an L1-I miss also missed in the L2. The first option is to propagate the result of the L2 tag check back to the L1-I using a dedicated 1-bit signal. The second option is to measure the delay of an outstanding L1-I request and compare that to the expected L2 hit latency (e.g., using a timer at the L1-I MSHR [27].) Jukebox can work equally with either of these options.

The FIFO order of the entries in the recorded metadata directly encodes the temporal order of accesses at the chosen granularity. That is, the first metadata entry written to memory will encode the addresses of the cache lines in the first code region accessed after the function was invoked. This organization allows Jukebox to prefetch the entries in approximately the same order they are likely to be accessed, thus improving timeliness at replay time.

Note that because the recording is done in a region-based manner, the replay stream will first prefetch all of the indicated cache blocks from one code region before moving on to the next region. As a result, some reordering of individual cache blocks will happen at prefetch time compared to recording. However, recording at the region granularity enables a small metadata footprint, makes better use of address translation resources (a single lookup for all the blocks in a region), and reduces the number of DRAM activations due to spatial locality within a page.

The record phase is triggered by the OS by programming a pair of architecturally-exposed registers containing the base and limit of the metadata storage. The limit register is optional and lets the OS control the amount of metadata stored per function instance process.



(a) Jukebox record **(b) Jukebox replay**

Figure 7: Jukebox prefetcher overview.

3.3 Replay

The replay phase in Jukebox is triggered by the OS upon receiving a new function invocation. The OS triggers the replay by programming a pair of base and limit registers, similarly to how recording is triggered. The replay can be triggered by the OS's scheduler whenever the function instance thread is assigned to a core to process an invocation.

The replay phase is outlined in Figure 7b. The prefetch engine starts the replay phase by issuing sequential reads starting from the beginning of the metadata region ①, reading the metadata in the same order it was written to memory. This enables prefetching of instruction cache blocks in the same temporal order as was recorded, albeit at page granularity.

The metadata entries are prefetched into a small FIFO inside the prefetch logic. Once the metadata entry is returned from memory, the prefetcher passes the base address of the code region to the I-TLB ②, triggering address translation like a normal code request. This serves two purposes: first, it ensures that Jukebox does not rely on physical addresses that may change as a consequence of normal OS activity such as paging or memory compaction. Second, it effectively pre-populates the TLB with translations for code pages. As soon as the physical base address of the code region is known, using the information from the entry's access vector, the prefetch engine reconstructs full addresses of each of the accessed cache lines within the code region, and enqueues them in the L2 prefetch queue ③.

Once prefetch requests for all of the cache blocks encoded in an entry's access vector have been launched, the entry is retired from the FIFO. The next set of entries is fetched using a single 64B cache line read once the equivalent of 64B of data have been consumed from the FIFO.

3.4 Discussion

3.4.1 Metadata memory management. Upon a function instance's start (i.e., first invocation received by the host), the OS allocates two memory regions for the function instance process, each of which is contiguous in the physical space. These regions are used for bookkeeping of the Jukebox metadata of the function instance process. The OS associates the physical addresses of the two buffers with the PID of the function instance process. For example, in Linux, the addresses of the buffers can be stored in `task_struct`. Upon an invocation of a function, as part of assigning the function instance process to a core for execution, the scheduler consults the instance's `task_struct` and write addresses of the buffers to the registers that define where the Jukebox metadata is written (at record) and where the metadata is fetched from (at replay). Once the invocation completes and the function instance process is descheduled, the values of buffer pointers are saved in the `task_struct`. A subsequent invocation received by the same instance thus replays the metadata from the memory region that has been written by the previous invocation. Operating with the metadata buffers using physical addresses avoids the need for address translation while fetching/recording metadata which (1) improves the timeliness of Jukebox prefetches and (2) does not cause contention for TLBs and hardware page walkers.

3.4.2 Virtualization. Under virtualization, the guest OS triggers Jukebox record and replay. Jukebox metadata is stored in guest physical memory, i.e., as a part of the virtual machine state. Hence, in addition to lukewarm execution, Jukebox can accelerate the lengthy cold boots of serverless instances provided that a function snapshotting technique [13, 44, 50] is used and that the Jukebox metadata has been recorded before taking the snapshot.

3.4.3 Enabling Jukebox. Jukebox can be enabled for a particular thread upon its creation, similar to choosing a thread's scheduling priority by setting the corresponding attribute of the created thread [39]. For example, when the serverless runtime of a function instance starts a gRPC/HTTP server with a number of worker threads (or when spawning them at run time), it can enable Jukebox by setting the corresponding attribute before making a `pthread_create` [34] system call.

