Model-driven Scheduling for Distributed Stream Processing Systems

Anshu Shukla and Yogesh Simmhan

Department of Computational and Data Sciences
Indian Institute of Science (IISc), Bangalore 560012, India
Email: shukla@grads.cds.iisc.ac.in, simmhan@cds.iisc.ac.in

Abstract

Distributed Stream Processing frameworks are being commonly used with the evolution of Internet of Things(IoT). These frameworks are designed to adapt to the dynamic input message rate by scaling in/out.Apache Storm, originally developed by Twitter is a widely used stream processing engine while others includes Flink [8] Spark streaming [73]. For running the streaming applications successfully there is need to know the optimal resource requirement, as over-estimation of resources adds extra cost.So we need some strategy to come up with the optimal resource requirement for a given streaming application. In this article, we propose a model-driven approach for scheduling streaming applications that effectively utilizes *a priori* knowledge of the applications to provide predictable scheduling behavior. Specifically, we use application performance models to offer reliable estimates of the resource allocation required. Further, this intuition also drives resource mapping, and helps narrow the estimated and actual dataflow performance and resource utilization. Together, this model-driven scheduling approach gives a predictable application performance and resource utilization behavior for executing a given DSPS application at a target input stream rate on distributed resources.

1 Introduction

Big Data platforms have evolved over the last decade to address the unique challenges posed by the ability to collect data at vast scales, and the need to process them rapidly. These platforms have also leveraged the availability of distributed computing resources, such as commodity clusters and Clouds, to allow the application to scale out as data sizes grow. In particular, platforms like Apache Hadoop and Spark have allowed massive data volumes to be processed with high throughput, and NoSQL databases like Hive and HBase support low latency queries over semi-structured data at large scales.

However, much of the research and innovation in Big Data platforms has skewed toward the *volume* rather than the *velocity* dimension [41]. On the

other hand, the growing prevalence of Internet of Things (IoT) is contributing to the deployment of physical and virtual sensors to monitor and respond to infrastructure, nature and human activity is leading to a rapid influx of streaming data [29]. These emerging applications complement the existing needs of micro-blogs like Twitter that already contend with the need to rapidly process tweet streams for detecting trends or malicious activity [38]. Such streaming applications require low-latency processing and analysis of data streams to take decisions that control the physical or digital eco-system they observe.

A *Distributed Stream Processing System (DSPS)* is a Big Data platform designed for online processing of such data streams [14]. While early stream processing systems date back to applications on wireless sensor networks [11], contemporary DSPS's such as Apache Storm from Twitter, Flink and Spark Streaming are designed to execute complex dataflows over tuple streams using commodity clusters [8, 62, 73]. These dataflows are typically designed as Directed Acyclic Graphs (DAGs), where user tasks are vertices and streams are edges. They can leverage data parallelism across tuples in the stream using multiple threads of execution per task, in addition to pipelined and task-parallel execution of the DAG, and have been shown to process $1000's$ of tuples per second [56, 62].

A DSPS executes streaming dataflow applications on distributed resources such as commodity clusters and Cloud Virtual Machines (VMs). In order to meet the required performance for these applications, the DSPS needs to schedule these dataflows efficiently over the resources. Scheduling for a DSPS has two parts: *resource allocation* and *resource mapping*. The former determines the appropriate degrees of parallelism per task (e.g., threads of execution) and quanta of computing resources (e.g., number and type of VMs) for the given dataflow. Here, care has to be taken to avoid both over-allocation, that can have monetary costs when using Cloud VMs, or under-allocation, that can impact performance. Resource mapping decides the specific assignment of the threads to the VMs to ensure that the expected performance behavior and resource utilization is met.

Despite their growing use, resource scheduling for DSPSs tends to be done in an *ad hoc* manner, favoring empirical and reactive approaches, rather than a model-driven and analytical approach. Such empirical approaches may arrive at an approximate resource allocation for the DSPS based on a linear extrapolation of the resource needs and performance of dataflow tasks, and hand-tune these to meet the Quality of Service (QoS) [55,71]. Mapping of tasks to resources may be round-robin or consider data locality and resource capacities [37,50]. More sophisticated research techniques support dynamic scheduling by monitoring the queue waiting times or tuple latencies to incrementally increase/decrease the degrees of parallelism for individual tasks or the number of VMs they run on [25, 40]. While these dynamic techniques have the advantage of working for arbitrary dataflows and stream rates, such schedules can lead to local optima for individual tasks without regard to global efficiency of the dataflow, introduce latency and cost overheads due to constant changes to the mapping, or offer weaker guarantees for the QoS.

In this article, we propose a model-driven approach for scheduling streaming applications that effectively utilizes *a priori* knowledge of the applications to provide predictable scheduling behavior. Specifically, we leverage our observation that dataflow tasks have diverse performance behavior as the degree of parallelism increases, and use performance models to offer reliable estimates of the resource allocation required. Further, this intuition also drives resource mapping to mitigate the impact of multi-tenancy of different tasks on the same resource, and helps narrow the estimated and actual dataflow performance and resource utilization. Together, this model-driven scheduling approach gives a predictable application performance and resource utilization behavior for executing a given DSPS application at a target input stream rate on distributed resources. Often, importance is given to lower latency of resource usage rather than predictable behavior. But in stream processing that can be latency sensitive, it may be more important to offer tighter bounds rather than lower bounds.

We limit this article to static scheduling of the dataflow on distributed resources, before the application starts running. This is complementary to dynamic scheduling algorithms that can react to changes in the stream rates [72], application composition [40] and make use of Cloud elasticity [52]. However, our work can be extended and applied to a dynamic context as well. Rather than incrementally increase or decrease resource allocation and the mapping until the QoS stabilizes, a dynamic algorithm can make use of our model to converge to a stable configuration more rapidly. Our work is of particular use for enterprises and service providers who have a large class of infrastructure applications that are run frequently [35, 58], or who reuse a library of common tasks when composing their applications [7, 13, 24], as is common in the scientific workflow community [15]. This amortizes the cost of building task-level performance models. Our approach also resembles scheduling in HPC centers that typically have a captive set of scientific applications that can benefit from such a model-driven approach [17] [65].

Specifically, we make the following key contributions in this article:

1. We highlight the gap between ideal and actual performance of dataflow tasks using performance models, that causes many existing DSPS scheduling algorithms to fail and motivates our model-based approach for reliable scheduling.

2. We propose an allocation and a mapping algorithm that leverage these performance models to schedule DSPS dataflows for a fixed input rate, minimizing the distributed resources used and offering predictable performance behavior.

3. We offer detailed experimental results and analysis evaluating our scheduling algorithm using the Apache Storm DSPS, and compare it against the state-of-the-art scheduling approaches, for micro and application dataflows.

The rest of the article is organized as follows: § 2 introduces the problem *motivation* and § 3 formalizes the scheduling *problem*; § 4 offers a high-level

intuition of the analytical *approach* taken to solving the problem; § 5 offers evidence on the diversity of task's behavior using *performance models*, leveraged in the solution; § 6 proposes a novel *Model Based Allocation (MBA) algorithm* using these models, and also describes a Linear Scaling Allocation (LSA) used as a contemporary baseline; § 7 presents our *Slot-Aware Mapping (SAM) algorithm* that leverages thread-locality in a resource slot, and lists two existing algorithms from literature and practice used as comparison; we offer detailed experimental *results and analysis* of these allocation and mapping algorithms in § 8; contrast our work against *related literature* in § 9; and lastly, present our *conclusions* in § 10.

2   Background and Motivation

We offer an overview of the generic composition and execution model favored by contemporary DSPSs such as Apache Storm, Apache Spark Streaming, Apache Flink and IBM InfoSphere Streams in this section. We further use this to motivate the specific research challenges and technical problems we address in this article; a formal definition follows in the subsequent section, § 3. While we use features and limitations of the popular Apache Storm as a representative DSPS here, similar features and short-comings of other DSPSs are discussed in the related work, § 9.

Streaming applications are composed as a dataflow in a DSPS, represented as a *directed acyclic graph (DAG)*, where *tasks* form vertices and *tuple streams* are the edges connecting the output of one task to the input of its downstream task. Contents of the *tuples* (also called *events* or *messages*) are generally opaque to the DSPS, except for special fields like IDs and keys that may be present for recovery and routing. *Source tasks* in the DAG contain user logic responsible for acquiring and generating the initial input stream to the DAG, say by subscribing to a message broker or pulling events over the network from sensors. For other tasks, their logic is executed once for each input tuple arriving at that task, and may produce zero or more output tuples for each invocation. These output tuples are passed to downstream tasks in the DAG, and so on till the *Sink tasks* are reached. These sinks do not emit any output stream but may store the tuples or notify external services. Apache Storm uses the terms *topology* and *component* for a DAG and a task, and more specifically *spout* and *bolt* for source tasks and non-source tasks, respectively.

Multiple outgoing edges connecting one task to downstream tasks may indicate different *routing semantics* for output tuples on that edge, based on the application definition – tuples may be *duplicated* to all downstream tasks, passed in a *round-robin* manner to the tasks, or *mapped* based on an output key in the tuple. Likewise, multiple input streams incident on a task typically have their tuples *interleaved* into a single logical stream, though semantics such as *joins* across tuples from different input streams are possible as well. Further, the *selectivity* of an outgoing edge of a task is the ratio between the average number of tuples generated on that output stream for each input tuple to the task.

4

Streaming applications are designed to process tuples with low latency. The *end-to-end latency* for processing an input tuple from the source to the sink task(s) is typically a measure of the *Quality of Service (QoS)* expected. This QoS depends on both the *input stream rate* at the source task(s) and the resource allocation to the tasks in the DSPS. A key requirement is that the execution performance of the streaming application remains *stable*, i.e., the end-to-end latency is maintained within a narrow range over time and the queue size at each task does not grow. Otherwise, an unstable application can lead to an exponential growth in the latency and the queue size, causing hosts to run out of memory.

The execution model of a DSPS can naturally leverage *pipelining* and *task parallelism* due to the composition of linear and concurrent tasks in the DAG, respectively. These benefits are bound by the length and the number of tasks in the DAG. In addition, they also actively make use of *data-parallel* execution for a single task by assigning multiple threads of execution that can each operate on an independent tuple in the input stream. This data parallelism is typically limited to stateless tasks, where threads of execution for a task do not share a global variable or state, such as a sum and a count for aggregation; stateful tasks require more involved distributed coordination [68]. Stateless tasks are common in streaming dataflows, allowing the DSPS to make use of this *important dimension of parallelism that is not limited by the dataflow size but rather the stream rate and resource availability.*

In operational DSPSs such as Apache Storm, Yahoo S4 [45], Twitter Heron [38] and IBM InfoSphere Streams [7], the platform expects the application developer to provide the *number of threads* or degrees of data parallelism that should be exploited for a single task. As we show in § 5, general rules of thumb are inadequate for deciding this and both over- and under-allocation of threads can have a negative effect on performance. This value may also change with the input rate of the stream. Thread allocation is one of the challenges we tackle in this paper.

In addition, the user is responsible for deciding the *number of compute resources* to be allocated to the DAG. Typically, as with many Big Data platforms, each host or Virtual Machine (VM) in the DSPS cluster exposes multiple *resource slots*, and those many *worker processes* can be run on the host. Typically, there are as many workers as the number of CPU cores in that host. Each worker process can internally run multiple threads for one or more tasks in the DAG. The user specifies the number of hosts or slots in the DSPS cluster to be used for the given DAG when it is submitted for execution. For e.g., in Apache Storm, the threads for the dataflow can use all available resource slots in the DSPS cluster by default. In practice, this again ends up being a trial and error empirical process for the user or the system to decide the resource allocation for the DAG, and will change for each DAG or the input rate that it needs to support. Ascertaining the Cloud VM resource allocation for the given DAG and input rate is another problem that we address in this article, and this is equally applicable to commodity hosts in a cluster as well.

The DSPS, on receiving a DAG for scheduling, is responsible for deploying

the dataflow on the cluster and coordinating its execution. As part of this deployment, it needs to decide the mapping from the threads of the tasks to the slots in the workers. There has been prior work on making this mapping efficient for stream processing platforms [19] [37]. For e.g., the native scheduler of Apache Storm uses a round-robin technique for assigning threads from different tasks to all available slots for the DAG. Its intuition is to avoid contention for the same resource by threads for the same task, and also balance the workload among the available workers. More recently, Storm has incorporated the R-Storm scheduler [48] that is both resource and network topology aware, and this offers better efficiency by considering the resource needs for an individual task thread. In InfoSphere Streams [7] and our own prior work [39], the mapping decision is dynamic and relies on the current load and previous resource utilization for the tasks.

This placement decision is important, as an optimal resource allocation for a given DAG and input rate may still under-perform if the task thread to worker mapping is not effective. This inefficiency can be due to additional costs for resource contention between threads of a task or different tasks on a VM, excess context switching between threads in a core, movement of tuples between distributed tasks over the network, among other reasons. This inefficiency is manifest in the form of additional latency for the messages to be processed, or a lower input rate that is supported in a stable manner for the DAG with the given resources and mapping. In this article, we enhance the mapping strategy for the DSPS by using a model-driven approach that goes beyond a resource-aware approach, such as used in R-Storm.

In summary, the aim of this paper is to *use a model-driven approach to perform predictable and efficient resource scheduling for a given DAG and input event rate.* The specific goals are to determine:

- the thread allocation per task,

- the VM allocation for the entire DAG, and

- the resource mapping from a task thread to a resource slot.

The outcome of this schedule will be to improve the efficiency and reduce the contention for VM resources, reduce the number of VM resources, and hence *monetary costs*, for executing the DAG, and ensure a stable (and preferably low) latency for execution. The *predictable performance* of the proposed schedule is also important as it reduces uncertainty and trial and error. Further, when using this scheduling approach for handling dynamism in the workload or resources, say when the input rate or the DAG structure changes, this predictability allows us to update the schedule and pay the overhead cost for the rebalancing once, rather than constantly tweak the schedule purely based on monitoring of the execution.

While our article does not directly address dynamism, such as changing input rates, non-deterministic VM performance or modifications to the DAG, the approach we propose offers a valuable methodology for supporting it. Likewise,

6

Notation	Description
$\mathcal{G} = \langle \mathbb{T}, \mathbb{E} \rangle$	DAG to be scheduled
$\mathbb{T} = \{t_1, ..., t_n\}$	Set of n task vertices in the DAG
$\mathbb{E} = \{e_{ij} | e_{ij} = \langle t_i, t_j \rangle\}$	Set of stream edges in the DAG
Ω	Input rate (tuples/sec) to be supported for DAG
$v_j \in \mathbb{V}$	VMs available
p_j	Number of resource slots available in VM v_j
q_i	Number of threads allocated to task t_i
$r_i^k \in R$	Set of task threads allocated to tasks in DAG
ρ	Sum of the resource slots allocated to the DAG
$v_j \in V$	VMs allocated to the DAG
$s_j^l \in S$	Resource slots in VMs allocated to DAG
$\mathcal{M} : R \to S$	Mapping function from a thread to its slot
$\mathcal{P}_i : \tau \to \langle \omega, c, m \rangle$	Performance model for task t_i. Maps from τ threads to the peak input rate supported ω, CPU% c and Memory% m
$\mathcal{C}_i(q), \quad \mathcal{M}_i(q)$	Incremental CPU%, memory% used by task t_i with q threads on a single resource slot
$\bar{c}_i = \mathcal{C}_i(1), \quad \bar{m}_i = \mathcal{M}_i(1)$	CPU% and memory% required by 1 thread of the task t_i on a single slot
$\mathcal{I}_i(q)$	Peak input rate supported by the task t_i with q threads on a single slot
$\mathcal{T}_i(\omega)$	Smallest thread count q needed to satisfy the input rate ω for task t_i on a single slot
$\bar{\omega}_i$	Peak rate sustained by 1 thread of task t_i running in 1 slot
$\widehat{\omega}_i$	Peak rate sustained across any number of threads of task t_i running in 1 slot
$C_j, \quad M_j$	Available CPU%, memory% on a VM
M_j^l	Available memory% on single slot
τ_i	Number of threads allocated to task t_i
$\widehat{v}$	Reference VM, VM on which last task thread was mapped
C_j^l, M_j^l	Available CPU%, memory% on single slot
$\widehat{\tau}_i$	Number of threads needed to support peak rate $\widehat{\omega}_i$ for task t_i on 1 slot

Figure 1: Summary of notations used in article

we limit our work here to scheduling a single streaming application on an exclusive cluster, that is common in on-demand Cloud environments, rather than multi-tenancy of applications in the same cluster. Our algorithm in fact helps determine the smallest number of VMs and their sizes required to meet the application's needs.

3 Problem Definition

Let the input DAG be given as $\mathcal{G} = \langle \mathbb{T}, \mathbb{E} \rangle$, where $\mathbb{T} = \{t_1, ..., t_n\}$ is the set of n *tasks* that form the vertices of the DAG, and $\mathbb{E} = \{e_{ij} | e_{ij} = \langle t_i, t_j \rangle, \quad t_i, t_j \in \mathbb{T}\}$ is the set of *tuple stream* edges connecting the output of a task t_i to the input of its downstream task t_j. Let σ_{ij} be the *selectivity* for the edge e_{ij}. We assume interleave semantics on the input streams and duplicate semantics on the output streams to and from a task, respectively.

Further, we are given a set of VMs, $v_j \in \mathbb{V}$, with each VM v_j having multiple identical resource slots, $s_j^1..s_j^p$. Each slot is homogeneous in resource capacity and corresponds to a single CPU core of a rated speed and a specific quanta of memory allocated to that core. Let p_j be the number of processing slots

present in VM v_j. The VMs themselves may be heterogeneous in terms of the number of slots that they have, even as we assume for simplicity that the slots themselves are homogeneous. This is consistent with the "container" or "slot" model followed by Big Data platforms like Apache Hadoop [63] and Storm, though it can be extended to allow heterogeneous slots as well.

