Scheduling for Reduced Tail Task Latencies in Highly Utilized Datacenters

Smita Vijayakumar
University of Cambridge
Cambridge, UK
sv440@cam.ac.uk

Anil Madhavapeddy
University of Cambridge
Cambridge, UK
avsm2@cam.ac.uk

Evangelia Kalyvianaki
University of Cambridge
Cambridge, UK
ek264@cam.ac.uk

ABSTRACT

Modern datacenters run diverse workloads that increasingly comprise data-parallel computational jobs. There has been a steady rise in their demand leading to high-volume traffic. To meet these demands, datacenter providers operate their clusters at levels of high utilization. We show that under such conditions, existing schedulers impose large wait times on tail tasks, leading to long job completion time. We propose a new decentralized scheduler, Murmuration, that reduces the total wait time of tasks. It employs multiple communicating schedulers to schedule tasks of jobs such that their start times are as close together as possible, ensuring small tail task completion time and better average job completion time.

Our evaluation of Murmuration using publicly available workloads on a real-world cluster shows 15% — 25% faster job completion time than that of the default Kubernetes scheduler for different arrival characteristics. We show that Murmuration scales to incoming workloads by scheduling more than a million tasks in a matter of minutes. We further enhance the design of Murmuration by incorporate queue re-ordering techniques to order the scheduling and execution of jobs and tasks. Simulations evaluated on two industry workloads show that with queue re-ordering, Murmuration outperforms other schedulers with a 100× better median job completion time than that of current schedulers.

CCS CONCEPTS

• Applied computing → Data centers; **• Computer systems organization → Cloud computing**; **Distributed architectures**.

KEYWORDS

Scheduling, tail latency, highly utilized, datacenters, bursty, heavy-load

ACM Reference Format:
Smita Vijayakumar, Anil Madhavapeddy, and Evangelia Kalyvianaki. 2024. Scheduling for Reduced Tail Task Latencies in Highly Utilized Datacenters. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3698038.3698522

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SoCC '24, November 20–22, 2024, Redmond, WA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698522

1 INTRODUCTION

Datacenters deliver cost-effective infrastructure to serve the needs of user applications. These applications comprise one or more *jobs* that, in turn, comprise one or more tasks. Schedulers allocate resources to tasks, aiming to achieve one or more goals like cluster utilization [4, 29, 73] and user-perceived service quality [17]. As the number of applications hosted in datacenters rises [5, 8, 62, 66], ensuring the availability of sufficient resources while being profitable becomes crucial [53, 73]. To this end, service providers operate datacenters in a region of high load [9, 40].

However, there are times when available resources lag behind increased user demands, such as short spikes in requests that cause temporary overload. It is hard to plan for such spikes since the mid- to long-term capacity of datacenters is predicted based on models that rely on past demand and supply patterns [6, 11, 56]. Often such predictions are ridden with inaccuracies leading to insufficient capacities [56]. Moreover, future demand patterns do not always mimic past trends, as with unforeseen global events like pandemics. Datacenters are also prone to pockets of high load on a small number of machines due to unexpected bursts in requests, network partitions, firmware bugs, over-subscription of resources and incorrect load balancing [55, 80]. Finally, not every datacenter is a hyperscaler, which are operated with spare idle resources [11]. Overload is a possibility in small- to medium-sized datacenters where resources are limited and temporary bursts can saturate clusters.

When faced with such increased demand on available resources providers may choose to temporarily drop traffic, eliminate low-priority batch load or enter degraded modes of operation [11, 30, 52, 56]. However, these measures directly

affect the quality of service perceived by users. Alternately, the approach we follow in this paper is to allow all outstanding requests to run to completion when queuing delays are tolerable. We demonstrate that in such busy clusters these requests take *longer* than required to complete using current scheduling approaches (§2). This is due to large variability in completion time of tasks that comprise these jobs.

Workloads increasingly comprise data processing frameworks that have thousands of tasks per job (*fanout*) [3, 16, 18, 67, 69]. It is well-studied that as fanout increases jobs become vulnerable to large completion time due to variability in task execution time [2, 4, 17, 78]. The inherently bursty nature of incoming requests leads to a build up of task queues on resources [17, 61] especially as cluster utilization increases [23, 37, 51, 61], leading to increased variation in execution time of tasks. In this work, we demonstrate that *in the context of highly utilized datacenters* schedulers aggravate variability in task completion time in high fanout jobs, leading to a further increase in completion time of jobs.