3.4.4 Generality. While Jukebox is particularly beneficial for lukewarm functions, it can accelerate start-up times for any memory-resident thread.

4 METHODOLOGY

4.1 Hardware Infrastructure

We perform the measurements on on an xl170 node in CloudLab [14] Utah datacenter, featuring a 2.4GHz 10-core Intel Broadwell CPU with 32KB L1-I, 32KB L1-D, 256KB L2, and 25MB LLC caches, 64GB DRAM. The node runs Ubuntu 20 with a stock Linux kernel v5.4. We disable SMT, following the production guidelines by AWS Lambda [2, 48]. Performance counters are collected using `linux perf` [30] during the entire execution of the target instance's container, thus capturing all kernel and user-level activity.

4.2 Simulation Infrastructure

To evaluate Jukebox, we use gem5 [8, 35], a full-system cycle-accurate simulator modeling a server-grade x86 CPU. Our simulated baseline is configured similar to Intel Skylake [1], with parameters of the modeled hardware summarized in Table 1. We also study a Broadwell-like configuration (Sec. 5.6), which resembles the real hardware platform used for the characterization studies in Sec. 2.3. We run an identical software stack in simulation as we do on real hardware; i.e., the same OS and the same containers running gRPC servers. Before performing the measurements, we boot the system in functional mode (KVM core) and execute 20000 invocations of each function, at which point we create a checkpoint of the system state. The checkpoints form the common starting state for all subsequent experiments. For the experiments, we switch to cycle-accurate timing mode and simulate 20 invocations.[2]

Core	
Architecture:	Skylake-like, ISA: x86-64, Freq.: 2.6GHz
Fetch BW	16 bytes / cycle
BP Unit	LTAGE (16K gShare 4K bimodal) + BTB 8K entries
ROB	224 entries
LSQs	72 load + 56 store entries
Scheduler	97 entries
Registers	180 Int + 168 FP
Memory Hierarchy	
L1-I Cache	32KB, 64B line, 8-way set assoc., 4 cycles access latency, private, LRU, 10 MSHR
L1-D Cache	32KB, 64B line, 8-way set assoc., 12-cycle access latency, private, 10 MSHRs, LRU, next-line prefetcher
L2 Cache	1MB, 8-way set assoc., 36 cycles, private, LRU, 32 MSHRs
LLC	8MB, 16-way set assoc., 36 cycles access latency, shared, non-inclusive, 32 MSHRs, 64 store buffers
Memory	DDR4 2400MHz, RCD(14ns), RP(14ns), CL(14ns)
Jukebox	CRRB: 16 entries, Region size: 1KB, 32KB metadata size (16KB record + 16KB replay)

Table 1: Parameters of the simulated processor.

4.3 Workloads

In our experiments, we evaluate Jukebox using a large set of short-running serverless functions, developed to work with a number of runtimes (namely, Python, NodeJS and Go), as listed in Table 2. The functions were adopted from the Hotel Reservation application from the DeathStarBench suite of microservices [18], Google's Online Boutique application [21], AWS' authentication serverless functions [6], and AES encryption application from FunctionBench [31, 32]. Similarly to vHive [50], the state-of-the-art serverless experimentation framework, each function is deployed as a handle of a gRPC [22] server, which represents a function instance. Each function is deployed in a separate container.[3]

Several functions in our workload are written in NodeJS, a language which utilize just-in-time (JIT) compilation for code optimization. In order to avoid performance noise induced by the JIT

[2] Configurations and guidance on how to setup gem5 to run containerized, serverless workloads are made available at https://github.com/ease-lab/vSwarm-u

[3] All function codes have been released and made available for the research community at https://github.com/ease-lab/vSwarm

Function	Abbreviation	Function	Abbreviation
Hotel Reservation [18]		**Online Boutique** [21]	
Geo	Geo-G	Currency	Curr-N
Profile	Prof-G	Email	Email-P
Rate	Rate-G	Payment	Pay-N
Recommendation	RecH-G	ProductCatalog	ProdL-G
User	User-G	Shipping	Ship-G
Other [6, 31, 32]		Recommendation	RecO-P
Authentication	Auth-P/N/G		
Fibonacci	Fib-P/N/G		
AES encryption	AES-P/N/G		

Table 2: Serverless functions and their language runtimes (legend – P: Python, N: NodeJS, G: Go).

engine to ensure stable and reproducible results, we invoke each JIT’ed function 20000 times before starting measurements [51]. We empirically found that for our functions more invocations do not trigger further code optimization. For the hardware simulation, we generate the gem5 checkpoint after these initial invocations. We note that Jukebox is orthogonal to JIT compilation and can speedup execution of unoptimized code. Moreover, by recording each function execution for subsequent replay, Jukebox can trivially adapt to changes in the instruction working set induced by the JIT engine.