Task threads, $r_i^1..r_i^q$, are responsible for executing the logic for a task t_i on a tuple arriving on the input stream for the task. Each task has one or more threads allocated to it, and each thread is mapped to a resource slot. Different threads can execute different tuples on the input stream, in a data-parallel manner. The order of execution does not matter. Each resource slot can run multiple threads from one or more tasks concurrently.

We are given an input stream rate of Ω tuples/sec (or events/sec) that should be supported for the DAG $\mathcal{G}$. Our goal is to schedule the tasks of the DAG on *VMs with the least number of cumulative resource slots, to support a stable execution of the DAG* for the given input rate. This problem can be divided into two phases:

1. *Resource Allocation:* Find the minimum number q_i of task threads required per task t_i, and the cumulative resource slots ρ required to meet the input rate to the DAG. Minimizing the slots translates to a minimization of costs for these on-demand VM resources from Cloud providers since the pricing for VMs are typically proportional to the number of slots they offer.

2. *Resource Mapping:* Given the set of task threads $r_i^k \in R$ for all tasks in the DAG, and the number of resource slots ρ allocated to the DAG, determine the set of VMs V such that they have an adequate number of slots, $\rho \leq \sum_{v_j \in V} p_j$. Further, for resource slots $s_j^l \in S$ present in the VMs $v_j \in V$, find an optimal mapping function $\mathcal{M} : R \rightarrow S$ for the allocated task threads on to available slots, to match the resources needed to support the required input rate Ω in a stable manner.

There are several qualitative and quantitative measures we use to evaluate the solution to this problem.

1. The number of resource slots and VMs estimated by the allocation algorithm should be minimized.

2. The actual number of resource slots and VMs required by the mapping algorithm to successfully place the threads should be minimized. This is the actual cost paid for acquiring and using the VMs. Closeness of this value to the estimate from above indicates a reliable estimate.

3. The actual stable input rate that is supported for the DAG by this allocation and mapping at runtime should be greater than or equal to the required input rate Ω. A value close to Ω indicates better predictability.

8

4. The expected resource usage as estimated by the scheduling algorithm and the actual resource usage for each slot should be proximate. The closer these two values, the better the predictability of the dataflow's performance and the scheduling algorithm's robustness under dynamic conditions.

4   Solution Approach

We propose a model-based approach to solving the two sub-problems of resource allocation and resource mapping, in order to arrive at an efficient and predictable schedule for the DAG to meet the required input rate. The intuition for this is as follows.

The stable input rate that can be supported by a task depends on the the number of concurrent threads for that task that can execute over tuples on the input stream in a data-parallel manner. The number of threads for a task in turn determines the resources required by the task. Traditional scheduling approaches for DSPSs assume that both these relationships – between thread count and rate supported, and thread count and resources required – are a linear function. That is, if we double the number of threads for a task, we can achieve twice the input rate and require twice the computing resources.

However, as we show later in § 5 and Fig. 3, this is not true. Instead, we observe that depending on the task, both these relationships may range anywhere from flat line to a bell curve to a linear function with a positive or a negative slope. The reason for this is understandable. As the number of threads increase in a single VM or resource slot, there is more contention for those specific resources by threads of a task that can mitigate their performance. This can also affect the actual resource used by the threads. For simplicity, we limit our analysis to CPU and memory resources, though it can be extended to disk IOPS and network bandwidth as well. As a result of this real-word behavior of tasks, scheduling that assumes a linear behavior can under-perform.

In our approach, we embrace this non-uniformity in the task performance and incorporate the empirical model of the performance into our schedule planning. First, we use micro-benchmarks on a single resource slot to develop a *performance model* function $\mathcal{P}_i$ for each task t_i which, given a certain number of threads τ for the task on a single resource slot, provides the peak input rate ω supported, and the CPU and memory utilization, c and m, at that rate. This is discussed in § 5.

Second, we use these performance models to determine the number of threads q_i for each task t_i in the DAG that is required to support a given input rate, and the cumulative number of resource slots ρ for all threads in the DAG. This *Model Based Allocation (MBA)* described in § 6 offers an accurate estimate of the resource needs and task performance, for individual tasks and for the entire DAG. We also discuss a commonly used baseline approach, *Linear Scaling Allocation (LSA)*, in that section. As mentioned before, it assumes that the performance of a single thread on a resource slot can be linearly extrapolated
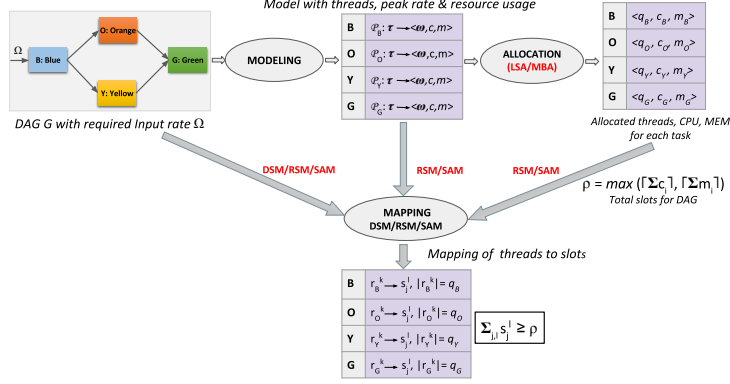
9

Figure 2: Illustration of *Modeling*, *Allocation* and *Mapping* phases performed when scheduling a sample DAG

to multiple threads on that slot, as long as the resource capacity of the slot is not exhausted. It under-performs, as we show later.

This resource allocation can subsequently be used by different resource mapping algorithms that exist, such as the round-robin algorithm used by default in Apache Storm [62], which we refer to as the *Default Storm Mapping (DSM)*, or a resource-aware mapping proposed in R-Storm [48] and included in the latest Apache Storm distribution, which we call *R-Storm Mapping (RSM)*. However, these mapping algorithms do not provide adequate co-location of threads onto the same slot to exploit the intuition of the model based allocation. We propose a novel *Slot Aware Mapping (SAM)* algorithm that attempts to map threads from the same task as a group to individual slots, as a form of *gang scheduling* [47]. Here, our goal is to maximize the peak event rate that can be exploited from that slot, minimize the interference between threads from different tasks, and ensure predictable performance. These allocation strategies are explored in § 7.

4.1 Illustration

We provide a high-level visual overview of the schedule planning in Fig. 2 for a given DAG $\mathcal{G}$ with four tasks, `Blue`, `Orange`, `Yellow` and `Green`, with a required input rate of Ω. The procedure has three phases. In the *Modeling* phase, we build a performance model $\mathcal{P}_i$ for tasks in the DAG that do not already have a model available. This gives a profile of the peak input tuple rates supported by a task with different numbers of threads, and their corresponding CPU and memory utilization, using a single resource slot. For e.g., the performance models for the four tasks in the DAG are given by $\mathcal{P}_B, \mathcal{P}_O, \mathcal{P}_Y$ and $\mathcal{P}_G$ in Fig. 2.

In the *Allocation* phase, we use the above performance model to decide the number of threads q_i for each task required to sustain the tuple rate that is incident on it. The input rate to the task is itself calculated based on the

10

DAG's input rate and the selectivity of upstream tasks. We use the Linear Scaling Allocation (LSA) and our Model Based Allocation (MBA) approaches for this. While LSA only uses the performance model for a single task thread, our MBA uses the full profile that is available. These algorithms also return the CPU% and memory% for the threads of each task that are summed up to get the total number of resource slots for the DAG. Fig. 2 shows the table for each of the four tasks after allocation, with the number of threads q_b, q_O, q_Y and q_G, and their estimated resource usages c and m that are summed up to calculate the total resource slot needs for the DAG ρ.

Lastly, the *Mapping* phase decides the number and types of VMs required to meet the resource needs for the DAG. It then maps the set of r_i^k threads allocated for the DAG to the slots s_j^l in the VMs, and the total number of these slots can be greater than the estimated ρ, depending on the algorithm. Here, we use the Default Storm Mapping (DSM), R-Storm Mapping (RSM) and our proposed Slot Aware Mapping (SAM) algorithms as alternatives. As shown in the figure, DSM is not resource aware and only uses the information on the number of threads q_i and the number of slots ρ for its round-robin mapping. RSM and SAM use the task dependencies between the DAG and the allocation table. RSM uses the performance of a single thread while SAM uses all values in the performance model for its mapping.

4.2 Discussion

As with any approach that relies on prior profiling of tasks, our approach has the short-coming of requiring effort to empirically build the performance model for each task before it can be used. However, this is mitigated in two respects.

First, as has been seen for scientific workflows, enterprise workloads and even HPC applications [16, 44, 54], many domains have common tasks that are reused in compositional frameworks by users in that domain. Similarly, for DSPS applications in domains like social network analytics, IoT or even Enterprise ETL (Extract, Transform and Load), there are common task categories and tasks such as parsing, aggregation, analytics, file I/O and Cloud service operations [24, 43, 56]. Identifying and developing performance models for such common tasks for a given user-base – even if all tasks are not exhaustively profiled – can help leverage the benefits of more efficient and predictable schedules for streaming applications.

Second, the effort in profiling a task is small and can be fully automated, as we describe in the next section. It also does not require access to the eventual DAG that will be executed. This ensures that as long as we can get access to the individual task, some minimal characteristics of the input tuple streams, and VMs with single resource slots comparable to slots in the eventual deployment, the time, costs and management overheads for building the performance model are mitigated.

5 Performance Modeling of Tasks

5.1 Approach

Performance modeling for a task builds a profile of the peak input tuple rate supported by the task, and its corresponding CPU and memory usage, using a single resource slot. It does this by performing a constrained parameter sweep of the number of threads and different inputs rates as a series of micro-benchmark trials. Algorithm 1 gives the pseudo-code for build such a performance model. For a given task t, we initialize the number of threads τ and the input rate ω to 1, and iteratively increase the number of threads (lines 6–17), and for each thread count, the input rate (lines 8–15). The steps at which the thread count and rates are increased, Δ_τ and Δ_ω, can either be fixed, or be a function of the iteration or the prior performance. This determines the granularity of the model – while a finer granularity of thread and rate increments offers better modeling accuracy, it requires more experiments, and costs time and money for VMs.

Algorithm 1 Performance Modeling of a Task

1: **procedure** PERFMODEL(TASK t)
2:　　$\mathcal{P} \leftarrow new\ Map(\ )$　　▷ *Holds the mapping from threads to input rate, resource usage*
3:　　$\lambda_\omega = 0$　　▷ *Slope of the last window of peak stable rates*
4:　　$\tau = 1$　　▷ *Number of threads being tested*
5:　　▷ *Increase the number of threads until τ_{max}, or slope of peak supported rate remains stable or drops*
6:　　**while** $\tau < \tau_{max}$ and $\lambda_\omega \leq \lambda_\omega^{max}$ **do**
7:　　　　▷ *For each value of τ, increase input rate in steps of Δ_ω until trial is unstable, or max rate ω_{max} reached*
8:　　　　**for** $\omega \leftarrow 1$ **to** ω_{max} **step** Δ_ω **do**
9:　　　　　　▷ *Run DSPS with task. Check if rate is supported. Get CPU and memory%.*
10:　　　　　　$\langle c, m, isStable \rangle \leftarrow$ RUNTASKTRIAL(t, τ, ω)
11:　　　　　　**if** $isStable = false$ **then**　　▷ *If rate not supported, break*
12:　　　　　　　　**break**
13:　　　　　　**end if**
14:　　　　　　$\mathcal{P}.put(\tau \rightarrow \langle \omega, c, m \rangle)$　　▷ *Add or update mapping from τ to peak rate, resource usage*
15:　　　　**end for**
16:　　　　$\lambda_\omega \leftarrow$ SLOPE$(\mathcal{P}, \omega)$　　▷ *Get slope of the last window of peak stable rates before ω*
17:　　　　$\tau \leftarrow \tau + \Delta_\tau$　　▷ *Increment thread count*
18:　　**end while**
19: **return** $\mathcal{P}$
20: **end procedure**

For each combination of τ and ω, we run a trial of the task (line 10) that creates a sequential 3-task DAG, with one source task that generates input tuples at the rate ω, the task t in the middle with task threads set to τ, and a sink task to collect statistics. The threads for task t are assigned one independent resource slot on one VM while on a second VM, the source and sink tasks run on one or more separate resource slots so that they are not resource-constrained.

This trial is run for a certain interval of time that goes past the "warm-up" period where initial scheduling and caching effects are overcome, and the DAG executes in a uniform manner. For e.g., in our experiments, the warm up period was seen to be $\leq 5\ mins$. During the trial, a profiler collects statistics on the CPU and memory usage for that resource slot, and the source and sink collect the latency times for the tuples. Running the source and sink tasks on the same VM avoids clock skews. At the end of the trial, these statistics are summarized and returned to the algorithm.

In running these experiments, two key termination conditions need to be determined for automated execution and generation of the model. For a given number of threads, we need to decide when we should stop increasing the input rate given to the task. This is done by checking the latency times for the tuples processed in the trial. Under stable conditions, the latency time for tuples are tightly bound and fall within a narrow margin beyond the warm-up period. However, if the task is resource constrained, then it will be unable to keep up its processing rate with the input rate causing queuing of input tuples. As the queue size keeps increasing, there will be an exponential growth in the end-to-end latency for successive tuples.

To decide if the task execution is stable, we calculate the slope λ_L of the tuple latency values for the trial period past the warm-up and check if it is constant or less than a small positive value, λ_L^{max}. If $\lambda_L \leq \lambda_L^{max}$, this execution configuration is stable, and if not, it is unstable. Once we reach an input rate that is not stable, we stop the trials for these number of threads for the task, and move to a higher number of threads (line 12). In our experiments, using a tight slope value of $\lambda_L^{max} \approx 0.001$ was possible, and none of the experiments ran for over 12 $mins$.

The second termination condition decides when to stop increasing the number of threads. Here, the expectation is that as the thread count increases there is an improvement, if any, in the peak rate supported until a point at which it either stabilizes or starts dropping. We maintain the peak rate supported for previous thread counts in the $\mathcal{P}$ hashmap object. As before, we take the slope λ_ω of the rates for the trailing window of thread counts to determine if the slope remains flat at 0 or turns negative. Once the rate drops or remains flat for the window, we do not expect to see an improvement in performance by increasing the thread count, and terminate the experiments. In our experiments, we set $\lambda_\omega^{min} \approx -0.001$.

5.2 Performance Modeling Setup

We identify 5 representative tasks, shown in Table 1, to profile and build performance models for. They also empirically motivate the need for fine-grained control over thread and resource allocation. These tasks have been chosen to be diverse, and representative of the categories of tasks often used in DSPS domains such as social media analytics, IoT and ETL pipelines [28, 64].

- *Parse XML.* It parses an array of in-memory XML strings for every in-

coming tuple. Parsing is often required for initial tasks in the DAG that receive text or binary encoded messages from external sources, and need to translate them to a form used by downstream tasks in the DAG. XML was used here due its popular usage though other formats like JSON or Google Protocol Buffers are possible as well. Our XML parsing implementation uses the Java SAX parser that allows serial single-pass parsing even over large messages at a fast rate. Parsing XML is CPU intensive and requires high memory due to numerous string operations (Table 1).

- *Pi Computation.* This task numerically evaluates the approximate value of π using an infinite series proposed by *Viete* [36]. Rather than running it non-deterministically till convergence, we evaluate the series for a fixed number of iterations, which we set to 15. This is a CPU intensive floating-point task, and is analogous to tasks that may perform statistical and predictive analytics, or computational modeling.

- *Batch File Write.* It is an accumulator task that resembles both window operations like aggregation or join, and disk I/O intensive tasks like archiving data. The implementation buffers a 100 *byte* string in-memory for every incoming tuple for a window size of $10,000$ tuples, and then writes the batched strings to a local file on a HDD attached to VM. The number of disk operations per second is proportional to the input message rate.

- *Azure Blob Download.* Streaming applications may download metadata annotations, accumulated time-series data, or trained models from Cloud storage services to use in their execution. Microsoft Azure Blob service stores unstructured data as files in the Cloud. This task downloads a file with a given name from the Azure Blob storage service using the native Azure Java SDK. In our implementation, a 2 MB file is downloaded and stored in-memory for each input tuple, making it both memory and network intensive.

- *Azure Table Query.* Some streaming applications require to access historic data stored in databases, say, for aggregation and comparison. Microsoft Azure Table service is a NoSQL columnar database in the Cloud. Our task implementation queries a table containing $2,000,000$ records, each with 20 columns and ≈ 200 *byte* record size [18], using the native Azure Java SDK. The query looks up a single record by passing a randomly generated record ID corresponding to a unique row key in the table.

As can be seen, these tasks cover a wide range of operations. These span from text parsing and floating-point operations to both local and Cloud-based file and table operations. There is also diversity in these tasks with respect to the resources they consume as shown in Table 1, be they memory, CPU, Disk or Network, and some rely on external services with their own Service Level Agreement (SLA).

14

Table 1: Characteristics of representative tasks for which performance modeling is performed

Task	CPU bound?	Memory bound?	N/W bound?	Disk I/O bound?	External Service?
Parse XML	✓	✓			
Pi Computation	✓				
Batched File Write				✓	
Azure Blob Download		✓	✓		✓
Azure Table Query			✓		✓

We wrap each of these tasks as a *bolt* in the Apache Storm DSPS. We compose a *topology* DAG consisting of 1 *spout* that generates synthetic tuples with 1 field (`message-id`) at a given constant rate determined by that trial, 1 bolt with the task that is being modeled, and 1 *sink* bolt that accepts the response from the predecessor. For each input tuple, The task bolt emits one output tuple after executing its application logic, keeping the selectivity $\sigma = 1 : 1$.