While cluster schedulers have been extensively studied, to the best of our knowledge, we are the first to quantitatively analyze job tail latency in the context of highly utilized clusters and bursty workloads, which presents significant design and implementation challenges. In particular, we make the following contributions. First, we show that current scheduling approaches contribute towards long job completion time due to long completion time of tail tasks, i.e., the task which finish the last among all tasks of a job. We present an analysis of tail task latency in highly utilized clusters under both centralized and distributed scheduling paradigms. We use Kubernetes which is a state-of-the-art container orchestration platform with a centralized scheduler [49], and Sparrow which is a distributed scheduler designed to schedule short tasks fast [59]. We show that under high load, large job completion time in Kubernetes is attributed to long *tail task* completion time, while in Sparrow it is attributed to long *average task* completion time (§2).

Second, based on our observations, we propose a new scheduling approach, Murmuration, that reduces tail task latency in highly utilized clusters by employing job-aware scheduling (§3). Our key insight is to schedule for *temporal proximity in the execution start time of tasks of a job.* Murmuration is a decentralized scheduler, defined as one in which multiple scheduler instances communicate placement information with each other to take independent scheduling decisions. This ensures scalability in handling incoming job requests (§4). In Murmuration, schedulers communicate placement information to build local cluster state. They use this information to place tasks of a job on nodes such that these tasks start execution at around the same time and as soon as possible, subject to the current state of the system. This leads to smaller tail task latency as compared to existing schedulers.

Third, we evaluate our implementation (§5) of Murmuration on Kubernetes (§6) using publicly available workloads on 50 node clusters. Murmuration shows 15 — 25% smaller median job completion time over the default Kubernetes scheduler, while exhibiting a small scheduling time for tasks. We also simulate large clusters with thousands of machines where Murmuration, implemented with queue re-ordering, shows a median job completion time that is 100× better than that of current schedulers when evaluated on two industry workloads. We contextualize our findings to reflect on the implications of our work to job scheduling (§7).

2 MOTIVATION

We first examine how current scheduling approaches affect job completion time when most resources are busy in a datacenter. Under such conditions, when a burst of jobs arrives, it is a challenge for schedulers to find free resources quickly enough for jobs to finish fast. We consider jobs that consist of independent tasks that can be executed in parallel [13], like data-parallel computational frameworks, machine learning application [3, 16, 18, 67–69] and HPC workloads which are increasingly moving to the cloud [7, 10]. We assume that tasks of a job are all ready for scheduling at around the same time, i.e., at job creation time. The design also handles tasks re-created following failure scenarios (§4.2). The focus of this work is on highly loaded clusters that regularly face sharp bursts of incoming jobs but have enough resources to eventually execute all jobs. We do not target latency-critical or long-running applications, rather workloads which are tolerant of queuing delays.

We first model the major steps that tasks go through between submission and completion (§2.1). We describe the workloads used in this work (§2.2) followed by a study of modern schedulers in relation to this model. For centralized schedulers (§2.3), we identify that the time spent at the scheduler can significantly increase job completion time. We show that the problem of long scheduling time is tackled by distributed schedulers (§2.4) which schedule tasks quickly, but their limited view of cluster resources makes their placements sub-optimal under high load as compared to centralized schedulers [29, 79].

2.1 End-to-End Task Timeline

We define *task completion time (TCT)* as the time interval between when a task is ready for scheduling and its completion. A job completes when the last of its tasks finishes, so a *job's completion time (JCT)* is the maximum *TCT* among all its tasks [29]. *Tail tasks* are tasks that finish the last among all tasks of a job. Since *JCT* is the maximum *TCT* among all

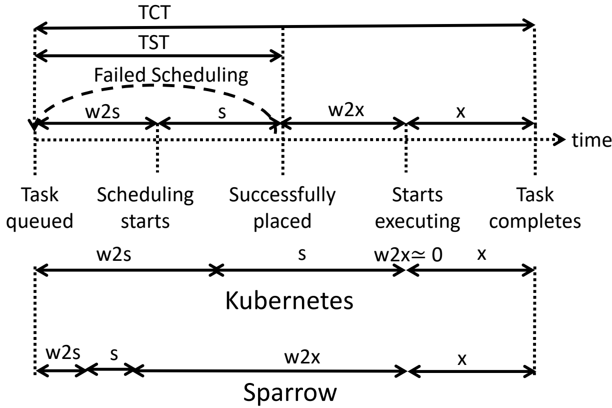


Figure 1: Task Lifecycle.

of its tasks, JCT is defined by the tail TCT, that is, for task t and job j,

$$JCT_j = \max_{t \in j}\{TCT_t\} \quad (1)$$

Figure 1 shows how TCT can be roughly divided into four non-overlapping time periods: *wait time to be scheduled (w2s), scheduling time (s), wait time to execute (w2x) and execution time (x)*. We use these timings, explained below, to study and compare different schedulers.