5 EVALUATION

5.1 Parameterizing Jukebox

Recall from Sec. 3.1 that Jukebox uses a spatio-temporal encoding that exploits locality in code accesses by organizing its metadata as a sequence of entries, each corresponding to a spatial region. Each entry contains the upper bits of the address of the region and a bit vector, with one bit per cache line. Larger regions require a longer bit vector but may allow for fewer entries given sufficiently high spatial locality in the code. The entries are created in the CRRB, which coalesces accesses to the same region before the entry is written to the in-memory metadata storage. A larger CRRB may allow more accesses to the same region to be coalesced before an entry is evicted, resulting in a smaller metadata footprint at the cost of more on-chip storage and higher access energy.

To find the preferred code region size and CRRB size, we measure the size of the metadata required to store *all* of the entries produced by the Jukebox recording logic while a function executes. We do this for a range of code region sizes, from 512B to 8KB, and three different CRRB buffer sizes: 8, 16 and 32 entries.

Figure 8 presents the results of the study for a 16-entry CRRB. For the majority of the workloads, the metadata size reaches a minimum with the code region size of 1KB, resulting in 9.6KB to 29.5KB of metadata storage. Our experiments with the two other CRRB sizes (not shown) reveal very similar trends and modest sensitivity to the size of the CRRB.

We next study the impact of limiting the size of Jukebox’s metadata storage on its efficiency. Since we found in Figure 8 the most space efficient code region size to be 1KB, we use this configuration and a 16-entry CRRB for this sensitivity study. Figure 9 shows the speedup Jukebox is able to achieved when constrained to various metadata storage capacities. In Figure 9, we plot only one representative function for each of the three implementation languages [4] together with the average across all 20 functions in our evaluation suite.

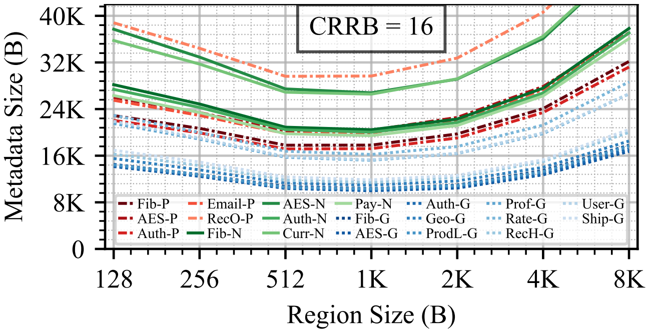


Figure 8: Sensitivity of Jukebox’s metadata size to the code region size with a 16-entry CRRB.

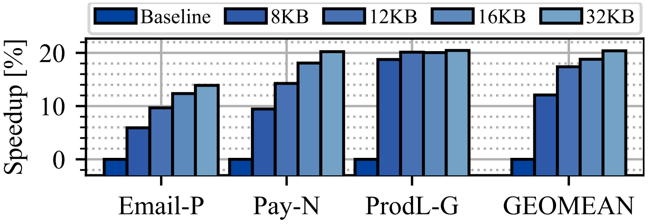


Figure 9: Speedup with Jukebox as a function of the size of its metadata storage compared to the baseline without Jukebox.

The figure shows that workloads with large working sets, e.g. Pay-N, tend to be more sensitive to the limited metadata size than workloads with small working sets, e.g. ProdL-N. This is expected given that the metadata represents a compressed form of a function’s working set. Note that Jukebox is designed to seamlessly extend to dynamic metadata sizes. For that a *metadata size* field needs to be added in the bookkeeping mechanism described in Sec. 3.4.1. When scheduling a thread, the OS sets up the size of the metadata of a function instance and assigns the addresses for record and replay metadata storage. We use the same size of metadata storage for all our workloads in further experiments.

As Figure 9 shows, on average, there is a little gain with increasing metadata storage beyond 16KB. Thus, unless stated otherwise, in the rest of the evaluation we use a Jukebox configuration with 16KB metadata storage, 1KB code-region size and 16-entry CRRB.