Apache Storm [1] is deployed on a set of standard *D type VMs* running Ubuntu 14.04 in Microsoft Azure's Infrastructure as a Service (IaaS) Cloud, in the Southeast Asia data-center. Each VM has 2^{i-1} resource slots, where i corresponds to the VM size, $D_i \in \{D_1, D_2, D_3, D_4\}$. Each slot has a one Intel Xeon E5-2673 v3 core @2.40 GHz processor with hyper-threading disabled [2], $3.5GB$ memory and $50GB$ SSD, for e.g., the D3 VM has 4 cores, 14 GiB RAM and 200 GiB SSD. A separate $50GB$ HDD is present for I/O tasks like Batch File Write.

For the performance modeling, we deploy the spout and the sink on separate slots of a single D2 size VM, and the task bolt being evaluated on one D1 size VM. The spout and sink have 2 threads each to ensure they are not the bottleneck, while the number of threads for the task bolt and the input rate to this topology is determined by Algorithm 1. Each trial is run for 12 *mins*, to be conservative. We measure the CPU% and memory% using the Linux `top` command, and record the peak stable rate supported for each of these task bolts for specific numbers of threads. The experiments are run multiple times, and the representative measurements are reported.

5.3 Performance Modeling Results

The goal of these experiments is to collect the performance model data used by the scheduling algorithms. However, we also supplement it with some observations on the task behavior. Fig. 3 shows the performance model plots for the 5 tasks we evaluate on a single resource slot. On the primary Y Axis (left), the

---

[1] Apache Storm v1.0.1, released on 6 May 2016

[2] Azure A-SERIES, D-SERIES and G-SERIES: Consistent Performances and Size Change Considerations, `https://goo.gl/0X6yT2`

(a) Parse XML   (b) Pi Computation

(c) Batched File Write   (d) Azure Blob Download   (e) Azure Table Query
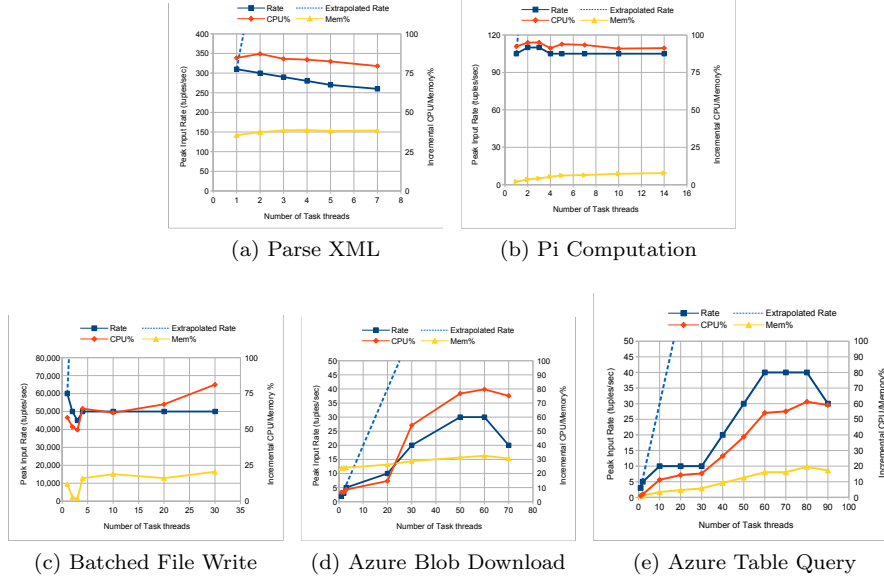
Figure 3: *Peak input rate supported* (primary Y Axis) on one resource slot for the specified task bolt, with an increase in the *number of threads* (X Axis). The *% incremental CPU and Memory usage* for these numbers of threads at the peak rate is shown in the secondary Y Axis.

plots show the peak stable rate supported (tuples/sec), and the corresponding CPU and memory utilization on the secondary Y Axis (right), as the number of threads for the task increases along the X Axis. The CPU% and memory% are given as a fraction of the utilization above the base load on the machine when no application topology is running, but when the OS and the stream processing platform are running. So a 0% CPU or memory usage in the plots indicate that only the base load is present, and a 100% indicates that the full available resource of the slot is used.

We see from Fig. 3a that the *Parse XML* task is able to support a peak input rate of about 310 *tuples/sec* with just 1 thread, and increasing the number of threads actually reduces the input throughput supported, down to about 255 *tuples/sec* with 7 threads. The CPU usage even for 1 thread is high at about 85%. Here, we surmise that since a single thread is able to utilize the CPU efficiently, increasing the threads causes additional overhead for context-switching and the performance deteriorates linearly. We also see that the Parse XML task uses about 35% of memory due to Java's memory allocation for string manipulation, which is higher than for other tasks.

Pi Computation is a floating-point heavy task and uses nearly 90% CPU at the peak rate of about 105 *tuples/sec* using a single thread. However, unlike XML Parse where the peak rate linearly reduces with an increase in threads, we

see Pi manage to modestly increase the peak rate with 2 threads, to about 110 *tuples/sec*, with a similar increase in CPU usage. However, beyond 2 threads, the performance drops and remains flat with the CPU usage being flat as well. This behavior was consistent across multiple trials, and is likely due to the Intel Haswell architecture's execution pipeline characteristics [3]. The memory usage is minimal at between $2 - 10\%$.

Batch File Write is an aggregation task that is disk I/O bound. It supports a high peak rate of about $60,000$ *tuples/sec* with 1 thread, which translates to writing 6 *files/sec*, each $1\ MB$ in size. This peak rate decreases with an increase in the number of threads, but is non-linear. There is a sharp drop in the peak rate to $45,000$ *tuples/sec* with 3 threads, but this increases and stabilizes at $50,000$ *tuples/sec* with more threads. The initial drop can be attributed to the disk I/O contention, hence the drop in CPU usage as well, but beyond that the thread contention may dominate, causing an increase in CPU usage even as the supported rate is stable

The *Azure Blob* and *Azure Table* tasks rely on an external Cloud service to support their execution. As such, the throughput of these tasks are dependent on the SLA offered for these services by the Cloud provider, in addition to the local resource constraints of the slot. We see the benefit of having multiple threads clearly in these cases. The peak rate supported by both increases gradually until a threshold, beyond which the peak rate flattens and drops. Their CPU and memory utilization follow a similar trend as well. Blob's rate grows from about 2 *tuples/sec* with 1 thread to 30 *tuple/sec* with 50 threads, while Table's increases from 3 *tuples/sec* to 60 *tuples/sec* when scaling from 1 to 60 threads. This closely correlates with the SLA of the Blob service which is $60\ MB/sec$, and matches with the 30 *files/sec* of $\approx 2\ MB$ each that are cumulatively downloaded [4]. Both these tasks are also network intensive as they download data from the Cloud services.

Summary. The first three tasks show a flat or decreasing peak rate performance with some deviations, but with differing CPU and memory resource behavior. The last two exhibit a bell-curve in their peak rates as the threads increase. These highlight the distinct characteristics of specific tasks (and even specific CPU architectures and Cloud services they wrap) that necessitate such performance models to support scheduling. Simple rules of thumbs assuming static of linear scaling are inadequate, and we see later, can cause performance degradation and resource wastage.

6   Resource Allocation

Resource allocation determines the number of resource slots ρ to be allocated for a DAG $\mathcal{G} : \langle \mathbb{V}, \mathbb{E} \rangle$ for a given input rate Ω, along with the number of threads q_j required for each task $t_j \in \mathbb{V}$. In doing so, the allocation algorithm needs to

---

[3] Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel, Anand Lal Shimpi, Oct 2012, `http://www.anandtech.com/show/6355/intels-haswell-architecture/8`

[4] `https://docs.microsoft.com/en-us/azure/storage/storage-scalability-targets`

be aware of the input rate to each task that will inform it of the resource needs and data parallelism for that task. We can define this *input rate* ω_j for a task t_j based on the input rate to the DAG, the connectivity of the DAG, and the selectivity of each input stream to a task, using a recurrence relation as follows:

$$\omega_j = \begin{cases} \Omega & \text{if } \nexists e_{ij} \in \mathbb{E} \\ \sum_{e_{ij} \in \mathbb{E}} \left(\omega_i \times \sigma_{ij} \right) & \text{otherwise} \end{cases}$$

In other words, if task t_j is a source task without any incoming edges, its input rate is implicitly the rate to the DAG, Ω. Otherwise, the input rate to a downstream task is the sum of the tuple rates on the out edges of the tasks t_i immediately preceding it. This output rate is itself given by the product of those predecessor tasks' input rates ω_i and their selectivities σ_{ij} on the out edge connecting them to task t_j. This recurrence relationship can be calculated in the topological order of the DAG starting from the source task(s). Let the procedure GETRATE$(\mathcal{G}, t_j, \Omega)$ evaluate this for a given task t_j.

Next, the allocation algorithm will need to determine the threads and resources needed by each task t_j to meet its input rate ω_j. Algorithms can use prior knowledge on resource usage estimates for the task, which may be limited to the CPU% and memory% for a single thread of the task, irrespective of the input rate, or approaches like ours that use a more detailed performance model.

Say the following functions are available as a result of the performance modeling algorithm, Alg. 1, or some other means. $\mathcal{C}_i(q)$ and $\mathcal{M}_i(q)$ respectively return the incremental CPU% and memory% used by task t_i when running on a single resource slot with q threads. Further, let $\mathcal{I}_i(q)$ provide the peak input rate that is supported by the task t_i on a single slot for q number of threads. Lastly, let $\mathcal{T}_i(\omega)$ be the inverse function of $\mathcal{I}_i(q)$ such that it gives the smallest number of threads q adequate to satisfy the given input rate ω on a single resource slot. Since the ω values returned by $\mathcal{I}_i(q)$ for integer values of q would be at coarse increments, $\mathcal{T}_i$ may offer an over-estimate depending on the granularity of Δ_ω and Δ_τ used in Alg. 1.

Next, we describe two allocation algorithms – a baseline which uses simple estimates of resource usage for tasks, and another we propose that leverages the more detailed performance model available for the tasks in the DAG.

6.1 Linear Scaling Allocation (LSA)

The Linear Scaling Allocation (LSA) approach uses a simplifying assumption that the behavior of a single thread of a task will linearly scale to additional threads of the task. This scaling assumption is made both for the input rate supported by the thread, and the CPU% and memory% for the thread. For e.g., the R-Storm [48] scheduler assumes this additive behavior of resource needs for a single task thread as more threads are added, though it leaves the selection of the number of threads to the user. Other DSPS schedulers make this assumption as well [55] [9].

Algorithm 2 Linear Scaling Allocation (LSA)

1: **procedure** AllocateLSA($\mathcal{G} : \langle \mathbb{T}, \mathbb{E} \rangle$, Ω)
2:     **for** $t_i \in \mathbb{T}$ **do**   ▷ *For each task in DAG...*
3:         $\omega_i = \text{GetRate}(\mathcal{G}, t_i, \Omega)$ ▷ *Returns input rate on task t_i if DAG input rate is Ω*
4:         $\bar{\omega}_i = \mathcal{I}_i(1)$ ▷ *Peak rate supported by task t_i with 1 thread*
5:         $\tau_i \leftarrow 0$ ▷ *Allocated thread count for task t_i*
6:         $c_i \leftarrow 0$ ▷ *Estimated CPU% for τ_i threads of task t_i*
7:         $m_i \leftarrow 0$ ▷ *Estimated Memory% for τ_i threads of task t_i*
8:         **while** $\omega_i \geq \bar{\omega}_i$ **do**
9:           ▷ *One additional thread added for t_i, with increase in cumulative rate supported and resources used*
10:             $\tau_i \leftarrow \tau_i + 1$
11:             $\omega_i \leftarrow \omega_i - \bar{\omega}_i$
12:             $c_i \leftarrow c_i + \mathcal{C}_i(1)$
13:             $m_i \leftarrow m_i + \mathcal{M}_i(1)$
14:         **end while**
15:         **if** $\omega_i > 0$ **then**   ▷ *Trailing input rate below $\bar{\omega}_i$. Add thread but scale down the resources needed.*
16:             $\tau_i \leftarrow \tau_i + 1$
17:             $c_i \leftarrow c_i + \mathcal{C}_i(1) \times \dfrac{\omega_i}{\bar{\omega}_i}$
18:             $m_i \leftarrow m_i + \mathcal{M}_i(1) \times \dfrac{\omega_i}{\bar{\omega}_i}$
19:             $\omega_i \leftarrow 0$
20:         **end if**
21:     **end for**
22: **return** $\langle \tau_i, c_i, m_i \rangle \ \ \forall t_i \in \mathbb{T}$ ▷ *Return number of threads, CPU% and Memory% allocated to each task*
23: **end procedure**

Algorithm 2 shows the behavior of such a linear allocation strategy. It first estimates the input rate ω_i incident at each task t_i using the GetRate procedure discussed before. It then uses information on the peak rate $\bar{\omega}_i$ sustained by a single thread of a task t_i running in one resource slot, and its CPU% and memory% at that rate, $\mathcal{C}_i(1)$ and $\mathcal{M}_i(1)$, as a baseline. Using this, it tries to incrementally add more threads until the input rate required, in multiples of the peak rate, is satisfied (line 8). When the remaining input rate to be satisfied is below the peak rate (line 15), we linearly scale down the CPU and memory utilization, proportional to the required rate relative to the peak rate.

The result of this LSA algorithm is the thread counts τ_i per task $t_i \in \mathbb{T}$. In addition, the sum of the CPU and memory allocation for all tasks, rounded up to the nearest integer, gives the nominal lower bound on the resource slots ρ required to support this DAG at the given rate.

6.2   Model-based Allocation (MBA)

While the LSA approach is simple and appears intuitive, it suffers from two key problems that make it unsuitable for may tasks. *First, the assumption that the input rate supported will linearly increase with the number of threads is not*

valid. Based on our observations of the performance models from Fig. 3, we can see that for some tasks like Azure Blob and Table, there is a loose correlation between the number of threads and the input rate supported. But even this evens out at a certain number of threads. Others like Parse XML, Pi Computation and Batch File Write see their peak input rate supported remain flat or decrease as the threads increase *on a single resource slot* due to contention. One could expect a linear behavior if the threads run on different slots or VMs, but that would increase the slots required (and the corresponding cost).

Second, the assumption that the resource usage linearly scales with the number of threads, relative to the resources for 1 thread, is incorrect. This again follows from the performance models, and in fact, the behavior of CPU and memory usage themselves vary for a task. For e.g., in Fig. 3b for Pi, the CPU usage remains flat while the memory usage increases even as the rate supported decreases with the number of threads. For Azure Table query in Fig. 3e, the CPU and memory increase with the threads but with very different slopes.

Our Model-based Allocation algorithm, shown in Algorithm 3, avoids such inaccurate assumptions and instead uses the performance models measured for the tasks to drive its thread and resource allocation. Here, the intuition is to select the sweet spot of the number of threads such that the peak rate $\widehat{\omega}_i$ among all number of threads (for which the model is available) is the highest for task t_i (lines 9–15). This ensures that we maximize the input rate that we can support from a single resource slot for that task. At the same time, when we allocate these many threads to saturate a slot, we also disregard the actual CPU% and memory% and instead treat the whole slot as being allocated (100% usage). This is because that particular task cannot make use of additional CPU or memory resources available in that slot due to a resource contention, and we do not wish additional threads on this slot to exacerbate this problem.

When the residual input rate to be supported for a task falls below this maximum peak rate (line 16), we instead select smallest number of threads that are adequate to support this rate, and use the corresponding CPU and memory%. If a single thread is adequate (line 22), just as for LSA, we scale down the resources needed proportion to the residual input rate relative to the peak rate using 1 thread.

The result of running MBA is also a list of thread counts τ_i per task in the DAG, which will be used by the mapping algorithm. It also gives the CPU% and memory% per task which, as before, helps estimate the slots for the DAG:

$$\rho = \max \left(\left\lceil \sum_{t_i \in \mathbb{T}} (c_i) \right\rceil, \left\lceil \sum_{t_i \in \mathbb{T}} (m_i) \right\rceil \right)$$

7 Resource Mapping

As we saw, the allocation algorithm returns two pieces of information τ_i, the number of threads per task, and ρ, the number of resource slots allocated. The goal of resource mapping is to assign these task threads, $r_i^k \in R$ where $|r_i^k| = \tau_i$, to specific resource slots to meet the input rate requirements for the DAG.