When a task first arrives for scheduling, it is placed in a scheduler's queue. *w2s* refers to the time a task spends in this queue waiting to be scheduled. Scheduling time (s) is the time a scheduler spends trying to find a suitable node for the task to run on. If a node is found, the task is sent to the designated node for execution. If no such node is found the task is put back into the queue to be selected for scheduling at a later time. *w2s* and *s* can repeat multiple times until a node is found for a task. We use the term *task scheduler time (TST)* to refer to the *total* time a task spends at the scheduler until a suitable node is found and includes all the times a task spends in *w2s* and *s*.

After a task is assigned to a node for execution, it could spend time waiting to be executed at the designated node ($w2x$). This is the time a task spends at the *worker-side queues*, if these exist [8, 59, 63]. However, when a scheduler does not support worker-side queues [29, 72], then *w2s* is practically zero as these schedulers find nodes with enough free resources for immediate task execution. Execution time x is the total time spent executing a task. So, $TCT = TST + w2x + x$. Table 1 summarizes the notations used to describe a task's lifecycle.

We examine how the various times identified during task processing affect JCT in different scheduling approaches.

Table 1: Notations used to describe task lifecycle.

Notation	Description
w2s	wait time to be scheduled
s	scheduling time
w2x	wait time to execute
x	execution time
TST	Task Scheduler Time
TCT	Task Completion Time

Broadly, schedulers can be categorized as centralized and distributed. Centralized schedulers [29, 41, 64, 72, 74] have the entire cluster's information available during scheduling and can make globally-informed high quality task placements. Distributed schedulers [46, 59] are scalable and cope well with high input request rates, but they lack a global cluster view for placing tasks well. We use the Kubernetes and Sparrow schedulers as baselines to examine how they perform with regard to TST and TCT in a highly loaded cluster. For each of these schedulers we examine the characteristics of tail task completion times (tail TCTs). We evaluate Kubernetes using a cluster hosted on CloudLab [25], and Sparrow using Eagle's simulator [19].

2.2 Workload Description

The Murmuration scheduler is designed for modern industry workloads that typically have more than one task per job [8, 69, 74]. In this work, we use two different industry workloads - Yahoo traces [13] which is our primary workload and Cloudera traces [12], our secondary workload (§6). Both workloads describe the characteristics of data-parallel jobs in large datacenters [12, 13]. They describe incoming jobs in terms of their arrival times, fanout, estimated job (task) running time, and the actual running times of the tasks. We summarize the characteristics of these jobs in Table 2. Both Kubernetes and Sparrow are evaluated in this section using the Yahoo workload.

Table 2: Workload characteristics.

Workload	Fanout		Job Inter-arrival(s)		Task Duration(s)	
	50th	99th	50th	99th	50th	99th
Yahoo	15	636	8	8	16	2410
Cloudera	126	3650	12	13	48	2515

2.3 Centralized Scheduling

We use Kubernetes as a reference centralized scheduler and its design does not support worker-side queues. Figure 1 shows the lifecycle of a task as scheduled by the default Kubernetes scheduler. First, tasks of new jobs are put into

the scheduler's *ready* queue where all tasks that are ready to be scheduled wait ($w2s$). The scheduler de-queues tasks based on the earliest creation timestamp and for each task it aims to find a node that fulfills the task's resource and affinity requirements.

Kubernetes maintains a cache with a list of nodes available in the cluster. When scheduling a task the scheduler filters these nodes to find feasible nodes that have enough free resources *currently* to meet the task's request. When such feasible nodes are found, it assigns a score to each of them based on configured scoring rules such as availability of the task's image and utilization of resources at the node. It assigns the task to the node with the highest score. When multiple nodes have the same score, it chooses one among them at random. The above operations constitute the scheduling of the task (s). The scheduler then sends the task to the node, where the task is ready to run as soon as it arrives since resources will be available (i.e. $w2x$ is negligible). The task's code and image are downloaded and it is executed at the node (x).