5.2 Performance

Figure 10 presents the main result of the evaluation. We compare three configurations: (1) the baseline, which represents a high degree of function interleaving; (2) Jukebox applied to the baseline setup; and (3) Perfect I-cache which draws the maximum opportunity without any instruction misses. The baseline (1) is modeled by flushing all microarchitectural state in-between function invocations. For (3), we use an infinite-sized L1-I cache that maintains

[4] We found that the language in which the function is written is the single biggest determinant of a given function’s runtime and Jukebox’s efficacy.

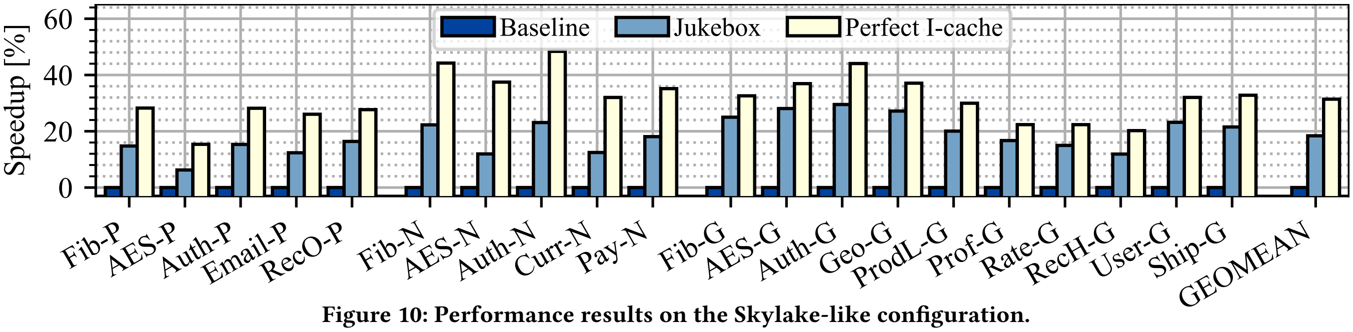


Figure 10: Performance results on the Skylake-like configuration.

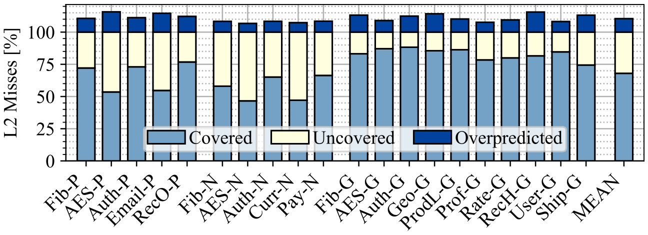


Figure 11: Fractions of L2 instruction misses covered and overpredicted by Jukebox (normalized to the number of L2 misses in the baseline).

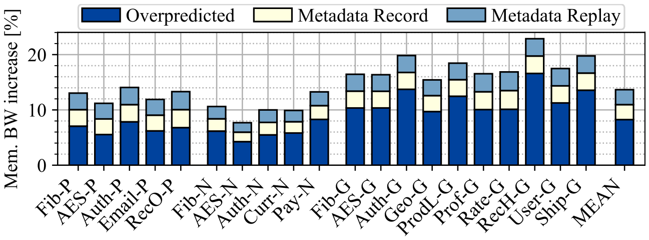


Figure 12: Jukebox's memory bandwidth overhead.

the complete instruction footprint a function accumulates over all simulated invocations.

The performance of the three evaluated configurations is shown in Figure 10 with the results normalized to the baseline. We find the maximum opportunity without any instruction misses boosts performance of the studied functions by 31% on average (46% max on Auth-N). Jukebox delivers consistent speedups that correlates well with the opportunity; that is, functions that have a large difference in performance between Perfect I-cache and the baseline enjoy large speedups (e.g., Auth-G: 29.5% speedup with Jukebox), while the opposite is true for functions with a small difference between Perfect I-cache and the baseline (e.g., AES-P: 6.2% speedup with Jukebox). On average, Jukebox speeds up interleaved executions by 18.7%.

5.3 Miss Coverage

We next study Jukebox's ability to cover instruction misses. Since Jukebox prefetches into the L2 cache, we present fractions of L2 instruction misses in the baseline that are (1) covered, (2) not covered, and (3) overpredicted (i.e., prefetched but not referenced) by Jukebox.