Algorithm 3 Model Based Allocation (MBA)

1: **procedure** AllocateMBA($\mathcal{G} : \langle \mathbb{T}, \mathbb{E} \rangle$), Ω)
2:     **for** $t_i \in \mathbb{T}$ **do**   ▷ *For each task in DAG...*
3:         $\omega_i = \text{GetRate}(\mathcal{G}, t_i, \Omega)$ ▷ *Returns input rate on task t_i if DAG input rate is Ω*
4:         $\widehat{\omega_i} = \max_j \left\{ \mathcal{I}_i(j) \right\}$ ▷ *Maximum of peak rates supported by task t_i with any number of threads*
5:         $\widehat{\tau_i} = \mathcal{T}(\widehat{\omega_i})$ ▷ *Number of threads needed to support rate $\widehat{\omega_i}$ for task t_i*
6:         $\tau_i \leftarrow 0$ ▷ *Allocated thread count for task t_i*
7:         $c_i \leftarrow 0$ ▷ *Estimated CPU% for τ_i threads of task t_i*
8:         $m_i \leftarrow 0$ ▷ *Estimated Memory% for τ_i threads of task t_i*
9:         **while** $\omega_i \geq \widehat{\omega_i}$ **do**
10:         ▷ *Add threads for t_i corresponding to maximum peak rate. Increase cumulative rate and resources.*
11:             $\tau_i \leftarrow \tau_i + \widehat{\tau_i}$
12:             $\omega_i \leftarrow \omega_i - \widehat{\omega_i}$
13:             $c_i \leftarrow c_i + 1.00$ ▷ *Assign 100% of resource slot to these threads*
14:             $m_i \leftarrow m_i + 1.00$
15:         **end while**
16:         **if** $\omega_i > 0$ **then**   ▷ *Trailing input rate below $\widehat{\omega_i}$ to be processed for t_i*
17:             $\tau'_i \leftarrow \mathcal{T}(\omega_i)$
18:             $\tau_i \leftarrow \tau_i + \tau'_i$
19:             **if** $\tau'_i > 1$ **then**
20:                 $c_i \leftarrow c_i + \mathcal{C}_i(\tau'_i)$
21:                 $m_i \leftarrow m_i + \mathcal{M}_i(\tau'_i)$
22:             **else**   ▷ *One thread adequate. Scale down resources needed*
23:                 $c_i \leftarrow c_i + \mathcal{C}_i(1) \times \dfrac{\omega_i}{\mathcal{I}_i(1)}$
24:                 $m_i \leftarrow m_i + \mathcal{M}_i(1) \times \dfrac{\omega_i}{\mathcal{I}_i(1)}$
25:             **end if**
26:             $\omega_i \leftarrow 0$
27:         **end if**
28:     **end for**
29: **return** $\langle \tau_i, c_i, m_i \rangle$ $\forall t_i \in \mathbb{T}$ ▷ *Return number of threads, CPU% and Memory% allocated to each task*
30: **end procedure**

7.1 Resource Acquisition

The first step is to identify and acquire adequate number and sizes of VMs that have the estimated number of slots. This is straight-forward. Most IaaS Cloud providers offer on-demand VM sizes with slots that are in powers of 2, and with pricing that is a multiple of the number of slots. For e.g., the Microsoft Azure D-series VMs in the Southeast Asia data center have $1, 2, 3$ and 4 cores for sizes D1–D4, with 3.5 GB RAM per core, and costing $\$0.098, \$0.196, \$0.392$ and $\$0.784$/hour, respectively [5]. Amazon AWS and Google Compute Engine IaaS Cloud services have similar VM sizes and pricing as well. So the total price for a given slot-count ρ does not change based on the mix of VM sizes used, and one can use a simple packing algorithm to select VMs $v_j \in V$ with sizes such
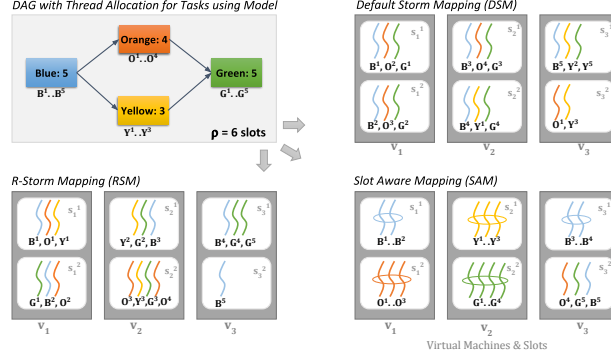
Figure 4: Mapping of a sample DAG to VMs and resource slots using DSM, RSM and SAM

that $\sum_{v_j \in V} p_j = \rho$, where p_j is the number of slots per VM v_j. At the same time, having more VMs means higher network transfer latency, and end-to-end latency for the DAG will increase. Hence, minimizing the number of distinct VMs is useful as well rather than having many small VMs.

One approach that we take to balance pricing with communication latency is to acquire as many VMs 'n' with the largest number of slots as possible, say, $\widehat{p}$, which cumulatively have $(n \times \widehat{p}) \leq \rho$. Then, for the remaining slots, we assign to the smallest possible VM whose number of slots is $\geq (\rho - n \times \widehat{p})$. This may include some additional slots than required, but is bound by $(2^{\widehat{p}-1} - 1)$ if slots per VM are in powers of 2, as is common. Other approaches are possible as well, based on the trade-off between network latency costs and slot pricing. For the $v_j \in V$ set of VMs thus acquired, let $s_j^l \in S$ be the set of slots in these VMs, where $|s_j^l| = p_j$ and $p_j \leq \widehat{p}$, such that $\left(\sum_{v_j \in V} p_j \right) \geq \rho$.

The second step, that we emphasize in this article, is to map the threads for each task to one of the slots we have acquired, and determine the mapping function $\mathcal{M} : R \to S$. Next, we discuss two mapping approaches available in literature, that we term DSM and RSM, as comparison and propose our novel mapping SAM as the third. While DSM is not "resource aware", i.e., it does not consider the resources required by each thread in performing the mapping, RSM and our SAM are resource aware, and use the output from the performance models developed earlier.

7.2   Default Storm Mapping (DSM)

DSM is the default mapping algorithm used in Apache Storm, and uses a simple round-robin algorithm to assign the threads to slots. All threads are assumed to have a similar resource requirement, and all slots are assumed to have homogeneous capacity. Under such an assumption, this naïve algorithm will balance the load of the number of threads across all slots. Algorithm 4 describes its

---
Algorithm 4 Default Storm Mapping (DSM)

1: **procedure** MAPDSM(R, S)   ▷ *Map each task thread in set R to a slot in set S*
2:    $\mathcal{M} \leftarrow new\ Map(\ )$
3:    $S'[\ ] \leftarrow$ SETTOLIST(S)   ▷ *Returns the slots in the set as a list, in some arbitrary order*
4:    $n \leftarrow 1$
5:    **for each** $r \in R$ **do**   ▷ *Round Robin mapping of threads to slots*
6:        $m \leftarrow n$ **mod** $|S|$
7:        $s = S'[m]$
8:        $\mathcal{M}.put(r \rightarrow s)$ ▷ *Assign n^{th} task thread to m^{th} resource slot*
9:        $n \leftarrow n + 1$
10:   **end for**
11: **return** $\mathcal{M}$
12: **end procedure**
---

behavior. The task threads and resource slots available are provided as a set to the algorithm. The slots are considered as a list in some random order. The algorithm iterates through the set of threads in arbitrary order. For each thread, it picks the next slot in the list and repeats this in a round-robin fashion for each pending thread (line 5), wrapping around the slot list if its end is reached.

Fig. 4 illustrates the different mapping algorithms for a sample DAG with four tasks, Blue, Orange, Yellow and Green, and say $5, 4, 3$ and 5 threads allocated to them, respectively, by some allocation algorithm. Let the resources estimated for them be 6 slots that are acquired across three VMs with 2 slots each.

Given this, the DSM algorithm first gets the list of threads and slots in some random order. For this example, let them be ordered as, $B^1, ..., B^5, O^1, ..., Y^1, ...,$ $G^1, ..., G^5$, and $s_1^1, s_1^2, s_2^1, ..., s_3^2$. Firstly, the five blue threads are distributed across the first 5 slots, $s_1^1 - s_3^1$ sequentially. Then, the distribution of the orange threads resumes with the sixth slot s_3^2, and wraps around to the first slot to end at s_2^1. The three yellow threads map to slots $s_2^2 - s_3^2$, and lastly, the five green threads wrap around and go to the first 5 slots. As we see, DSM distributes the threads evenly across all the acquired slots irrespective of the resources available on them or required by the threads, with only the trailing slots having fewer threads. This can also help distribute threads of the same task to different slots to avoid them contending for the same type of resources. However, this is optimistic and, as we have seen from the performance models, the resource usages sharply vary not just across tasks but also based on the number of threads of a task present in a slot.

7.3 R-Storm Mapping (RSM)

The R-Storm Mapping (RSM) algorithm [48] was proposed earlier as a resource-aware scheduling approach to address the deficiencies of the default Storm scheduler. It has subsequently been included as an alternative scheduler for Apache Storm, as of v1.0.1. RSM offers three features that improve upon DSM. First, in contrast to DSM that balances the thread count across all available slots, RSM

instead maximizes the resource usage in a slot and thus minimizes the number of slots required for the threads of the DAG. For this, it makes use of the resource usage for *single threads* that we collect from the performance model $(\bar{c}_i, \bar{m}_i)$, and the resource capacities of slots and VMs. A second side-effect of this is that it prevents slots from being over-allocated, which can cause an unstable DAG mapping at runtime. For e.g., DSM could place threads in a slot such that their total CPU% or memory% is greater than 100%. This is avoided by RSM. Lastly, RSM is aware of the network topology of the VMs, and it places the threads on slots such that the communication latency between adjacent tasks in the dataflow graph is reduced.

At the heart of the RSM algorithm is a *distance function* based on the available and required resources, and a network latency measure. This Euclidean distance between a given task thread $r_i^k \in R$ and a VM $v_j \in V$ is defined as:

$$d = w_M \times (M_j - \bar{m}_i)^2 + w_C \times (C_j - \bar{c}_i)^2 + w_N \times \text{NWDist}(\hat{v}, v_j)$$

where $\bar{c}_i = \mathcal{C}_i(1)$ and $\bar{m}_i = \mathcal{M}_i(1)$ are the incremental CPU% and memory% required by a single thread of the task t_i on one slot, and C_j and M_j are the CPU% and memory% not yet mapped across all slots of VM v_j. A network latency multiplier, from a *reference VM*, $\hat{v}$, to the candidate VM v_j, is also defined using the NWDist function. This reference VM is the last VM on which a task thread was mapped, and the network latency multiplier is set to 0.0 if the candidate VM is the reference VM, 0.5 if the VM is in the same rack as the reference, and 1.0 if on a different rack. Lastly, the weights w_C, w_M and w_N are coefficients to tune this distance function as appropriated.

Given this, the RSM Algorithm, given in Alg. 5, works as follows. It initializes the CPU% and memory% resources available for the candidate VMs to 100% of their number of slots, the memory available per slot to 100%, and the number of threads to be mapped per task (lines 2–4). The initial reference VM is set to some VM, say $v_1 \in V$. Then, it performs one sweep where one thread of each task, in topological order rooted at the source task(s), is mapped to a slot (lines 8–24). This mapping in the order of BFS traversal increases the chance that threads of adjacent tasks in the DAG are placed on the same VM to reduce network latency.

During the sweep, we first check if the task has any pending threads to map, and if so, we test the VMs in the ascending order of their distance function, returned by function GetSortedVMs, to see if they have adequate resources for that task thread (lines 11–15). There are two checks that are performed: one to see if the VM has adequate CPU% available for the thread, and second if any slot in that VM has enough memory% to accommodate that thread. This differentiated check is because in Storm, the memory allocation per slot is tightly bound, while the CPU% available across slots is assumed to be usable by any thread placed in that VM.

If no available slot meets the resource needs of a thread, then RSM fails. As we show later, this is not uncommon depending on the allocation algorithm. If a valid slot, s_j', is found, the task thread is mapped to this slot, and the thread

---

Algorithm 5 R-Storm Mapping (RSM)

1: **procedure** MapRSM($\mathcal{G} : \langle \mathbb{T}, \mathbb{E} \rangle$, R, V, S)
2:   $C_j = p_j \times 1.00$, $M_j = p_j \times 1.00$, $\forall v_j \in V$ ▶ *Initialize available CPU%, Memory% for all VMs*
3:   $M_j^l = 1.00$, $\forall s_j^l \in S$ ▶ *Initialize available Memory% for all slots of VMs*
4:   $\tau_i = |r_i^k|$ $\forall r_i^k \in R$ ▶ *Initialize number of task threads to map for task t_i*
5:   $\mathcal{M} \leftarrow new\ Map(\ )$ ▶ *Initialize mapping function*
6:   $\widehat{v} \leftarrow v_1$ ▶ *Initialise the reference VM to the first VM in set*
7:   **while** $\sum_{t_i \in \mathbb{T}} \tau_i > 0$ **do**   ▶ *Repeat while tasks have unmapped threads*
8:     **for each** $t_i \in$ TasksTopoOrder($\mathcal{G}$) **do**
9:       **if** $\tau_i \neq 0$ **then**
10:          ▶ *Get list of VMs sorted by distance, based on their available resources for task t_i*
11:          $V'[\ ] \leftarrow$ GetSortedVMs($V, t_i, \widehat{v}$)
12:          $s_j' \leftarrow \varnothing$
13:          **for** $v_j' \in V'[\ ]$ & $s_j' == \varnothing$ **do**
14:            $s_j' \leftarrow s_j^l \in S \mid C_j \geq \bar{c}_i$ & $M_j^l \geq \bar{m}_i$ ▶ *Does VM have CPU%, some slot in it have mem% for 1 thread?*
15:          **end for**
16:          **if** $s_j' == \varnothing$ **then return** "Error: Insufficient resources for task t_i"
17:          **end if**
18:          $r_i' \leftarrow r_i^k \in R \mid \nexists \mathcal{M}(r_i^k)$ ▶ *Pick one unmapped thread for task t_i*
19:          $\mathcal{M}.put(r_i' \rightarrow s_j')$ ▶ *Assign the thread to the selected slot with available resources*
20:          $C_j \leftarrow (C_j - \bar{c}_i)$, $M_j \leftarrow (M_j - \bar{m}_i)$, $M_j' \leftarrow (M_j' - \bar{m}_i)$ ▶ *Reduce available resources by 1 thread*
21:          $\tau_i \leftarrow \tau_i - 1$
22:          $\widehat{v} \leftarrow v_j'$ ▶ *Update reference node to be the current mapped VM*
23:        **end if**
24:      **end for**
25:    **end while**
26: **return** $\mathcal{M}$
27: **end procedure**

---

count and resource availability updated (lines 20– 21). The reference VM is also set to the current VM having that slot. Then the next threads in the sweep are mapped, and this process repeated till all task threads are mapped.

Fig. 4 shows the RSM algorithm applied to the sample DAG. Mapping of the threads to slots is done in BFS ordering of tasks, B, O, Y and G. For each thread of the task in this order, a slot on the VM with the minimum distance and available resources is chosen. Say in the first sweep, the threads B^1, O^1 and Y^1 are mapped to the same slot s_1^1, and then next thread G^1 be mapped to new slot s_1^2 due to resource constraint on s_1^1 for this thread. The new slot s_1^2 is picked on same VM as it has the least distance among all VMs. In the second sweep, thread B^2, O^2, Y^2 and G^2 are mapped to slots s_1^2 and s_2^1, and likewise for the third sweep. In the fourth sweep, there are no threads for the Yellow task pending. Also, we see that thread B^4 is not mapped to slot s_2^2 due to lack of resources, instead going to s_3^1. However, a slot s_2^2 does have resources for a subsequent thread O^4, and the distance to s_2^2 is lesser than s_3^1. Thus RSM tries to perform a network-aware best fit packing to minimize the number of slots.

7.4 Slot Aware Mapping (SAM)

While the RSM algorithm is resource aware, it linearly extrapolates the resource usage for multiple threads of a task in a VM or slot based on the behavior of a single thread. As we saw earlier in the § 5, this assumption does not hold, and as we show later in § 8, it causes inefficient mapping, non-deterministic performance and needs over-allocation of resources. Our Slot Aware Mapping (SAM) addresses these deficiencies by fully utilizing the performance model and the strategy used by our model based allocation. They key idea is to map a *bundle of threads* of the same task exclusively to a single slot such that the stream throughput is maximized for that task on that slot based on its performance model, and the interference from threads of other tasks on that slot is reduced.

In Algorithm 6, as for RSM, we initialize the resources for the VMs and slots. Further, in addition to the total slots ρ required by the DAG, we also have the quantity of CPU% and memory% required by all the threads of each task available as c_i and m_i. Recollect that the MBA algorithm returns this information based on the performance model. As for RSM, we iterate through the tasks in topological order (line 7). However, rather than map one thread of each task, we first check if the number of pending threads forms a *full bundle*, which we define to be as $\widehat{\omega}_i$, the number of threads at the peak rate supported by the task on a single slot (line 8). If so, we select an empty slot in the last mapped VM, or if none exist, in its neighboring one (line 9). We $\widehat{\omega}_i$ unmapped threads for this task and assign this whole bundle of threads to this exclusive slot, i.e., 100% of its CPU and memory (line 13). The resource needs of the task are reduced concomitantly, and this slot is fully mapped.

It is possible that the task has a *partial bundle* of unmapped threads, having fewer than $\widehat{\omega}_i$ ones (line 17). In this case, we find the best-fit slot as the one whose sum of available CPU% and memory% is the smallest, while being adequate for the CPU% and memory% required for this partial bundle (line 18). We assign this partial bundle of threads to this slot and reduce the resources available for this slot by c_i and m_i. At this point, all threads of this task will be assigned (line 24).

Notice that slots co-locate threads from different tasks only for the last partial bundle of each task. So we have an upper bound on the number of slots with mixed thread types as $|\mathbb{V}|$. Since the performance models offers information on the behavior of the same thread type on a slot, this limits the interference between threads of different types, that is not captured by the model. In practice, as we show in the experiments, most slots have threads of a single task type. As a result, SAM has a more predictable resource usage and behavior for the mapped DAG.

It is possible that even in SAM, the resources allocated may not be adequate for the mapping (lines 10, 19), though the chances of this happening is smaller than for RSM since SAM uses a strategy similar to the allocation algorithm, MBA. This is a side-effect of the binning, when resource available in partly used slots are not adequate to fully fit a partial bundle. Also, while we do not

explicitly consider network distance unlike in RSM, the mapping of tasks in topological order combined with picking a bundle at a time achieves a degree of network proximity between threads of adjacent tasks in the DAG.