There may be cases when no feasible node is found for a task when the cluster is busy and resources requested by the task are not immediately available. In such cases, the scheduler keeps the task in a *retry* queue where it waits for some time ($w2s$) before being re-admitted into the ready queue for scheduling (s). This retry mechanism prevents head-of-line blocking by giving smaller tasks the opportunity to be scheduled with currently available resources.

However, scheduling retries become the common case in highly utilized clusters as tasks go through multiple iterations of queuing and scheduling attempts before being successfully scheduled (TST). For purposes of evaluation, we quantitatively define a highly utilized Kubernetes cluster as one which is actively executing the maximum number of pods it can support. This is derived from the fact that Kubernetes restricts the maximum number of simultaneously active pods on a node to 110, irrespective of its compute cores [48].

Once a Kubernetes cluster is highly utilized, it is challenging for the scheduler to find placement for tasks in the ready queue. Tasks fail to get scheduled and get moved to the retry queue, increasing their TST. The more utilized the cluster, the larger the tail TST. Eventually, as the cluster load reduces, pending tasks get allocated and the waiting time $w2s$ for newly arriving tasks reduces.

2.3.1 **Testbed Setup.** Our Kubernetes cluster is hosted on a CloudLab setup that comprises 50 nodes. The setup is a heterogeneous mix of bare metal machines with 8 cores and 64GB RAM (`m510`), 16 cores and 128GB RAM (`c6525-25g`) and 32 cores and 128GB RAM (`d6515`). All machines run Ubuntu 20.04 and Kubernetes 1.23.6. Unless specified, the cluster has a single master node running all control plane services. We run $5,000 - 10,000$ jobs starting from a randomly selected job in the Yahoo workload trace. The total number of pods offered over the course of each experiment is approximately $150,000$. An additional management node runs the job creation script and records the JCT upon a job's completion. Unless stated otherwise, we run 10 schedulers across nodes. Pods are created as specified in the trace, and each pod is given a default CPU limit of 0.1 cores. These pods run the *sha1sum* application on */dev/zero* device for the duration of the task's actual running time as specified in the trace. Each run takes approximately 20 hours to complete.

Arrival Rates. We study the effects of various arrival characteristics on a Kubernetes cluster. Figure 2 shows the four different arrival patterns of unscheduled pods that we use in our evaluation. Each pattern shows the cumulative number of new unscheduled pods added into scheduler's queue every second, throughout the duration of our experiments.

All four arrival distributions have a median value of 8 - 10 pods (tasks) per second (tps), though their actual arrival rates vary. These arrival patterns vary drastically in their burstiness, defined as the variance in arrival rates with respect to the median [15], as is seen in the tail of these distributions. In A and B, the peak arrival rates are 28 and 40tps respectively, while in C and D they are 195 and 373tps. Hence, A and B are relatively steady as compared to the more bursty arrival rates of C and D. We use these rates to achieve different cluster utilization states in our evaluation. For purposes of motivation, we use pattern A which has a small burstiness and a median arrival rate of 8tps on our 50 node cluster.

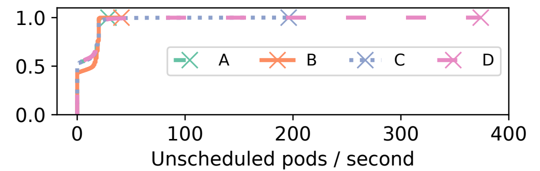


Figure 2: Arrival rates of unscheduled pods.

2.3.2 **Evaluation.** Figure 3 shows the cumulative distribution of x (blue line), average of TST and TCT for all tasks of a job taken over all jobs (green lines), and similarly, tail TST and TCT of jobs (red lines) once the cluster reaches steady-state. Note that the figure discounts the time before the cluster reaches steady-state. We see that the average completion time TCT per job almost equals the average scheduling time TST (green lines overlapping) which indicates that the majority of time a task spends in its lifecycle is on scheduling. The average TCT can be significantly higher than the average execution time x. The tail TCT shows the dramatic effect that multiple scheduling cycles have on TCT. As in the case of