Figure 11 shows the result of this study. One can see that coverage correlates well with the choice of a programming language; benchmarks that are written in Go show high Jukebox coverage (75-90%) while those written in Python and NodeJS exhibit lower coverage (48-74%). This can be explained by the fact that for the majority of the Go benchmarks, metadata fits into Jukebox's metadata storage, which is not the case for Python and NodeJS benchmarks (see Figure 8).

Furthermore, the figure shows that Jukebox induces few wrong prefetches with an overprediction rate of just 10% (max. 15.8%). This result is anticipated by the high commonality in instruction footprints across invocations (Sec. 2.5). The high accuracy of Jukebox's prefetches affirm its record and replay approach to be highly effective in delivering the relevant instruction blocks on chip.

5.4 Memory Bandwidth

Figure 12 plots memory bandwidth usage of Jukebox normalized to the baseline. Memory bandwidth includes all requests issued to memory, which includes both instruction and data, demand and prefetches. Note that Jukebox does not change the amount of bandwidth consumed for correct timely prefetches. Overheads lie in overpredicted (i.e. unused) prefetches as well as metadata traffic associated with recording and replaying. Jukebox introduces a modest memory bandwidth overhead of 14% on average and 23% in the worst case. The overhead comprises 40% of Jukebox's metadata and 60% overpredicted traffic.

Similar to the coverage study (Sec. 5.3), we note that a memory bandwidth increase correlates with the choice of a programming language. Go workloads experience a higher bandwidth increase than workloads written in Python and NodeJS. Higher memory bandwidth on Go workloads compared to Python and NodeJS ones can be explained by a larger number of uncovered misses in Go workloads observed in the coverage study. Jukebox's metadata storage is too small to hold all metadata required for prefetching the whole instruction working set of Python and NodeJS workloads. As a result, at replay, Jukebox stops restoring the working set of Python and NodeJS workloads before completing prefetching of the whole instruction working set, which results in a lower number of overpredictions on Python and NodeJS workloads compared to Go ones.

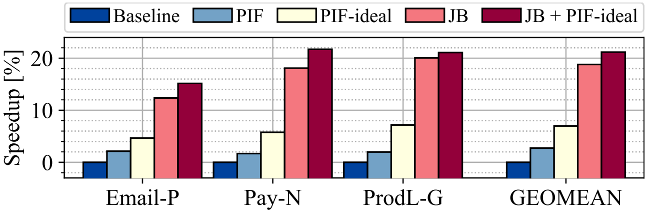


Figure 13: Comparison of performance with PIF and Jukebox.

5.5 Comparison to a State-of-the-Art Instruction Prefetcher

In this section, we compare Jukebox to a state-of-the-art instruction prefetcher, called PIF [16]. PIF is a stream-based prefetcher, which works by recording and replaying the sequences of retired instruction addresses. Recording all instruction addresses allows PIF to be be independent of variations in L1-I cache content and application's control flow, thus achieving good prefetch accuracy. To enable replay of instruction streams, PIF requires an *index*, which uses an instruction address to find the most recent recorded stream that starts with that address.

By using the same parameters as in [16], we configure PIF with a 49KB index, 164KB of stream metadata storage and an unrealistic single-cycle lookup latency for each of these components. Because PIF was designed for long-running traditional server workloads, it does not save its state across function invocations. To understand the best possible performance of PIF, we simulate another design, PIF-ideal, with an unlimited index and unlimited metadata storage that persist across function invocations.

Figure 13 plots the results of this study. We find that PIF delivers 2.4% speedup on average (4.8% max) while PIF-ideal boosts performance by 6.7% (12.4% max). Meanwhile, Jukebox with metadata size limited to 16KB provides a 18.7% speedup, on average – a significant improvement over PIF and PIF-ideal. The reason for PIF's relatively poor efficacy can be explained by the fact that whenever the recorded stream differs from the actual access stream of the core, PIF stops prefetching and re-indexes to find the correct stream. Re-indexing prevents PIF from running far enough ahead of the core to cover the long latency of a main memory access.

The big picture is that PIF was designed to reduce L1-I misses for accesses expected to hit in the L2 or L3 caches. In contrast, lukewarm functions have instruction footprints in main memory, which requires a prefetcher that can effectively hide the associated high latency. PIF needs to stop and re-index any time the actual control flow diverges from the prefetch stream, whereas Jukebox prefetches all of the instruction blocks recorded in its metadata without synchronizing with the core. By doing such bulk prefetching, Jukebox sacrifices the ability to prefetch into the small L1-I but achieves high instruction miss coverage in the L2 and L3.