Algorithm 6 Slot Aware Mapping (SAM)

1: **procedure** MAPSAM($\mathcal{G} : \langle \mathbb{T}, \mathbb{E} \rangle$, R, V, S)   ▶ c_i and m_i are CPU% and memory% required by task t_i from MBA
2:     $\tau_i = |r_i^k| \quad \forall r_i^k \in R$ ▶ Initialize number of task threads to map for task t_i
3:     $\widehat{\tau}_i = \mathcal{T}(\widehat{\omega}_i)$ ▶ Number of threads needed to support peak rate $\widehat{\omega}_i$ for task t_i on 1 slot
4:     $C_j^l = 1.00, \quad M_j^l = 1.00, \quad \forall s_j^l \in S$ ▶ Initialize available CPU%, Memory% for all slots of VMs
5:     $\mathcal{M} \leftarrow new\ Map(\ )$ ▶ Initialize mapping function
6:     **while** $\sum_{t_i \in \mathbb{T}} \tau_i > 0$ **do**   ▶ Repeat while tasks have unmapped threads
7:         **for each** $t_i \in$ TASKSTOPOORDER($\mathcal{G}$) **do**
8:             **if** $\tau_i \geq \widehat{\tau}_i$ **then**   ▶ At least 1 full bundle of threads remains for task t_i
9:                 $s_j' \leftarrow$ GETNEXTFULLSLOT(V)   ▶ Returns next full slot in current or next VM
10:                 **if** $s_j' == \varnothing$ **then return** "Error: Insufficient resources for task t_i"
11:                 **end if**
12:                 $r_i'[\ ] \leftarrow \{r_i^k\} \in R \mid \nexists \mathcal{M}(r_i^k), \ |r_i'| = \widehat{\tau}_i$ ▶ Pick unmapped full bundle of $\widehat{\tau}_i$ threads for task t_i
13:                 $\mathcal{M}.putAll(r_i'[\ ] \rightarrow s_j')$ ▶ Assign threads in bundle to the selected slot
14:                 $\tau_i \leftarrow \tau_i - \widehat{\tau}_i, \quad c_i \leftarrow c_i - 1.00, \quad m_i \leftarrow m_i - 1.00$ ▶ Reduce resource needs for bundle
15:                 $C_j' \leftarrow 0.00, \quad M_j' \leftarrow 0.00$ ▶ Used all resources in slot
16:             **else**
17:                 **if** $\tau_i > 0$ **then**   ▶ Partial bundle of threads remains for task t_i
18:                     $s_j' \leftarrow$ GETBESTFITSLOT(V, c_i, m_i)   ▶ Find smallest slot with sufficient resources for partial bundle
19:                     **if** $s_j' == \varnothing$ **then return** "Error: Insufficient resources for task t_i"
20:                     **end if**
21:                     $r_i'[\ ] \leftarrow \{r_i^k\} \in R \mid \nexists \mathcal{M}(r_i^k)$ ▶ Pick all remaining threads for task t_i
22:                     $\mathcal{M}.putAll(r_i'[\ ] \rightarrow s_j')$
23:                     $C_j' \leftarrow C_j' - c_i, \quad M_j' \leftarrow M_j' - m_i$ ▶ Reduce resources in slot by partial bundle
24:                     $\tau_i \leftarrow 0, \quad c_i \leftarrow 0.00, \quad m_i \leftarrow 0.00$ ▶ All done for this task
25:                 **end if**
26:             **end if**
27:         **end for**
28:     **end while**
29: **return** $\mathcal{M}$
30: **end procedure**

Fig. 4 shows the SAM algorithm applied to the sample DAG. Say a full bundle of the four tasks, B, O, Y and G, have 2, 3, 3 and 4 threads, respectively. We iteratively consider each task in the BFS order of the DAG, similar to RSM, and attempt to assign a full bundle from their remaining threads to an exclusive slot. For e.g., in the first sweep, the full bundles $B^1..B^2, O^1..O^3, Y^1..Y^3, G^1..G^4$ are mapped to the four slots, $s_1^1, s_1^2, s_2^1, s_2^2$, respectively, occupying 2 VMs. In the next sweep, we still have a full bundle for the Blue task, $B^3..B^4$, that takes an

independent slot s_3^1, but the Orange and Green tasks have only partial bundle consisting of one thread each. O^4 is mapped to a new slot s_3^2 as there are no partial slots, and G^5 is mapped to the same slot as it is the best-fit partial slot. All threads of the Yellow task are already mapped. In the last sweep, the only remaining partial bundle for the Blue task, B^5 is mapped to the partial slot s_3^2 as the best fit.

8 Results and Analysis

8.1 Implementation

We validate our proposed allocation and mapping techniques, MBA and SAM, on the popular Apache Storm DSPS, open-sourced by Twitter. Streaming applications in Storm, also called *topologies*, are composed in Java as a DAG, and the resource allocation – number of threads per task (*parallelism hint*) and the resource slots for the topology (*workers*) – is provided by the user as part of the application. Here, we implement our MBA algorithm within a script that takes the DAG and the performance model for the tasks as input, and returns the number of threads and slots required. We manually embed this information in the application, though this can be automated in future. We take a similar approach and implement the LSA algorithm as well, which is used as a baseline.

A Storm cluster has multiple hosts or VMs, one of which is the master and the rest are compute VMs having one or more resource slots. When the application is submitted to the Storm cluster, the master VM runs the *Nimbus scheduling service* responsible for mapping the threads of the application's tasks to worker slots. A *supervisor service* on each compute VM receives the mapping from Nimbus and assigns threads of the DAG for execution. While it is possible to run multiple topologies concurrently in a cluster, our goal is to run each application on an exclusive on-demand Storm cluster with the exact number of required VMs and slots, determined based on the allocation algorithm. For e.g., one scenario is to acquire a Storm cluster on-demand from Azure's HDInsight Platform as a Service (PaaS) [6].

Nimbus natively implements the default round-robin scheduler (DSM) and recently, the scheduling algorithm of R-Storm (RSM) using the `DefaultScheduler` and `ResourceAwareScheduler` classes, respectively. We implement our SAM algorithm as a custom scheduler in Nimbus, `SlotAwareScheduler`. It implements the `schedule` method of the `IScheduler` interface which is periodically invoked by the Nimbus service with the pending list of threads to be scheduled. When a thread for the DAG first arrives for mapping, the SAM scheduler generates and caches a mapping for all the threads in the given DAG to slots available in the cluster. The host IDs and slot IDs available in the cluster is retrieved using methods in Storm's `Cluster` class. Then, the algorithm groups the threads by their slot ID as Storm requires all thread for a slot to be mapped at once. The

---

[6]Apache Storm for HDInsight, `https://azure.microsoft.com/en-in/services/hdinsight/apache-storm/`

actual mapping is enacted by calling the `assign` method of the `Cluster` class that takes the slot ID and the list of threads mapped to it.

8.2 Experiment Setup

The setup for validating and comparing the allocation and mapping algorithms are similar to the one used for performance modeling, § 5.2. In summary, Apache Storm $v1.0.1$ is deployed on Microsoft Azure D-series VMs in the Southeast Asia data center. The type and number of VMs depend on the experiment, but each slot of this VM series has one core of Intel Xeon E5-2673 v3 CPU @2.4 GHz, 3.5 GB RAM and a 50 GB SSD. We use three VM sizes in our experiments to keep the costs manageable – `D3` having $2^{(3-1)} = 4$ slots, `D2` with 2 slots, and `D1` with 1 slot.

We perform two sets of experiments. In the first, we evaluate the resource benefits of our Model Based Allocation (MBA) combined with our Slot Aware Mapping (SAM), in comparison with the baseline Linear Storm Allocation (LSA) with the resource-aware R-Storm Mapping (RSM), for a given input rate (§ 8.4). For the allocated number of resource slots, we acquire the largest size VMs (`D3`) first to meet the needs, and finally pick a `D2` or a `D1` VM for the remaining slot(s), as discussed in § 7.1.

In the second set of experiments (§ 8.5), we verify the predictability of the performance of our MBA and SAM approaches, relative to the existing techniques. Here, we perform five experiments, using a combination of 2 allocation and 3 mapping algorithms. We measure the highest stable input rate supported by the DAGs using these algorithms on a fixed number of five `D3` VMs, and compare the observed rate and resource usage against the planned rate, and with the rate and usage estimated by our model.

In all experiments, a separate `D3` VM is used as the master node on which the Nimbus scheduler and other shared services run. For the RSM implementation, we need to explicitly specify the memory available to a slot and to the VM, which we set to 3.5 GB and the number of slots times 3.5 GB, respectively. For RSM and SAM, we set the available CPU% per slot to 100% and for the VM to be the number of VM slots times 100%.

8.3 Streaming Applications

In our experiments, we use two types of streaming dataflows – *micro-DAGs* and *application DAGs*. The micro-DAGs capture common dataflow patterns that are found as sub-graphs in larger applications, and are commonly used in literature, including by R-Storm [2,48,72]. These include *Linear*, *Diamond* and *Star* micro-DAGs that respectively capture a sequential flow, a fan-out and fan-in, and a hub-and-spoke model (Fig. 5). While the linear DAG has a uniform input and output tuple rate for all tasks, the diamond exploits task parallelism, and the star doubles the input and output rates for the hub task. All three micro-DAGs have 5 tasks, in addition to the source and sink tasks, and we randomly assign the five different tasks that were modeled in § 5 to these five
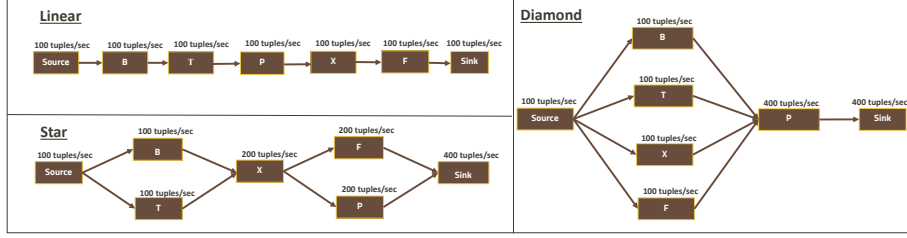
Figure 5: Micro DAG used in experiments. Tasks are referred to by their initials: B-Azure Blob Download, F-Batched File Write , P-Pi Computation, T-Azure Table Query, X-XMLparse. Since selectivity is 1:1, the input and output rates are the same, and indicated above the task
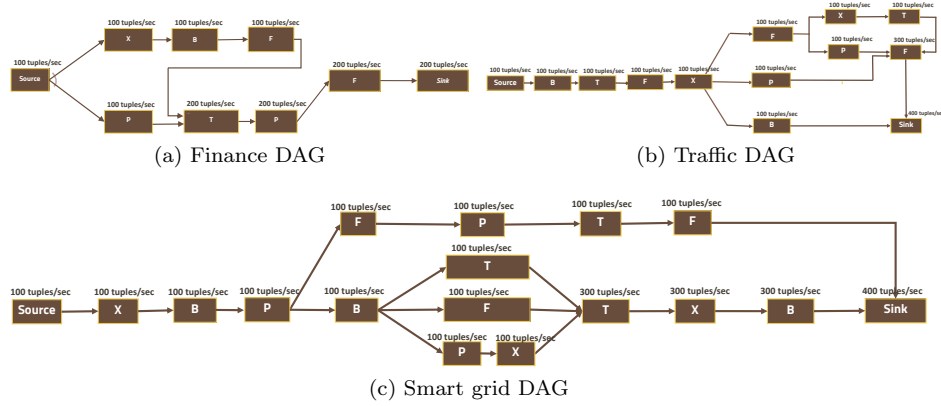

(a) Finance DAG

(b) Traffic DAG

(c) Smart grid DAG

Figure 6: Application DAGs [Notation: B-Azure Blob Download, F-Batched FileWrite , P-Pi Computation, T-Azure Table Query, X-XMLparse]

DAG vertices, as labeled in Fig. 5. The figure also shows the input rates to each task based on a sample input rate of 100 $tuples/sec$ to the DAG. All tasks have a selectivity of $\sigma = 1 : 1$.

The application DAGs have a structure based on three real-world streaming applications that analyze traffic patterns from GPS sensor streams (*Traffic*) [7], compute the bargain index value from real-time stock trading prices (*Finance*) [27], and perform data pre-processing and predictive analytics over electricity meter and weather data streams from Smart Power Grids (*Grid*) [57]. In the absence of access to the actual application logic, we reuse and iteratively assign the five tasks we have modeled earlier to random vertices in these application DAGs and use a task selectivity of $\sigma = 1 : 1$.

These three applications DAGs have between $7 - 15$ logic tasks, and exhibit different composition patterns. Their overall DAG selectivity ranges from $1 : 2$ to $1 : 4$. We also see that the five diverse tasks we have chosen as proxies for these

domain tasks are representative of the native tasks, as described in literature. For e.g., a task of the Traffic application does parsing of input streams, similar to our XML Parse task, and another archives data for historical analysis, similar to the Batch File Write task. The moving average price and bargain index value tasks in the Finance DAG are floating-point intensive like the Pi task. The Grid DAG performs parsing and database operations, similar to XML Parse and Azure Table, as well as time-series analytics that tend to be floating-point oriented. As a result, these are reasonable equivalents of real-world applications for evaluating resource scheduling algorithms.

Along with the application logic tasks, we have separate source and sink tasks for passing the input stream and logging the output stream for all DAGs. The source task generates synthetic tuples with a single opaque field of 10 bytes at a given constant rate. The sink task logs the output tuples and helps calculate the latency and other empirical statistics. Both these tasks are mapped by the scheduler just like the application tasks. Given the light-weight nature of these tasks, we empirically observe that a single thread for each of these tasks is adequate, with a static allocation of 10% CPU and 15% memory for the source and 10% CPU and 20% memory for the sink.

Each experiment is run for 15 minutes and over multiple runs, and we report the representative values seen for the entire experiment.

8.4 Resource Benefits of Allocation and Mapping

We compare our combination of MBA allocation and SAM mapping, henceforth called **MBA+SAM**, against LSA allocation with RSM mapping, referred to as **LSA+RSM**. The metric of success here is the ability to *minimize the overall resources allocated* for a stable execution of the DAG at a given fixed input rate. We consider both micro-DAGs and applications DAGs. First the allocation algorithm determines the minimum number of resource slots required and then the mapping algorithm is used to schedule the threads on slots for the DAG. There may be cases where the resource-aware mapping algorithm is unable to find a valid schedule for the resource allocation, in which case, we incrementally increase the number of slots by 1 until the mapping is successful. We report and discuss this as well. We then execute the DAG and check if it is able to support the expected input rate or not.

8.4.1 Micro DAG

The experiments are run for the micro-DAGs with input rates of $50, 100$ and 200 *tuples/sec*. This allows us to study the changes in resource allocation and usage as the input rate changes. These specific rates are chosen to offer diversity while keeping within reasonable VM costs. For e.g., each run costs $\approx US\$1.00$, and many such runs are required during the course of these experiments.

Fig. 7 shows the number of resource slots allocated by the LSA and MBA algorithms (yellow bars, left Y axis) for the three different input rates to the three micro-DAGs. Further, it shows the additional slots beyond this allocation

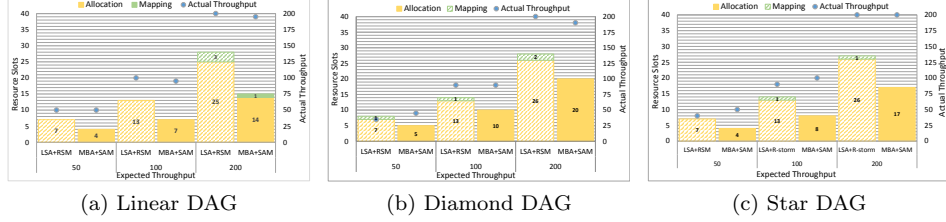(a) Linear DAG (b) Diamond DAG (c) Star DAG

Figure 7: Micro DAGs: Required Slots on primary, Actual throughput on secondary Y-axis

(green bars, left Y axis) that is required by the resource-aware RSM and SAM mapping algorithms to ensure that threads in a slot are not under-provisioned. The DAGs are run with the input rate they the schedule was planned for (i.e., 50, 100 or 200 *tuples/sec*), and if they were not stable, we incrementally reduced the rate by 5 *tuples/sec* until the execution is stable. The stable input rate, less than or equal to the planned schedule, is shown on the right Y axis (blue circle).

We can observe that LSA allocates more slots than MBA in all cases. In fact, the resources allocated by LSA is nearly twice as that by MBA requiring, say, 7, 13 and 28 slots respectively for the Linear DAG for the rates of 50, 100 and 200 *tuples/sec* compared to MBA that correspondingly allocates only 4, 7 and 15 slots. This is observed for the other two micro-DAGs as well. The primary reason for this resource over-allocation in LSA is due to a linear extrapolation of resources with the number of threads. In fact, while MBA allocates $\approx 3\times$ more threads than LSA for the DAGs, the resource allocation for these threads by LSA is much higher. For e.g., LSA allocates 337% CPU and 1196% memory for its 50 threads of the Blob Download task for the Linear DAG at 100 *tuples/sec* while MBA allocates only 315% of CPU and 326% of memory for its corresponding 170 Blob Download threads. This alone translates to a difference between ≈ 12 slots allocated by LSA (based on memory%) and ≈ 3 slots by MBA.

Despite the higher allocation by LSA, we see that RSM is often unable to complete the mapping without requiring additional slots. This happens for 6 of the 9 combination of DAG and rates, for e.g., requiring 1 more slot for the Diamond DAG with 50 *tuples/sec* (Fig. 7b, green bar) and 3 more for the Linear DAG at 200 *tuples/sec* (Fig. 7a). In contrast, our SAM mapping uses 1 additional slot, only in the case of Linear DAG at 50 *tuples/sec* (Fig. 7a) and none other, despite being allocated fewer resources by MBA compared to LSA.

Both RSM and SAM are resource aware, which means they will fail if they are unable to pack threads on to allocated slots such that their expected resource usage by all threads on a slot is within the slot's capacity. RSM more often fails to find a valid bin-packing than SAM. This is because of its distribution of a task's threads across many slots, based on the distance function, which causes resource fragmentation. We see memory fragmentation to be more common,

32

causing vacant holes in slots that are each inadequate to map a thread but are cumulatively are sufficient.