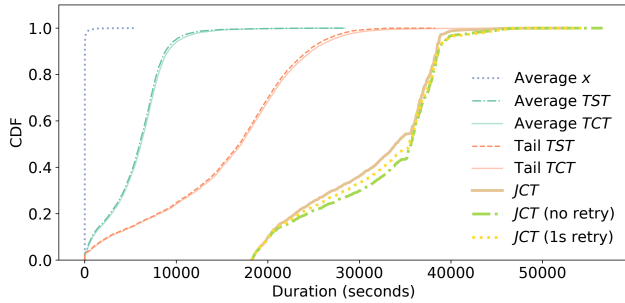


Figure 3: Analysis of JCT in Kubernetes using arrival rate A. Averages are mean taken over all tasks per job.

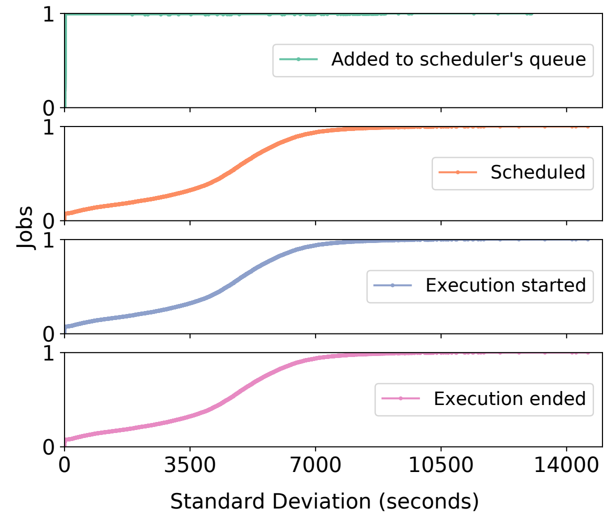


Figure 4: Variance in the completion of stages of lifecycle across tasks of jobs.

the average TCT, the tail TCT is dominated by the tail TST (red lines overlapping) indicating that tail tasks also spend most of their time in scheduling. However, large numbers of scheduling retries for tail tasks increases the scheduling time variance between tasks belonging to a job. The result is that the tail TCT is much higher than the average, leading to longer JCT.

Even when we reduce or eliminate the time that a task spends in the retry queue, we still observe long tail TSTs. We made changes to the default codebase to implement two other versions of Kubernetes. The first is with a reduced retry queue wait time from the default 30sec to 1sec. The second is with no retry queue, where tasks are queued directly in the ready queue when scheduling fails due to resource unavailability. In both these cases, we still observe long tail TST, as seen in Figure 3. This is because while the earlier tasks wait in the retry queue (or at the end of the ready queue), scheduler may attempt to schedule later tasks right when a resource is freed, causing long TCT for earlier tasks.

For this evaluation, we now compare the effect of heterogeneity in our cluster, which is known to cause straggler tasks, load imbalances and increased variance in completion times [1, 2, 4, 78], with scheduling times. We analyze the *spread* of completion times of tasks *belonging to the same job* across all jobs of the experiment above. Figure 4 shows the standard deviation in timestamps of tasks belonging to a job at four different events as identified in Figure 1; when tasks: a) are added to the Kubernetes' scheduler queue (top); b) are scheduled (second from top); c) start their execution (second from bottom); and d) finish their execution (bottom).

The figure shows that tasks of a job are added to the scheduler's queue in a synchronized manner with a mean standard deviation of 1sec (figure on top). Once tasks are scheduled, this value increases to 1.25h (three bottom figures) and tasks belonging to a job are no longer in close temporal proximity. The increased spread in time of tasks of the same job introduced after scheduling remains almost unchanged as tasks progress to execution and completion. This shows that the primary reason for long tail task completion times in highly utilized clusters is scheduling, and that under such cases, the effects of heterogeneity are much smaller.