5.6 Jukebox on a Broadwell-like CPU

As we perform the Top-Down analysis on a platform with an Intel Broadwell CPU (Section 2.3), we compare Jukebox performance results obtained with a simulated Intel Skylake configuration to the Broadwell-based platform. A distinguishing feature in the Broadwell configuration is a different cache hierarchy, with a 32KB L1-I, 256KB L2 and 8MB LLC. We re-assess Jukebox parameters in light of the smaller L2 as compared to the 1MB L2 in Skylake, and find that the smaller L2 results in more conflict misses for instructions, thus necessitating a larger 32KB Jukebox metadata store per function. Other Jukebox parameters (region size and CRRB size) are unchanged from the Skylake configuration.

	L2 instructions misses	LLC instruction misses
Skylake	-74%	-86%
Broadwell	-15%	-91%

Table 3: Reduction in L2 and L3 MPKI with Jukebox.

Across our suite of serverless functions, we find that Jukebox delivers a 12% geomean speedup on the Broadwell configuration. Noting that the speed-up is smaller than the 18.7% achieved on Skylake (Sec. 5.2) despite a similar opportunity with a perfect L1-I, we examine the cache miss rates for instructions in the two simulated platforms.

The data is presented in Table 3, which shows the reduction in MPKI for instructions in the L2 and L3 of the two simulated platforms with Jukebox. The table shows that Jukebox is highly effective at eliminating the vast majority of LLC misses for instructions in both platforms. These are the crucial misses to cover due to their excessively high latency. When it comes to the L2, however, Jukebox struggles to cover most L2 misses for instructions in the Broadwell configuration. This can be explained by the high incidence of conflicts in Broadwell's small L2, which result in many of Jukebox's prefetches being evicted from the L2 before they are consumed.

To summarize, Jukebox is most effective in CPUs with a large L2, which have featured in recent server processors [1, 40, 46], yet also provides a tangible benefit in CPUs with a smaller L2.

6 RELATED WORK

To date, there has been little work in understanding microarchitectural implications of serverless programming. Shahrad et al. [42] examined execution of five serverless functions, identifying issues such as a high cold-start latency, high variability in execution time, and performance overheads due to containerization. The work observed that short-running functions experienced much higher variability in execution time than long-running tasks, but did not attempt to understand the sources of variability other than high branch mispredictions. In contrast, the focus of our paper is on understanding and improving the microarchitectural behavior of lukewarm functions. To that end, we conducted a Top-Down performance analysis [52] of 20 diverse functions on a modern server, identifying concrete microarchitectural sources of performance loss stemming from a high degree of function interleaving. Based on our analysis, we identified on-chip instruction misses as the main performance bottleneck in lukewarm executions and proposed a specialized prefetcher to tackle the problem.

Prior works examine the problem of fine-grained context switches in highly-consolidated virtual machines [10, 53]. These works focus on a setting where a single virtual machine occupies the entire

CPU for multiple milliseconds, followed by a context switch to another VM. The problem solved in these works is restoring the entire multi-megabyte LLC state via prefetching by saving the address footprint of the LLC to main memory upon a context switch. The proposed designs suffer from large metadata overheads for high coverage and high bandwidth overheads associated with indiscriminate restoration of the entire LLC (in some cases more than doubling the amount of memory traffic compared to the no-prefetch baseline [10, 53]).

In contrast, Jukebox targets on-chip instruction misses by prefetching directly into the L2 cache using a minimal amount of metadata. Due to high instruction commonality across invocations of a given serverless function instance, Jukebox achieves high accuracy with low overprediction. While both Jukebox and prior works save prefetcher metadata in main memory, prior works save physical addresses, which are the only ones available at the LLC; in contrast, Jukebox saves virtual addresses, which makes Jukebox naturally compatible with a modern virtual memory manager that can move pages in memory (e.g., for memory compaction purposes).

Jevdjic et al. [24] record spatial footprints of data pages to reduce off-chip bandwidth pressure for aggressive data prefetching. While Jukebox also uses the idea of footprints, it targets instruction prefetching with low metadata cost.