For e.g., the Linear DAG at 200 *tuples/sec* is assigned 25 slots by LSA. During mapping by RSM, the 100 Blob Download threads dominate the memory and occupy $4 \times 23.9\%$ of memory in each of the 25 slots, leaving only 8% memory on each slot. This is inadequate to fit threads for other tasks like XML Parse which requires 22.98% of memory for one of its thread, though over $25 \times 8\% = 200\%$ of fragmented memory is available across slots.

This happens much less frequently in SAM due to its preference for packing a slot at a time with a full bundle of threads in a single slot without any fragmentation. Fragmenting can only happen for the last partial thread bundle for each task. A full bundle also takes less resources according to the performance models than linear extrapolation from a single thread. For e.g., MBA packs 50 threads of the same Blob Download task from above in a single slot.

We see that in several cases, the DAGs are unable to support the rate that the schedule was planned for. This reduction in rate is up to 30% for LSA+RSM and up to 10% for MBA+SAM. For e.g., in the Diamond DAG in Fig. 7b, the observed/expected rates in *tuples/sec* for LSA+RSM is 35/50 and 90/100 while it is 90/100 and 190/200 for MBA+SAM. The reasons vary between the two approaches.

In LSA+RSM, LSA allocates threads assuming a linear scaling of the rate with the threads but this holds only if each thread is running on an exclusive slot. As RSM packs multiple threads to a slot, the rate supported by these threads is often lower than estimated. For e.g., for the Diamond DAG in Fig. 7b, LSA allocates 18 threads for the Azure Table task for an input rate of 50 *tuples/sec* based on a single thread supporting 3 *tuples/sec*. However, RSM distributes 2 threads each on 4 slots and the remaining 9 threads on 1 slot. As per our performance model for the Azure Table task, 2 threads on a slot support 5 *tuples/sec* and 9 threads support 10 *tuples/sec*, to give a total of $4 \times 5 + 1 \times 10 = 30$ *tuples/sec*. This is close to the observed 35 *tuples/sec* supported by this DAG for LSA+RSM.

While SAM's model-based mapping of thread bundles mitigates this issue, it does suffer from an imbalance in message routing by Storm to slots. Storm's *shuffle grouping* from an upstream task sends an equal number of tuples to each downstream thread. However, the individual threads may not have the same per-capita capacity to process that many tuples on its assigned slot, as seen from the performance models. This can cause a mismatch between tuples that arrive and those that can be processed on slots.

For e.g., the Diamond DAG at 100 *tuples/sec* (Fig. 7b), MBA allocates 160 threads for the Azure Table task and SAM maps two full bundles of 60 threads each to 2 slots, and the remaining 40 threads on 1 partial slot. As per the model, SAM expects the threads in a full slot to support 40 *tuples/sec* and the partial slot to support 20 *tuples/sec*. However, due to Storm's shuffle grouping, the full slots receive 37 *tuples/sec* while the partial slot receives 26 *tuples/sec*. This problem does not affect RSM since it distributes threads across many slots achieving a degree of balance across slots. As future work, it is worth considering
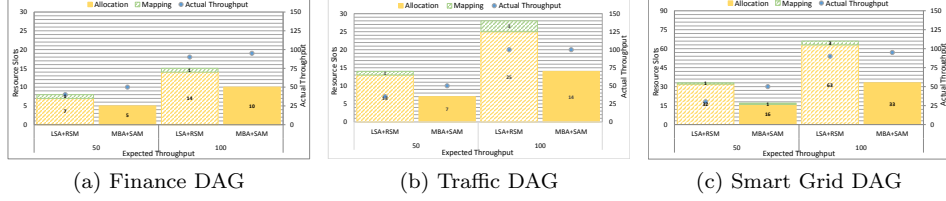
	(a) Finance DAG	(b) Traffic DAG	(c) Smart Grid DAG

Figure 8: Application DAGs: Required Slots on primary, Actual throughput on secondary Y-axis

a slot-aware *routing* in Storm as well [7].

Also Figs. 7 show that as expected, *the resource requirements increase proportionally with the input rate for both LSA+RSM and MBA+SAM.* Some minor variations exist due to rounding up of partial slots to whole, or marginal differences in fragmentation. For e.g., LSA+RSM assigns the Star DAG $\lceil 6.23 \rceil = 7$ *slots* and $\lceil 12.47 \rceil = 13$ *slots* for 50 and 100 *tuples/sec* rates, respectively.

Lastly, we also observe that all three micro-DAGs acquire about the same number of slots, for a given rate, using LSA+RSM, e.g., using about 7 slots for 50 *tuples/sec* for all three micro-DAGs. These three DAGs have the same 5 tasks though their composition pattern is different. However, for LSA, the memory% of the Blob Download task threads dominates the resource requirements for each DAG, and the input rate to this task is the same as the DAG input rate in all cases. As a result, for a given rate, the threads and resource needs for this task is the same for all three DAGs at 25 threads taking 598% memory for 50 *tuples/sec*, while the memory% for the entire Linear DAG is marginally higher at 623% for this rate and its total CPU% need is only 242%. Hence, the resource needs for all other tasks, which are more CPU heavy, falls within the available 7 slots that is dominated by this Blob task, i.e., $\sum_{\text{All task threads}} CPU\% < \sum_{\text{Blob task threads}} Memory\%$.

In case of MBA+SAM, there is diversity both in CPU and memory utilization, and the number of threads for each task for the different DAGs. So the resource requirements are not overwhelmed by a single task. For e.g., the same Blob task at 50 *tuples/sec* requires only 128% memory according to MBA while the CPU% required for the entire Linear DAG is 323%, which becomes the deciding factor for slot allocation.

8.4.2 Application DAG

We perform similar experiments to analyze the resource benefits for more complex applications DAGs, limited to input rates of 50 and 100 *tuples/sec* that require as much as 65 slots for LSA+RSM costing $\approx US\$2$ per run. Figs. 8 plot the results. Several of our observations here mimic the behavior of the scheduling approaches for the micro-DAGs, but at a larger scale. We also see

---

[7] https://issues.apache.org/jira/browse/STORM-162

more diversity across the DAGs, with Finance taking $3 - 5\times$ fewer resources than Grid for the same data rates.

As before for micro-DAGs, we see that MBA+SAM consistently uses $33 - 50\%$ fewer slots than LSA+RSM for all the application DAGs and rates. This is seen both for the allocation and in the incremental slots acquired by the mapping during packing. In fact, RSM acquires additional slots for all application DAGs allocated by LSA while MBA needs this only for Grid DAG at 50 $tuples/sec$. The resource benefit for MBA+SAM relative to LSA+RSM is the least for the Finance DAG in Fig. 8a, using 3 fewer slots for 50 $tuples/sec$ rate and 5 fewer for 100 $tuples/sec$ rate. This is because total CPU% tends to dominate for MBA+SAM, and this is higher for Finance compared to the other two due to a higher input rate that arrives at the compute-intensive Pi task. On the other hand, LSA+RSM consumes over twice the slots for Traffic and Grid DAGs due having one less Pi task and one more Blob task, which is memory intensive. This is due to random mapping of our candidate tasks to application tasks. As we saw earlier for the micro-DAG, this memory intensive task tends to be sub-optimal when bin-packing by RSM, and that causes a the resource needs to grow for the Traffic and Grid DAGs.

That said, while the DAGs are complex for these application workloads, the fraction of additional slots required by mapping relative to allocation does not grow much higher. In fact, the additional slots required by RSM is small in absolute numbers, at $1-3$ slots, as the bin-packing efficiency improves with more slots and threads. This shows that the punitive impact of RSM's additional slot requirements is mitigated for larger application DAGs.

As for the micro-DAGs, several of the application DAGs are also unable to support the planned input rate. The impact worsens for LSA+RSM with its stable observed rate up to 40% below expected, while this impact is much smaller for MBA+SAM with only up to 5% lower rate observed despite the complex DAG structure.

The reasoning for SAM is the same as for the micro-DAGs, where the shuffle grouping unformly distributes the output tuple rate across all threads. For RSM, an additional factor comes into play for these larger DAGs. In practice, this algorithm allows threads in a slot to access all cores in a VM while restraining their memory use to only that slot's limit. This means threads in a single slot of a `D3` VM can consume up to 400% CPU, as long as their memory% is $\leq 100\%$. This causes more CPU bound threads like Pi and XMLParse to be mapped to a single slot, consuming $\approx 300\%$ of a VM's CPU in the Grid DAG for 50 $tuples/sec$. However, each slot has just a single worker thread responsible for tuple buffering and routing between threads and across workers. Having many CPU intensive task threads on the same slot stresses this worker thread and cause a slowdown, as seen for the Grid DAG which has an observe/expected tuple rates of $30/50$ [8] [9]. This consistently happens across all VMs where threads with high CPU% are over-allocated to a single slot. In MBA, the mapping of

---

[8]`http://stackoverflow.com/questions/20371073/how-to-tune-the-parallelism-hint-in-storm`
[9]`http://mail-archives.apache.org/mod_mbox/storm-user/201606.mbox/browser`

full bundles to a slot rather than over-allocating CPU% means that we have a better estimate of the collective behaviour of threads on each worker slot and these side-effects are avoided.

As before, we see that the resource requirements increase proportionally as the rate doubles from 50 $tuples/sec$ to 100 $tuples/sec$ in most cases. However, unlike the micro-DAGs where all the dataflows for a given input rate consumed about the same number of slots using LSA+RSM, this is not the case for the application DAGs. Here, the number of tasks of each type vary and their complex compositions cause much higher diversity in input rates to these tasks. For e.g., the same Table task in Traffic, Finance and Smart Grid DAGs in Figs 6 have input rates of 100, 200 and 300 $tuples/sec$. In fact, this complexity means that the resource usage does not just proportionally increase with the number of tasks either. This argues the need for non-trivial resource- and DAG-aware scheduling strategies for streaming applications, such as RSM, MBA and SAM.

8.5   Accuracy of Models

In the previous experiments, we showed that our MBA+SAM scheduling approach offered *lower resource costs* than the LSA+RSM scheduler while meeting the planned input rate more often. In these experiments, we show that our model based scheduling approach offers *predictable resource utilization*, and consequently *reliable performance* that can be generalized to other DAGs and data rates. Further, we also show that it is possible to independently use our performance-model technique to accurately predict the resource usage and supported rates for other scheduling algorithms as well.

Rather than determine the allocation for a given application and rate, we instead design these experiments with a fixed number of VMs and slots – five D3 VMs with 20 total slots, for the three micro-DAGs. We then examine the highest input rate that our performance model estimates will be supported by the given schedule, and what is actually supported on enacting the schedules.

The *planned input rate* is the peak rate for which the DAG's resource requirements can be fulfilled with the fixed number of five D3 VMs, according to the allocation+mapping algorithm pair that we consider. For this, we independently run the allocation *and* mapping algorithm plans outside Storm, adding incremental input rates of 10 $tuples/sec$ until the resources required is just within or equal to 20 slots according to the respective algorithm pairs. Subsequently, for the threads and their slot mappings determined by the scheduling algorithm, we use our performance models to find the *predicted rate* supported by that DAG. We also use our model to *predict the CPU% and memory%* for the slots as well, and report the cumulative value for each of the 5 VMs. The actual input rate for the DAG is obtained empirically by increasing the rate in steps of 10 $tuples/sec$ as long as the execution remains stable. The actual CPU and memory utilization corresponding to the peak rate supported is reported for each VM as well. Besides comparing the predicted and actual input rates, we also compare the predicted and actual VM resource usage in the analysis since there is a causal effect of the latter on the former.
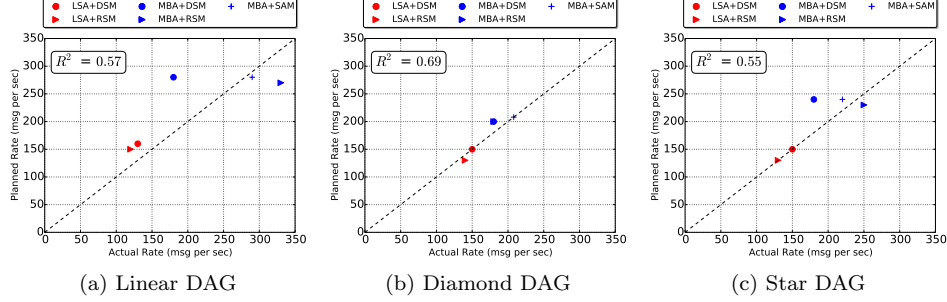
36

Figure 9: Scatter plot of *Planned* and *Actual* input rates supported for the Micro-DAGs on 5 VMs using the scheduling strategy pairs
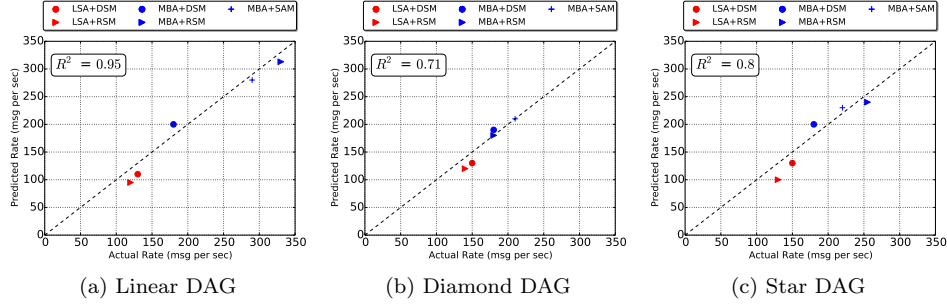

Figure 10: Scatter plot of *Predicted* and *Actual* input rates supported for the Micro-DAGs on 5 VMs using the scheduling strategy pairs

We further show that our model based allocation algorithm can be used independently with other mapping algorithms, besides SAM. To this end, we evaluate and compare the baseline combination of LSA allocation with DSM and RSM mappings available in Storm (LSA+DSM and LSA+RSM), against our MBA allocation with DSM, RSM and SAM mapping algorithms (MBA+DSM, MBA+RSM and MBA+SAM).

8.5.1 Prediction and Comparison of Input Rates

Fig. 9 shows a scatter plot comparing the *Actual rate* (X axis) and the *Planned rate* by the scheduling algorithm (Y axis) for the Linear, Diamond and Star micro-DAGs, while Fig. 10 does the same for the *Actual rate* (X axis) and our *Model Predicted rate* (Y axis). We see that our performance model is able to *accurately predict the input rate for these DAGs with a high correlation coefficient of R^2 ranging from $0.71 - 0.95$.* This is actually significantly better than the Planned rate by the schedulers for the three DAGs whose R^2 values fall between $0.55 - 0.69$. Thus, our performance model is able to accurately predict

the input rates for the schedules from all 5 algorithm pairs, better than even the scheduling algorithms themselves.

There are also algorithm-specific behavior of the prediction models, which we analyze. The input rate predictions are more accurate for MBA+SAM (Fig. 10, blue '+'), falling close to the 1:1 line in all cases, since it uses the model both for allocation and for mapping. However, it is not 100% accurate due to Storm's shuffle grouping that routes a different rate to downstream threads than expected.

We also see our model underestimate the supported rate for LSA in Fig. 10 by a small fraction. This happens due to the granularity of the model. With LSA, several slots have 3 table threads mapped to them. As we do not have exact performance models with 3 threads, we interpolate between the available thread values which estimates the rate supported at 6 *tuples/sec* while the observed rate is closer to 9 *tuples/sec*. In such cases, the predictions can be made more accurate if the performance modeling is done at a finer granularity of thread counts (Δ_τ) in Algo. 1.

The algorithms also show distinctions in the actual rates that they support the same quanta of resources for a DAG. When using MBA with DSM, the actual rate is often much smaller than the planned rate and, to a lesser extent, than the predicted rate as well. For e.g., the Linear DAG's planned rate is 280 *tuples/sec*, predicted is 200 *tuples/sec* and actual is 180 *tuple/sec*. Since DSM does a round-robin mapping of threads without regard to the model, it is unable to leverage the full potential of the allocation. In the case of the Linear DAG, the allocation estimates the planned performance for, say, the Blob Download task with 470 *threads* based on it being distributed in bundles of 50 threads each on 9 slots but DSM assigns them uniformly with ≈ 23 *threads per slot*. Hence, using MBA with DSM is not advisable, compared to RSM or SAM mapping approaches.

However, compared to LSA, MBA offers a higher predicted and actual input rate irrespective of the mapping, offering improvements of $20 - 175\%$. We observe from the plots that the cluster of points for MBA (in blue) is consistently at a higher rate than the LSA cluster (in red) despite both being allocated the same fixed number of resources. As discussed before, LSA allocates fewer data-parallel threads than MBA due to its linear-scaling assumption, and they are unable to fully exploit the available resources. This is consistent with the lower CPU% and memory% for LSA observed in Figs. 11 and 12. For e.g., the Azure Table task in the Linear DAG is assigned only 54 threads by LSA, with a planned rate of 160 *tuples/sec* whereas MBA assigns it 420 threads with a planned rate of 280 *tuples/sec*.