We conclude by taking note of the fact that a host of schedulers implement policies like fairness that limit multiplexing of tasks [8, 24, 29]. By design, these schedulers leave some tasks behind, experiencing high tail TCT similar to our observation here. These results suggest that to reduce the JCT *the goal should be to reduce the tail task completion time (tail* TCT*) by reducing the dominant tail task scheduling time (tail* TST*) component of centralized schedulers.*

We end this section by noting that the three JCT lines in Figure 3 are moved further to the right from the tail TCT lines, although using Equation 1 we would expect them to be equal or very close. This is because in Kubernetes additional implementation-specific delays occur *before* jobs arrive at the scheduler's queue, primarily caused by rate-limiting in Kubernetes' control plane. The JCT values are obtained using Kubernetes' API and are shifted right owing to these delays. We treat Kubernetes as a black-box and the delay occurring before jobs arrive at a scheduler *are not the focus of this paper*. Our implementation (§5) makes changes only at the scheduling stage, so our comparison against Kubernetes is fair as both systems (default Kubernetes and Murmuration on Kubernetes) experience the same delays before scheduling. We also present an extensive evaluation using simulations which does not suffer from such delays. To summarize, the focus of this paper is on job timings at and after scheduling and not before.

2.4 Distributed Scheduling

Next, we examine how distributed scheduling with worker-side queues affects JCT. Distributed schedulers achieve scalable scheduling by running multiple instances of schedulers that take independent [46, 59] or coordinated [8, 63] decisions for placement. Due to their inherent scalability, they do not impose limits on the cluster sizes that they support as opposed to centralized approaches; e.g., Kubernetes limits the cluster size to 5,000 nodes [49]. Distributed schedulers place tasks on worker-side queues even when nodes do not have free resources at the time of placement.

Figure 1 shows the task lifecycle for Sparrow [59], our reference distributed scheduler. Sparrow employs multiple scheduler instances each with its own scheduling queue ($w2s$). To find a placement, Sparrow probes a limited number of cluster nodes and places a batch of tasks of a job on the least loaded among the probed nodes. Each task is considered for scheduling exactly once (s) and so the TST component is smaller as compared to Kubernetes due to lack of retries. Sparrow queues redundant probes on multiple nodes and only considers the probes that execute the earliest, while canceling the rest.

2.4.1 **Simulator Setup.** We use Eagle's simulator [20] to evaluate the performance of Sparrow (we describe the simulator in more detail in §6.2). Sparrow is implemented with batch scheduling and late binding where a job's probes are queued on all nodes contacted. We simulate a large-scale cluster with 10,000 nodes and an incoming request rate of 2,000tps. Sparrow's evaluation uses different arrival rates and patterns as compared to the evaluation of Kubernetes above. This is because Kubernetes rate-limits input API requests, leading to smaller rate of unscheduled tasks at the scheduler. However, no such limits apply to the simulator.

2.4.2 **Evaluation.** Figure 5 shows the performance of Sparrow using simulations [20]. The tail TCT (solid red line) closely follows the average TCT (dashed green line) as there are no repeated scheduling attempts. This is a desired feature, and one that Kubernetes lacks. However, $w2x$ is significantly large (solid pink line) and dominates the average TCT. This is because the placement quality in Sparrow is degraded under high load conditions where the probability of finding a lightly loaded node is small [20, 22], leading to longer worker-side queues and longer TCT. This is shown by the curves of both the average and tail TCT that have large completion times. The task which encounters the largest $w2x$ among all tasks of a job likely becomes the tail task, since $w2s$ and s are both small in Sparrow. So, though the tail TCT is close to the average TCT, they are both visibly large, leading to a large average JCT.

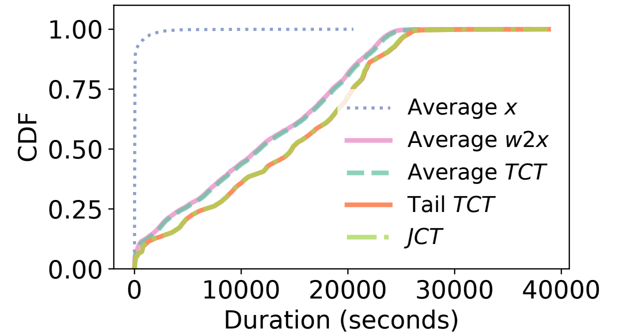


Figure 5: Analysis of JCT in Sparrow.