Ahn et al. also examines fine-grained context switches for virtualized systems and proposes a context-preservation technique that controls the LLC capacity available for each virtual machine, to preserve LLC working set across context switches [3]. Zhu et al. examine event-driven server-side applications and identify L1-I misses to be a major performance bottleneck [55]. The authors observe that instruction working sets of the studied applications fit in the LLC and propose a specialized cache replacement policy to preserve an instruction working in the LLC and augment it with a temporal prefetcher in the L1-I cache. Both of these techniques target settings where the instruction working set fits in the LLC, which is not the case for serverless functions with infrequent invocations and a huge degree of interleaving.

There is a long history of works in instruction prefetching for server workloads. These papers fall into one of two categories: temporal streaming and BTB-directed. The former category records entire traces of instruction cache accesses or misses at the cache block granularity, resulting in metadata size of hundreds of kilobytes and requiring a complex indexing mechanism to find the correct metadata when the actual execution diverges from the recorded trace [16, 17, 28]. To store the metadata, existing temporal streaming proposals either use dedicated on-chip storage [16] or virtualize the metadata into the LLC [28]. In contrast, Jukebox uses aggressive filtering and spatial encoding resulting in low metadata costs, does not require any indexing, prefetches into the L2 to further simplify the prefetcher design and stores its metadata in memory to support thousands of warm functions.

The second category uses the BTB together with the branch predictor to identify upcoming control flow discontinuities to drive the instruction prefetch engine [33, 41]. This approach relies on a fully warmed up BTB and branch predictor, which makes it fundamentally at odds with lukewarm executions that have to contend with a cold core.

7 CONCLUSION

This paper identifies and addresses microarchitectural bottlenecks in the execution of serverless functions, thousands of which may reside concurrently in the memory of a modern cloud server. Due to the long invocation inter-arrival times as compared to execution latencies of each function, numerous other functions' executions may be interleaved between two invocations of a given function instance. This leads to the *lukewarm execution* phenomenon, whereby an invoked memory-resident function instance may find all of the on-chip microarchitectural state obliterated by other function instances.

The analysis of performance counters on a real server shows the core front-end to be the critical performance bottleneck in lukewarm executions due to the need to fetch instructions from main memory. In response, we proposed Jukebox, a record-and-replay prefetcher specifically designed to accelerate lukewarm invocations. Jukebox exploits instruction commonality across invocations to provide high instruction miss coverage with a low metadata cost and low design complexity.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers as well as the members of the EASE Lab at the University of Edinburgh for their valuable feedback on this work. We are grateful to Yuchen Niu for developing the initial platform for the Top-Down analysis of a serverless system and Harshit Garg for helping make the studied workloads publicly available. This research was generously supported by the University of Edinburgh, Arm and by EASE Lab's industry partners and sponsors: Facebook, Google, Huawei, Intel and Microsoft.

REFERENCES

[1] 7-Zip LZMA Benchmark. 2022. Intel Skylake. Retrieved April 12, 2022 from https://www.7-cpu.com/cpu/Skylake_X.html

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications.. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*. 419–434.

[3] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh. 2014. Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems.. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 394–405.

[4] Amazon. 2022. A Demo Running 4000 Firecracker MicroVMs. Retrieved April 12, 2022 from https://github.com/firecracker-microvm/firecracker-demo

[5] Amazon Web Services. 2022. AWS Lambda Pricing. Retrieved April 12, 2022 from https://aws.amazon.com/lambda/pricing

[6] Amazon Web Services. 2022. Use API Gateway Lambda Authorizers. Retrieved April 12, 2022 from https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-use-lambda-authorizer.html

[7] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2020. Divide and Conquer Frontend Bottleneck.. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. 65–78.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[9] Colin Ian King. 2022. Stress-ng. Retrieved April 12, 2022 from https://github.com/ColinIanKing/stress-ng

[10] David Daly and Harold W. Cain. 2012. Cache restoration for highly partitioned virtualized systems.. In *Proceedings of the 18th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 225–234.

[11] Datadog. 2020. The State of Serverless 2020. Retrieved April 12, 2022 from https://www.datadoghq.com/state-of-serverless-2020

[12] Datadog. 2021. The State of Serverless 2021. Retrieved April 12, 2022 from https://www.datadoghq.com/state-of-serverless

[13] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting.. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 467–481.

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 1–14.

[15] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2020. A Review of Serverless Use Cases and their Characteristics. *CoRR* abs/2008.11110 (2020).

[16] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch.. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 152–162.

[17] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming.. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–10.

[18] 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 *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*. 3–18.