Interestingly, when using MBA, RSM is able to offer a higher actual input rate compared to SAM in two of the three DAGs, Linear and Star (Fig. 10), even as its planned rate is lower than SAM. For e.g., we see that RSM's distance measure is able find the sweet spot for distributing the 470 *threads* of Blob Download for the Linear DAG across 15 slots with $25 - 30$ threads each and 3 slots with under 10 threads each, to offer a predicted rate of 315 *tuples/sec* and actual rate of 330 *tuples/sec*. SAM on the other hand favors predictable perfor-
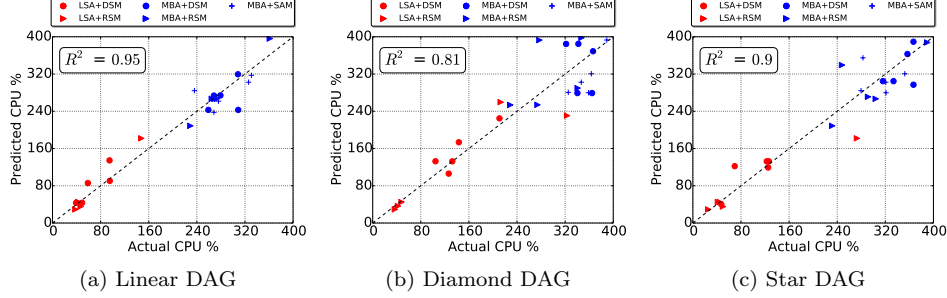
Figure 11: Scatter plot of *Predicted* and *Actual* CPU% per VM for the Micro-DAGs on 5 VMs using the 5 scheduling strategy pairs

mance within exclusive slots and bundles 50 *threads* each on 9 slots and the rest in a partial slot to give a predicted and actual rate close to 280 *tuples/sec*. This highlights the trade-off that SAM makes in favor of a predictable model-driven performance, while sacrificing some of performance benefits relative to RSM.

8.5.2 Prediction and Comparison of CPU and Memory Utilization

Figs. 11 and 12 show the Actual (X axis) and Predicted (Y axis) CPU% and memory% values for the three DAGs. Each scatter plot has a data point for each of the 5 VMs and for every scheduling algorithm pair. It is immediately clear from Figs. 11 that our performance model is able to *accurately predict the CPU% for each VM for these DAGs with a high correlation coefficient $R^2 \geq 0.81$*. This consistently holds for all three DAGs, scheduling algorithms, and for CPU utilization that ranges from $10 - 90\%$.

While for the Linear DAG, the CPU utilization accuracy is high, there are a few cases in the Diamond and Star DAGs where our predictions deviate from the observed for higher values of CPU%. The under-prediction of CPU for Diamond DAG with MBA+SAM is because the VMs with Pi thread bundles receive a slightly higher input rate than expected due to Storm's shuffle grouping that impacts 4 of the 5 slots, and Pi's CPU model is sensitive to even small changes in the rate. For e.g., in Fig. 11b, a VM with a predicted CPU use of 80% for a predicted input rate of 110 *tuples/sec* ends up having an actual CPU usage of 88% as it actually gets 116 *tuples/sec*. This happens for the Star DAG with Pi and Blob threads we well. As mentioned before, enhancing Storm's shuffle grouping to be sensitive to resource allocation for downstream slots will address this skew while improving the performance as well. At the same time, just from a modeling perspective, it is also possible to capture the round-robin routing of Storm's shuffle grouping in the model to improve the predictability.

For Star DAG in Fig. 11c, there is one VM whose predicted CPU% is more than the actual for both MBA+RSM and MBA+SAM. We find that both these VMs have 2 threads of the Parse task, each on a separate slot, that are meant
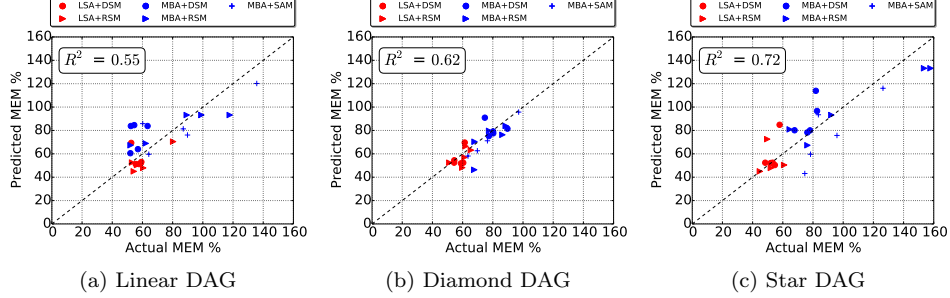
39

Figure 12: Scatter plot of *Predicted* and *Actual* Memory% per VM for the Micro-DAGs on 5 VMs using the 5 scheduling strategy pairs

to support a required input rate of 480 *tuples/sec*. However, a single thread of Parse supports 310 *tuples/sec*. Since these two threads receive less than the peak rate of input, our model proportionally scales down the expected resource usage and estimates it at 47% CPU usage. However, the actual turns out to be 32%, causing an overestimate. As we mentioned, there is a balance between the costs for building fine-grained models and the accuracy of the models, and this prediction error is an outcome of this trade-off that causes us to interpolate.

For the Diamond DAG in Fig. 11b, we see two VMs with expected CPU% of $\approx 100\%$ for MBA+RSM but the observed values that are much lesser. These correspond to two Pi threads that the MBA algorithm expects the pair to be placed on the same slot with 95% combined usage while RSM actually maps them onto different VMs with 10% fewer usage by each for 1 thread.

The prediction of memory utilization shown in Figs. 12, while not as good as the CPU% is still valuable at $R^2 \geq 0.55$. Unlike the CPU usage that spans the entire spectrum from $0 - 400\%$ for each VM, the memory usage has a compact range with a median value of $\approx 60\%$. This indicates that the DAGs are more compute-bound than memory-bound. Due to this low memory%, even small variations in predictions has a large penalty on the correlation coefficient.

We do see a few outlying clusters in these memory scatter plots. In the Linear and Star DAGs, we see that MBA+DSM over-predicts the memory usage. This is because the round-robin mapping of DSM assigns single threads of XML Parse to different slots, each of which receive fewer than their peak supported input rate. As a result, our model proportionally scales down the resources but ends up with an over-estimate.

On the other hand, we also see cases where we marginally under-predict the memory usage for these same DAGs for MBA+SAM. Here, the shuffle grouping that sends a higher rate than expected to some slots with full thread bundles, and consequently a lower to other downstream threads, causing the resource usage to be lower than expected.

We also see broader resource usage trends for specific scheduling approaches that can impact their input rate performance. *We see that plans that use LSA*
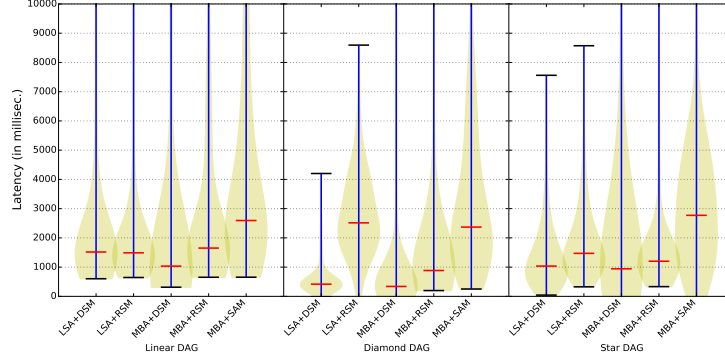
Figure 13: Violin Plot of observed *latency per tuple* for the Micro-DAGs on 5 VMs using the 5 scheduling strategy pairs

consistently under-utilize resources. The CPU% used is particularly bad for LSA, with the 5 VMs for LSA-based plans using an average of just $15-35\%$ CPU each while the MBA-based schedules use an average of $70-90\%$ per VM. This reaffirms our earlier insight that the allocation of the number of data-parallel threads by LSA is inadequate to utilize the resources available in the given VMs. Among DSM and RSM, we do see that RSM clearly has a better CPU% when using LSA though the memory% between DSM and RSM is relatively similar. The latter is because RSM ends up distributing memory intensive threads across multiple slots due to constraints on a slot, which has a pattern similar to DSM's round-robin approach. This shows the benefits of RSM over DSM, as is also seen in the input rates supported.

However, *RSM has a more variable CPU% and memory% utilization across VMs irrespective of the allocation.* For e.g. in Fig. 11a, the Linear DAG has CPU% that ranges from $10-40$ for LSA+RSM and from $55-90$ for MBA+RSM. This is because RSM tries to pack all slots in a VM as long as the cumulative CPU% for the *VM* and the memory% *per slot* is not exhausted. This causes the CPU% of initially mapped VMs to grow quickly due to the best-fit distance measure, while the remaining VMs are packed with more memory-heavy tasks. This causes the skew. The DSM mapping uses a round-robin distribution of threads to slots and hence has is more tightly grouped. While SAM uses a best-fit packing similar to RSM, this is limited to the partial thread bundles, and hence its resource skew across VMs is mitigated.

8.6 Comparison of Latency

Reducing the latency is not an explicit goal for our scheduling algorithms, though ensuring a stable latency is. However, some applications may require a low absolute latency value that is a factor in the schedule generator. So we also report the average latency distribution observed for the different scheduling algorithm pairs for the three micro-DAGs executed on a static set of 5 VMs.

41

The *average latency* of the DAG is the average of time difference between each message consumed at the source tasks and all its causally dependent messages that are generated at the sink tasks. The latency of a message depends on the input rate and resources allocated to the task. It includes the network, queuing and processing time of tuple. The average latency is relevant only for a DAG with a stable latency and resource configuration.

We have used separate spout and sink tasks for logging each input and output tuple with a timestamp, and use this to plot the distribution of the average latency for a DAG. Fig. 13 shows a Violin Plot for the average latency for the three micro-DAGs executed on 5 VMs using both LSA and MBA based allocation with DSM, RSM, MBA mappings, at stable rate. These results are for the same experiment setup as § 8.5.

We can make a few observations based on these plots. We see that the Diamond micro-DAG has a consistently lower latency, followed by the Star DAG and then the Linear DAG. As is evident from the dataflow structure, this is proportional to the number of tasks on the critical path of the DAG, from the source to the sink. This is 4 for Diamond, 5 for Star and 7 for Linear.

The median observed latency values typically increase in the order: MBA+DSM $\leq$ {LSA+DSM, MBA+RSM} $\leq$ LSA+RSM $\leq$ MBA+SAM. However, this has to be tempered by the input rates that these schedules support for the same DAG and resource slots. While MBA+DSM has a low latency, it supports the lowest rate among the three scheduling pairs that use MBA, though all support a higher rate than the LSA-based algorithm pairs. So this is suitable for low latency and average throughput. MBA+RSM has the next best latency given that RSM is network-aware and hence, able to lower the network transfer latency. This is positive given that it is also able to support a high input rate. The LSA+RSM schedule have the second worst latency and also the worst input rate, seen earlier. So this algorithm pair is not a good selection. Separately, we also report that the MBA schedules have a long tail distribution of latency values, indicating that the threads are running at peak utilization that is sensitive to small changes.

9   Related Work

9.1   Scheduling for Storm

The popularity of Storm as a DSPS has led to several publications on streaming DAG scheduling that is customized for Storm. As discussed before, Storm natively supports the default round-robin scheduler and the R-Storm resource-aware scheduler. Both of these only participate in the mapping phase and not in thread or resource allocation. The round-robin scheduler [51]does not take into account the actual VM resources required for a thread instead distributes them uniformly on all available VMs.

In R-storm [48], the user is expected to provide the CPU%, memory% and network bandwidth for each task thread under a stable input message rate, along

with the number of threads for each task. It uses its resource-aware distance function to pack threads to VMs with the goal of achieving a high resource utilization and minimum network latency costs. As we have shown earlier, this linear extrapolation is not effective in many cases. Further, R-Storm does not consider the input rates to the DAG in its model. This means the resource utilization provided by the user is not scaled based on the actual rate that is received at a task thread. Our techniques use both a performance model and interpolation over it to support non-linear behavior and diverse input rates that make it amenable to efficient scheduling even under dynamic conditions.

However, R-Storm is well suited for collections of dataflows that execute on large, shared Strom clusters with hundred's VMs that can be distributed across many racks in the data center. Here, the network latency between VMs vary greatly depending on their placement, and this can impact applications that have a high inter-task communication. Our algorithms do not actively consider the network latency other than scheduling the threads in BFS order, like R-Storm, to increase the possibility of collocating neighboring task threads in the DAG on the same slot. Consequently, our latency values suffer even as we offer predictable performance. Our target is streaming applications launched on a PaaS Storm cluster with tens of VMs that have network proximity, and for this scenario, the absence of network consideration is acceptable. That said, including network distance is a relatively simple extension to our model.

Others have considered *dynamic scheduling* for Apache Storm as well, where the scheduler adapts to changes in input rates at runtime. Latency is often the objective function they try to minimize while also limiting the resource usage. [4] proposes an offline and an online scheduler which aim at minimizing the inter-node traffic by packing threads in decreasing order of communication cost into the same VM, while taking CPU capacity as constraint based on the resource usage at runtime. The goal here is on the mapping phase as well with the number of threads and slots for the tasks assumed to be known *a priori*. The offline algorithm just examines the topological structure and does not take message input rate or resource usage into consideration for scheduling. It just place the threads of adjacent tasks on same slot and then slots are assigned to worker nodes in round robin fashion. The online algorithm monitors the communication patterns and resource usage at run time to decide the mapping. It tries to reduce the inter-node and inter-slot traffic among the threads. The online algorithm have two phases, In the first phase threads are partitioned among the workers assigned to DAG, minimizing the traffic among threads of distinct workers and balancing the CPU on each worker. In the second phase these workers are assigned to available slots in the cluster, minimizing the inter-node traffic. Both these algorithms uses tuning parameters that controls the balancing of threads assigned per worker. These algorithms also have the effect of reducing intra-VM communication traffic besides inter-VM messaging.

T-Storm [71] also takes a similar mapping approach, but uses the summation of incoming and outgoing traffic rates for the tasks in descending order to decide the packing of threads to a VM. It further seeks to limit the messaging within a VM by just running one worker on each VM that is responsible for all

threads on the VM. The algorithm monitors the load at run time and assigns the thread to available slot with minimum incremental traffic load. The number of threads for each task is user defined and their distribution among worker nodes is controlled by some parameter (consolidation factor), which is obtained emperically. Also, algorithm does not gurantee that communicating threads will be mapped to the same node as ordering is done based on total traffic and not on traffic between threads.

Both these schedulers [4, 71] use only CPU% as the packing criteria at runtime, and this can cause memory overflow for memory intensive tasks or when the message queue size grows. Their online algorithms also require active monitoring of the various tasks and slots at runtime, which can be invasive and entail management overheads. At the same time, despite their claim of adapting to runtime conditions, neither scheduler actually acquires new VM resources or relinquishes them, and instead just redistributes the existing threads on the captive set of VMs for load balancing and reducing the active slots. Thus, the input-rate capacity of the dataflow is bounded or the acquired captive resources can be under-utilized. Further, the use of a single worker per VM in T-Storm can be a bottleneck for queuing and routing when the total input and output rate of threads on that VM are high. While we do not directly address dynamic scheduling, our algorithms can potentially be rerun when the input rate changes to rebalance the schedule, without the need for fine-grained monitoring. This can include acquiring and releasing VMs as well since we offer both allocation and mapping algorithms. We consider both memory and CPU usage in our model-based approach. A predictable schedule behavior is a stated goal for us rather than reducing absolute latency through reduced communication.

The *P-Scheduler* [21] uses the ratio of total CPU usage from all threads to the VM's CPU threshold to find the number of VMs required for the DAG at runtime. The goal here is to dynamically determine the number of VMs required at runtime based on CPU usage and then map the threads to these VMs such that the traffic among VMs is minimized. The mapping hierarchically partitions the DAG, with edge-weights representing tuple transfer rate. It first maps threads to VMs and then re-partitions threads within the VM to slots. This reduces the communication costs but the partitioning can cause unbalanced CPU usage, and the memory usage is not considered at all. While the algorithm does VM allocation, it does not consider thread allocation that can cause VMs to be under-utilized. It also requires a centralized global monitoring of the data rates between threads and CPU% to perform the partitioning and allocation.As mentioned before, our scheduling offers both VM and thread allocation in addition to mapping, consider input rate, CPU% and memory% for the decisions, and our model does not rely on active runtime monitoring.

There have been few works on resource and thread allocation for Storm. The *DRS* [25] is one such scheduler that models the DAG as open queuing network to find the expected latency of a task for a given number of threads. Its goal is to limit the expected task latency to within a user-defined threshold at runtime, while minimizing the total number of threads required and hence the resources. It monitors the tuple arrival and service rate at every task to find

the expected latency using *Erlang formula* [61]. The approach first finds the expected number of threads required for the task so that latency bound is met. This is done by increasing a thread for the task which gives maximum latency improvement obtained by Erlang formula discussed in paper, but it requires that an upper bound on total number of threads to be set by user. Also paper assumes that only a fixed number of threads can run on a VM, independent of thread type. The number of VMs are identified using total number of threads and number of threads that can run on a VM, already fixed by user. Mapping is done by default scheduler only. DRS uses an analytical approach like us, but based on queuing theory rather than empirical models. They apply it for runtime application but do not consider mapping of the threads to VM slots. We consider both allocation and mapping, but do not apply them to a dynamic scenario yet. Neither approaches require intensive runtime monitoring of resources and rates. Like us, they consider CPU resources and threads for allocation and not network, but unlike us, they do not consider memory%. Their experiments show a mismatch between expected and observed performance from failing to include network delays in their model while our experiments do not highlight network communication to be a key contributor, partly because of the modest rates supported on the DAGs and resources we consider. They also bound the total number of CPU slots and the number of threads per VM, which we relax.

9.2 Scheduling for DSPS

While our scheduling algorithms were designed and evaluated in the context of Storm, it is generalizable to other DSPS as well. There has been a body of work on scheduling for DSPS, both static and adaptive scheduling on Cloud VMs, besides those related to Storm.

Borealis [1] an extnesion to Aurora [10] provides parallel processing of streams. It uses local and neighbor load information for balancing load across the cluster by moving operators. They also differ from cloud based DSPS as they assume that only fixed amount resources are available beforehand. Some extensions to Borealis like [49, 70], does not use intra operator level parallelism and considers only dynamic mapping of tasks for load balancing. TelegraphCQ [12] uses adaptive routing using special operators like Eddy and Juggle to optimize query plans. These special operators decides how to route data to different operators, reorders input tuples based on their content. It also dynamically decides the optimal stream partitioning for parallel processing. These systems allocate queries to seperate nodes for scaling with the number of queries and are not designed to run on cloud.