To summarize, schedulers like Sparrow show a desirable characteristic of near-constant scheduling time for all tasks. This leads to the average and tail TCT values being close to each other, unlike in centralized schedulers. However, scheduling itself samples a few nodes and places tasks on lightly loaded of probed nodes. Under high load, this increases the overall TCT as compared to centralized schedulers. These results suggest that *in distributed schedulers, the average waiting time to execute (average $w2x$) is the dominant component of the average task completion time (average TCT). Therefore, by reducing the average $w2x$, the average TCT, and the average JCT, can be reduced.*

3 DESIGN GOALS

Our overarching goal is to design a scalable scheduler to reduce the average job completion time (JCT) in highly utilized clusters where free resources are scarce. When reducing JCT we focus on reducing tail task completion time (TCT). To this end, *we aim to reduce the variance in the TCT among tasks belonging to a job.* Keeping the variance in TCT small is challenging as tail latency can easily grow in large-scale systems [17]. Our exploratory results earlier suggest that in centralized scheduling TST can be highly variable (like in Kubernetes), while in distributed scheduling $w2x$ can be very high (like in Sparrow). Therefore, among the four different timings (§2.1) that contribute to TCT, we focus on *reducing the variance in the total time tasks of a job spend in being scheduled and waiting to execute, i.e., $TST + w2x$.*

Our work aims to reduce the variance of $TST + w2x$ for all tasks of a job, and ultimately to reduce the tail TCT. We aim to achieve this by ensuring that all tasks of a job (i) spend comparable times at the scheduler (reducing the variance in TST); and (ii) are placed on nodes where they can start executing around the same time (reducing the variance in $w2x$). Our work does not consider how TCT could be further reduced by having a smaller execution time x (by using faster hardware, for example), which is orthogonal to our work.

Our design goals are threefold:
G1 Low Average Scheduling Time: we aim to keep the scheduling time small by reducing the variance in task scheduling time TST among tasks of the same job, similar to distributed schedulers.
G2 Low Variance in Total Wait Times: we aim to ensure that the spread of the time spent by tasks of the same job when waiting to be scheduled and to execute, that is ($TST + w2x$), is small. In this work, we do not focus on the variance in execution time x between tasks of a job.
G3 Scalable Scheduling: we aim to make the new scheduler scalable with respect to the rates of incoming jobs and to the number of nodes in the cluster.

Our overarching goal is to reduce the *average* job completion times in highly utilized clusters (**G0**), achieved as a combination of goals **G1** and **G2**. To achieve our goals we aim to place tasks belonging to the same job fast and with a small variance in their scheduling time, achieving goal **G1**. Tasks are placed on nodes such that they start executing near-simultaneously and as soon as possible, reducing the average tail TCT and average JCT and achieving goal **G2**.

Note that goal **G0** *aims to reduce the average* JCT *by reducing the variance in* $(TST + w2x)$ *among tasks of a job, reducing its tail* TCT. As the tail TCT is reduced for most jobs, the average JCT is improved. We do not aim to reduce the tail of the distribution of JCT itself, though that may happen as a result of smaller tail TCT.

Since we consider our work in the context of bursty traffic, we would like to ensure scheduling can scale to spikes in incoming requests (goal **G3**). We aim to achieve this by deploying multiple scheduler instances capable of handling varying input request rates. In the next section, we design a scheduler that targets these goals and effectively schedules tasks in busy clusters.

4 MURMURATION

We propose Murmuration, a scalable decentralized scheduler designed to reduce tail TCT and average JCT in highly utilized datacenters by reducing the variance in the total wait time between tasks of a job. To achieve this, Murmuration *(i)* schedules tasks in exactly one scheduling attempt and *(ii)* places tasks of the same job on nodes with the smallest expected waiting time *at the time of scheduling*. This approach ensures that all tasks of a job are scheduled quickly and begin execution in close *temporal proximity*.

Our work combines the strengths of both centralized and distributed scheduler designs. Murmuration employs a distributed approach where multiple schedulers place tasks in parallel to handle bursty arrivals. Incoming job requests are load-balanced across schedulers. All tasks of a job are handled by the same scheduler and each scheduler has its own