[19] Google. 2022. Cloud Functions Pricing. Retrieved April 12, 2022 from https://cloud.google.com/functions/pricing

[20] Google Cloud. 2022. Implementing SLOs. Retrieved April 12, 2022 from https://sre.google/workbook/implementing-slos

[21] GoogleCloudPlatform. 2022. Online Boutique. Retrieved April 12, 2022 from https://github.com/GoogleCloudPlatform/microservices-demo

[22] gRPC Authors. 2022. gRPC: A High-Performance, Open Source Universal RPC Framework. Retrieved April 12, 2022 from https://grpc.io

[23] Paul Jaccard. 1912. THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE.1. *New Phytologist* 11, 2 (1912), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x arXiv:https://nph.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-8137.1912.tb05611.x

[24] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. *SIGARCH Comput. Archit. News* 41, 3 (jun 2013), 404–415. https://doi.org/10.1145/2508148.2485957

[25] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices.. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*. 152–166.

[26] Jonathan Corbet. 2010. Memory compaction. Retrieved April 12, 2022 from https://lwn.net/Articles/368869

[27] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings 28th annual international symposium on computer architecture*. IEEE, 240–251.

[28] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. SHIFT: shared history instruction fetch for lean-core server processors.. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 272–283.

[29] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers.. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 166–177.

[30] kernel.org. 2020. perf: Linux profiling with performance counters. Retrieved April 12, 2022 from https://perf.wiki.kernel.org/index.php/Main_Page

[31] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service.. In *Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD)*. 502–504.

[32] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS.. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*. 477.

[33] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A Metadata-Free Architecture for Control Flow Delivery.. In *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 493–504.

[34] Linux Foundation. 2008. pthread_create – Linux manual page. Retrieved April 12, 2022 from https://man7.org/linux/man-pages/man3/pthread_create.3.html

[35] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. https://doi.org/10.48550/ARXIV.2007.03152

[36] Mikhail Shilkov. 2021. When Does Cold Start Happen on AWS Lambda? Retrieved April 12, 2022 from https://mikhail.io/serverless/coldstarts/aws/intervals

[37] Mikhail Shilkov. 2021. When Does Cold Start Happen on Azure Functions? Retrieved April 12, 2022 from https://mikhail.io/serverless/coldstarts/azure/intervals

[38] Mikhail Shilkov. 2021. When Does Cold Start Happen on Google Cloud Functions? Retrieved April 12, 2022 from https://mikhail.io/serverless/coldstarts/gcp/intervals

[39] Oracle Corporation. 2010. Chapter 3 Thread Create Attributes. Retrieved April 12, 2022 from https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032j/index.html

[40] Paul Lilly. 2021. Leaked AMD Zen 4 Cache Upgrades Could Be Key In Competing With Alder Lake. Retrieved April 12, 2022 from https://hothardware.com/news/amd-zen-4-cache-key-competing-alder-lake

[41] Glenn Reinman, Brad Calder, and Todd M. Austin. 1999. Fetch Directed Instruction Prefetching.. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 16–27.

[42] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing.. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1063–1075.

[43] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider.. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 205–218.

[44] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing.. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 419–433.

[45] Akshitha Sriraman and Thomas F. Wenisch. 2018. µTune: Auto-Tuned Threading for OLDI Microservices.. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*. 177–194.

[46] The Cloudflare Blog. 2021. ARMs Race: Ampere Altra Takes on the AWS Graviton2. Retrieved April 12, 2022 from https://blog.cloudflare.com/arms-race-ampere-altra-takes-on-aws-graviton2

[47] The Cloudflare Blog. 2021. The EPYC Journey Continues to Milan in Cloudflare's 11th Generation Edge Server. Retrieved April 12, 2022 from https://blog.cloudflare.com/the-epyc-journey-continues-to-milan-in-cloudflares-11th-generation-edge-server

[48] The Firecracker Authors. 2022. Production Host Setup Recommendations. Retrieved April 12, 2022 from https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md

[49] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. 2021. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

[50] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots.. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*. 559–572.

[51] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 39, 16 pages. https://doi.org/10.1145/3302424.3303978

[52] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture.. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44.

[53] Jason Zebchuk, Harold W. Cain, Xin Tong, Vijayalakshmi Srinivasan, and Andreas Moshovos. 2013. RECAP: A region-based cure for the common cold (cache).. In *Proceedings of the 19th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 83–94.

[54] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference.. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. 456–468.

[55] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. 2015. Microarchitectural implications of event-driven server-side web applications.. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 762–774.