COLA [37] for System S, scalable distributed stream processing system aims at finding best operator fusion (multiple operators within same process) possible for reducing inter process stream traffic.The approach first uses list scheduling (longest processing time) to get operators schedule then it checks for VM capacity(only CPU) if schedule is unfeasible, uses graph partitioning to split processing element to other VMs.Thus COLA also does not take memory intensive

operators in to account. Infosphere streams [7] uses component-based programming model. It helps in composing and reconfiguring individual components to create different applications. The scheduler component [67] finds the best partitioning of data-flow graph and distributes it across a set of physical nodes. It uses the computational profiles of the operators, the loads on the nodes for making its scheduling decisions. Apache S4 [46] follows the actor model and allocation is decided manually based on distinct keys in the input stream. The messages are distributed across the VMs based on hash function on all keyed attribute in input messages. S4 schedules parallel instances of operators but does not manage their parallelism and state. Since it does not support dynamic resource management thus unable to handle varying load. IBM System S [3] run jobs in the form of data-flow graphs. It supports intra-query parallelism but management is manual. It also supports dynamic application composition and stream discovery, where multiple applications can directly interact. This support for sharing of streams across applications is done by annotating the messages with already declared types in global type system. This enables sharing of applications written by different developers through streams.

Esc [53] which process streaming data as key-value pairs. Hash functions are used to balance load by dynamically mapping the keys to the machines and function itself can be updated at run time. Hash function can also use the cpu,memory load based on the VM statistics for message distribution. Dynamic updation of the DAG based on the custom rules from user is also supported for load balance. A processing element in a DAG can have multiple operators and can be created at run time as per need. There can be many workers for a processing element. Since it dynamically adjusts the required computational resources based on the current load of the system it is good fit for use cases with varying load, with deployment on cloud.

[39] have used variant called dynamic dataflows that adapts to changing performance of cloud resources by using alternate processing elements.The logic uses variable sized bin packing for allocating processing element over the VMs on cloud. Dynamic rates are managed by allocating resources for alternate processing elements thus making tradeoff between cost and QoS on cloud.

In [28] have proposed elastic auto-parallelization for balancing the dynamic load in case of SPL applications. The scaling is based on a control algorithm that monitors the congestion and throughput at runtime to adjust data parallelism. Each ope rator maintains a local state for every stream partition. An incremental migration protocol is proposed for maintaining states while scaling, minimizing the amount of state transfer between hosts.

StreamCloud [30] modifies the parallelism level by splitting queries into sub queries minimizing the distribution overhead of parallel processing, each of which have utmost one stateful operator that stores its state in a *tuple-bucket*, where the key for a state is a hash on a tuple. At the boundary between sub-queries, tuples are hashed and routed to specific stateful task instances that hold tuple-bucket with their hash key. This ensures consistent stateful operations with data-parallelism. It uses special operators called Load Balancers placed over outgoing edge of each instance of subcluster, LB does Bucket In-

stance Mapping to map buckets with instances of downstream clusters.

ChronosStream [69] hash-partitions computation states into collection of fine-grained slices and then distributes them to muliple nodes for scaling. Each slice is a computationally independent unit associated with a subset of input streams and and can be transparently relocated without affecting the consistency. The elasticity is achieved by migrating the workload to new nodes using a lightweight transactional migration protocol based on states.

ElasticStream [34] considers a hybrid model for processing streams as it is impossible to process them on local stream computing environment due to finite resources. The goal is to minimize the charges for using the cloud while satisfying the SLA, as a trade-off between the applications latency and charges uisng linear programming. The approach dynamically adjusts the resources required with dynamic rates in place of over-provisioning with fixed amount of resources. The implementation done on System S is able to assign or remove computational resources dynamically.

Twitter Heron [38] does user defined thread allocation and mapping by Aurora scheduler. In the paper [42] proposed an analytical model for resource allocation and dynamic mapping to meet latency requirement while maximizing throughput, for processing real time streams on hadoop. Stela [72] uses effective throughput percentage (ETP) as the metric to decide the task to be scaled when user requests scaling in/out with given number of machines. The number of threads required for the tasks and their mapping to slots is not being discussed in the paper.

9.3   Parallel Scheduling

Our model-based approach is similar to scheduling strategies employed in parallel job and thread scheduling for HPC applications.

The Performance Modeling frameworks [5,59,60] for large HPC systems predicts application performance from a function of system profiles (e.g., memory performance, communication performance). These profiles can be analysed to improve the application performance by understanding the tuning parameters. Also [5] proposes methods to reduce the time required for performance modelling, like combining the results of several compilation, execution, performance data analysis cycles into a application signature, so that these steps need not to be repeated each time a new performance question is asked.

Warwick Performance Prediction (WARPP) [31] simulator is used to construct application performance models for complex parallel scientific codes executing on thousands of processing cores. It utilises coarse-grained compute and network models to enable the accurate assessment of parallel application behaviour at large scale. The simulator exposes six types of discrete events ranging from compute to I/O read,write to generate events representing the behaviour of a parallel application. [26] models the aplication performance for future architectures with several millions or billions of cores. It considers algebraic multigrid (AMG), a popular and highly efficient iterative solver to discuss the model-based predictions. It uses local computation and communi-

cation models as baseline for predicting the performance and its scalability on future machines. The paper [32] proposes simple analytical model to predict the execution time of massively parallel programs on GPU architecture. This is done by estimating the number of parallel memory requests by considering the number of running threads and memory bandwidth. The aplication execution time in GPU is dominated by the latency of memory instructions. Thus by finding the number of memory requests that can be executed concurrently (memory warp parallelism) the effective costs of memory requests is estimated.

[33] proposes Planning systems and compares them to Queuing systems for resource managament in HPC. Features like advance resource reservation, request diffusing can not be achieved using queuing because it considers only present resource usgae. Planning systems like CCS, Maui Scheduler does resource planning for present and future by assigning start time to all requests and using run time estimates for each job.

Recent works like [20] uses statistical approach to predict application execution time using emperical analysis of execution time for small input sizes. The paper uses a collection of well known kernel benchmarks for modelling the execution time of each phase of an application. The approach collects profiles obtained by few short application runs to match phases to kernels and uses it for predicting the execution times accurately.

Our model-based mapping of a bundle of threads also has some similarities with *co-scheduling* [47] or *gang scheduling* [23] of threads in concurrent and parallel systems. In the former, a set of co-dependent processes that are part of a working set are scheduled simultaneously by the Operating System (OS) on multi-processors to avoid process thrashing. In gang scheduling, a set of interacting threads are scheduled to concurrently execute on different processors and coordinate through busy-waits. The intution is to assign the threads to dedicated processors so that all dependent threads progress together without blocking. Our allocation and mapping based on performance models tries to identify and leverage the benefits of co-scheduling coarse-grained thread bundles from the same task onto one CPU, with deterministic performance given by the models, and by separating out thread bundles from different tasks onto different slots to execute concurrently without resource contention.

We also encounter issues seen in gang scheduling that may cause processor over or under-allocation if sizes of the gangs do not match the number processors, which is similar to the partial thread bundles mapped to the same slot in our case, causing interference but reusing partial slots [22]. At the same time, we perform the thread to slot mapping once in the streaming dataflow environment, and do not have to remap unless the input rate changes. Hence, we do not deal with recurrent or fine-grained scheduling issues such as constantly having to schedule threads since the number of threads are much more than the CPU cores, paired gang scheduling for threads with interleaved blocking I/O and compute [66], or admission control due to inadequate resources [6].

10   Conclusion

Based on these results, we see that LSA+RSM consistently allocates more resources than MBA+SAM, often twice as many slots due to its linear extrapolation of rate and resources. However, it still misses the planned input rate supported by $30 - 40\%$ in several cases due to unbalanced mapping by RSM where the rate does not scale as it expects. We see a $5 - 10\%$ drop for MBA+SAM due to the shuffle grouping that uniformly routes tuples to threads. Also, RSM often requires additional resources than ones allocated by LSA due to fragmentation during bin-packing, though this tends to be marginal. SAM has less fragmentation due to packing full bundles to exclusive slots. These hold for all DAGs, both micro and application, small and large.

The model-based prediction of input rates is much more accurate than the planned prediction, correlating with the actual rate with and $R^2 \geq 0.71$. The few outliers we see are due to the model expecting a different routing compared to Storm's shuffle grouping, and due to the interpolation of rates based on the granularity of the performance models. MBA is consistently is better than LSA in the input rate supported for the same quanta of resources, through MBA+DSM shows the least improvement. MBA+RSM is often better than MBA+SAM in actual rate though MBA+SAM gives a predictable observed rate.

Our performance model is able to predict the resource utilization for individual VMs with high accuracy, having R^2 value ≥ 0.81 for CPU% and ≥ 0.55 for MEM%, independent of the allocation and mapping technique used. The few prediction errors we see are due to threads receiving fewer than the peak rate for processing, where our model proportionally scales down the estimated resource usage relative to a single-thread usage at the peak rate. The low memory% also causes the error to be sensitive to even small skews in the prediction, giving a lower correlation coefficient value.

MBA consistently has a higher resource utilization than LSA, that is also reflected in the better input rate performance. While the resource usage across VMs for schedules based on MBA are close together, RSM shows a greater variation of its CPU% and memory% across VMs.

11   Future Work

The current slot aware mapping does not consider load aware shuffle groping, we can leverage it to have more accuracy for predicting supported input rate and resource requirement. Also Dynamic resource allocation and mapping for the given distribution of input rate or monitored input rate at run time is part of our future work.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[3] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, DMSSP, pages 27–37. ACM, 2006.

[4] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218. ACM, 2013.

[5] David H Bailey and Allan Snavely. Performance modeling: Understanding the past and predicting the future. In *European Conference on Parallel Processing*, pages 185–195, 2005.

[6] Anat Batat and Dror G Feitelson. Gang scheduling with memory considerations. In *IPDPS*, pages 109–114. Citeseer, 2000.

[7] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *ACM SIGMOD*, 2010.

[8] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

[9] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. Elastic stateful stream processing in storm. In *High Performance Computing & Simulation (HPCS)*, pages 583–590. IEEE, 2016.

[10] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Very large data bases-Volume 29*, pages 838–849. VLDB Endowment, 2003.

[11] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668. ACM, 2003.

[12] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *International Conference on Management of Data*, SIGMOD, pages 668–668. ACM, 2003.

[13] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime data processing at facebook. In *ICMD*, SIGMOD '16, pages 1087–1098. ACM, 2016.

[14] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[15] David De Roure, Carole Goble, and Robert Stevens. The design and realisation of the virtual research environment for social sharing of workflows. *Future Generation Computer Systems*, 25(5):561–567, 2009.

[16] David De Roure, Carole Goble, and Robert Stevens. The design and realisation of the virtual research environment for social sharing of workflows. *Future Generation Computer Systems*, 25(5):561–567, 2009.

[17] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014.

[18] Brian Donovan and Daniel B. Work. Using coarse gps data to quantify city-scale transportation system resilience to extreme events. In *Transportation Research Board 94th Annual Meeting*, 2014.

[19] Raphael Eidenbenz and Thomas Locher. Task allocation for distributed stream processing. 2016.

[20] Rodrigo Escobar and Rajendra V. Boppana. Performance prediction of parallel applications based on small-scale executions. In *High Performance Computing (HIPC)*, pages 208–216. IEEE, 2016.

[21] Leila Eskandari, Zhiyi Huang, and David Eyers. P-scheduler: Adaptive hierarchical scheduling in apache storm. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '16, pages 26:1–26:10. ACM, 2016.

[22] Dror G Feitelson and Larry Rudolph. Wasted resources in gang scheduling. In *Information Technology, 1990.'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No. 90TH0326-9)*, pages 127–136. IEEE, 1990.

[23] Dror G Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[24] Rosa Filgueira, Amrey Krause, Malcolm Atkinson, Iraklis Klampanos, Alessandro Spinuso, and Susana Sanchez-Exposito. dispel4py: An agile framework for data-intensive escience. In *e-Science (e-Science)*, pages 454–464. IEEE, 2015.

[25] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. Drs: dynamic resource scheduling for real-time analytics over fast streams. In *Distributed Computing Systems (ICDCS), 2015*, pages 411–420. IEEE, 2015.

[26] Hormozd Gahvari, Allison H. Baker, Martin Schulz, Ulrike Meier Yang, Kirk E. Jordan, and William Gropp. Modeling the performance of an algebraic multigrid cycle on hpc platforms. In *International Conference on Supercomputing*, ICS, pages 172–181, 2011.

[27] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: The system s declarative stream processing engine. In *ICMD*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[28] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *TPDS*, 25(6):1447–1463, 2014.

[29] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[30] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE TPDS*, 2012.

[31] Simon D Hammond, Gihan R Mudalige, JA Smith, Stephen A Jarvis, JA Herdman, and A Vadgama. Warpp: a toolkit for simulating high-performance parallel scientific codes. In *International Conference on Simulation Tools and Techniques*, page 19. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[32] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[33] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in hpc resource management systems: Queuing vs. planning. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–20, 2003.

[34] Atsushi Ishii and Toyotaro Suzumura. Elastic stream computing with clouds. In *International Conference on Cloud Computing*, CLOUD, pages 195–202. IEEE Computer Society, 2011.

[35] Shantenu Jha, Daniel S. Katz Andre Luckow, Omer Rana, Yogesh Simmhan, and Neil Chue Hong. Introducing distributed dynamic data-intensive (d3) science: Understanding applications and infrastructure. *Concurrency and Computation: Practice and Experience*, 2016.

[36] Thomas P. Dence Joseph B. Dence. A rapidly converging recursive approach to pi. *The Mathematics Teacher*, 86(2):121–124, 1993.

[37] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 16:1–16:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[38] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[39] Alok Kumbhare, Yogesh Simmhan, and Viktor K Prasanna. Exploiting application dynamism and cloud elasticity for continuous dataflows. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 57. ACM, 2013.

[40] Alok Gautam Kumbhare, Yogesh Simmhan, and Viktor K Prasanna. Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2014*, pages 344–353. IEEE, 2014.

[41] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

[42] Boduo Li, Yanlei Diao, and Prashant Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015.

[43] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks. In *IEEE/ACM UCC, 2014*, 2014.

[44] C. B. Medeiros, J. Perez-Alcazar, L. Digiampietri, G. Z. Pastorello, Jr., A. Santanche, R. S. Torres, E. Madeira, and E. Bacarin. Woodss and the web: Annotating and reusing scientific workflows. *SIGMOD Rec.*, 34(3):18–23, 2005.

[45] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *IEEE ICDMW*, 2010.

[46] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[47] John K Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS*, volume 82, pages 22–30, 1982.

[48] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015.

[49] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *International Conference on Data Engineering*, ICDE, pages 49–53, 2006.

[50] Marek Rychly et al. Scheduling decisions in stream processing on heterogeneous clusters. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on*, pages 614–619. IEEE, 2014.

[51] Marek Rychly et al. Scheduling decisions in stream processing on heterogeneous clusters. In *Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 614–619. IEEE, 2014.

[52] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Computing (CLOUD)*, pages 348–355. IEEE, 2011.

[53] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 348–355. IEEE, 2011.

[54] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Claudio T. Silva. Querying and re-using workflows with vstrails. In *SIGMOD*, pages 1251–1254. ACM, 2008.

[55] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *IPDPS-2009*, pages 1–12. IEEE, 2009.

[56] Anshu Shukla and Yogesh Simmhan. Benchmarking distributed stream processing platforms for iot applications. *Technology Conference on Performance Evaluation and Benchmarking (TPCTC), VLDB-2016*, 2016.

[57] Yogesh Simmhan, Saima Aman, Alok Kumbhare, Rongyang Liu, Sam Stevens, Qunzhi Zhou, and Viktor Prasanna. Cloud-based software platform for data-driven smart grid management. *IEEE/AIP Computing in Science and Engineering*, July/August, 2013.

[58] Yogesh Simmhan, Anshu Shukla, and Arun Verma. Benchmarking fast-data platforms for the aadhaar biometric database. In *Big Data Benchmarking:7th International Workshop, WBDB*, pages 21–39, 2015.

[59] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Supercomputing*, pages 21–21. IEEE, 2002.

[60] Allan Snavely, Xiaofeng Gao, Cynthia Lee, Laura Carrington, Nicole Wolter, Jesus Labarta, Judit Gimenez, and Philip Jones. Performance modeling of hpc applications. *Advances in Parallel Computing*, 13:777–784, 2004.

[61] H.C. Tijms. *Stochastic Modeling and Analysis: A Computational Approach*. Wiley series in probability and mathematical statistics: Applied probability and statistics. John Wiley & Sons, 1986.

[62] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *ACM SIGMOD*, pages 147–156. ACM, 2014.

[63] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[64] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA)*, pages 488–499. IEEE, 2014.

[65] Josef Weidendorfer and Jens Breitbart. Detailed characterization of hpc applications for co-scheduling. In *COSH@ HiPEAC*, pages 19–24, 2016.

[66] Yair Wiseman and Dror G. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):581–592, 2003.

[67] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *USENIX*, Middleware, pages 306–325. Springer-Verlag New York, Inc., 2008.

[68] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? In *ACM ICDES*, DEBS, 2012.

[69] Yingjun Wu and Kian-Lee Tan. Chronostream: Elastic stateful stream computation in the cloud. In *Data Engineering (ICDE)*, 2015.

[70] Ying Xing, Stan Zdonik, and J-H Hwang. Dynamic load distribution in the borealis stream processor. In *International Conference on Data Engineering (ICDE)*, pages 791–802. IEEE, 2005.

[71] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, pages 535–544, 2014.

[72] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *IEEE International Conference on Cloud Engineering (IC2E)*, 2016.

[73] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.