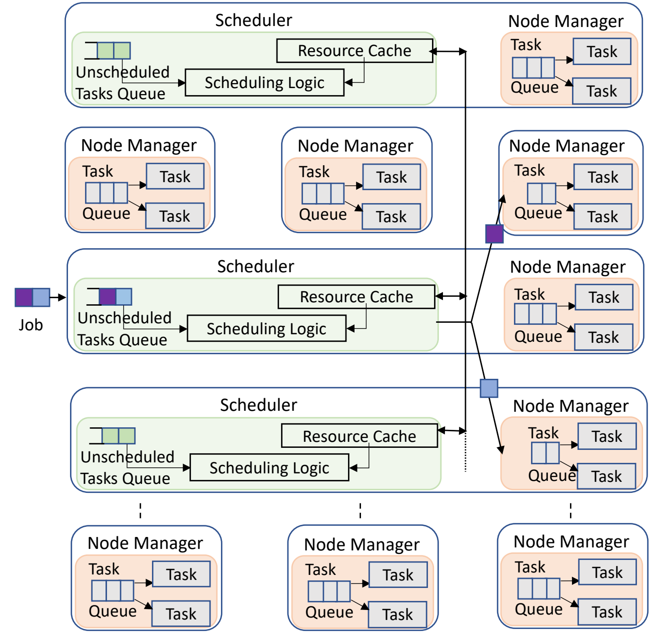


Figure 6: Murmuration's components. Nodes (white blocks) host a node manager and optionally, one or more schedulers.

queue. This leads to shorter scheduler queue lengths as compared to that of a single centralized queue, while ensuring scheduling scales to handle bursts of incoming requests.

Murmuration also employs a scheduling approach similar to centralized where all schedulers share *near up-to-date information on the estimated waiting times at nodes* which is used for making scheduling decisions. In Murmuration, schedulers communicate placement information with each other and build their own local view of waiting times across nodes. This communication between schedulers makes Murmuration a decentralized scheduler rather than a distributed scheduler (§1), ensuring good task placements comparable with that of centralized scheduling (§6).

We consider a datacenter to be composed of thousands of multi-processor machines [16, 27, 69, 74]. When a job arrives for scheduling its arrival time, number of tasks and estimated job running time are provided. Note that estimated job running time is a single value denoting the time that each of its tasks are estimated to run for, though mis-estimations in running times are tolerated (§6). Nodes can process multiple tasks in parallel by allocating a share of their available CPU resources among tasks. We consider CPU as the only dominant resource in this work. However, Murmuration can be extended to include estimated wait times for additional resources like memory, disk and network [28, 32].

4.1 Design Overview

Figure 6 shows Murmuration's main components and their interactions on a cluster. A node executes tasks by running a *node manager* (in orange). It optionally participates in scheduling by hosting one or more *schedulers* (in green) which consume a configurable subset of its resources. Every scheduler has a queue for incoming jobs. When a job is submitted it is assigned a scheduler and its tasks are enqueued in the scheduler's queue. Tasks are dequeued in the order of job creation time, ensuring earliest jobs and their tasks are scheduled first as soon as their required resources are available. This is equivalent to running the first-come-first-served (FCFS) scheduling policy locally at every scheduler. The node manager handles the execution of tasks assigned to a node. It has a single designated worker-side queue for new tasks scheduled on that node, and tasks are serviced in FCFS order.

Schedulers make placement decisions based on a near up-to-date global view of task assignments on nodes. This information is available locally at every scheduler in its *resource cache*, as shown in Figure 6, which contains a list of all nodes and the *expected wait times* for CPU at each node. The expected wait time ($\mathbb{E}_n$) of a node n is defined as the time a newly scheduled task is expected to wait at n before it is selected for execution. $\mathbb{E}_n$ provides the estimated value for $w2x$ at n, and is the sum of the estimated job running time (x) of all tasks t currently queued or executing at n. So, $\mathbb{E}_n = \sum_t x_t$. When scheduling a task, the node with the smallest expected wait time is chosen for placement and ties are broken at random. The figure shows a job with two tasks arrives at a scheduler and its tasks are scheduled at two of the least-loaded nodes in the scheduler's cache. Though this design does not consider the distribution of execution times across the workload, we do so later when describing queue re-ordering techniques (§ 4.2).

To populate values in resource caches, schedulers send and process *resource update messages* using peer-to-peer gossip which enable schedulers to have near-current information on cluster resources. An update message is sent when either a scheduler schedules a new task or a task completes execution on a node. Update messages contain the identity of the node and the estimated running time of the newly placed or completed task. These messages are used to update the estimated wait times of the corresponding nodes in the caches of schedulers. Update messages are processed in the background when schedulers are not actively scheduling a task and therefore, are not in the critical path of the scheduling flow. Note that Murmuration assumes reliable delivery of update messages, even if out-of-order.

Algorithm 1 provides details of Murmuration's scheduling running at each scheduler. As long as a scheduler s has tasks

Algorithm 1: Murmuration Scheduling

```
1  ℕ := set of all nodes
2  𝕊 := set of all schedulers
3  s_q := queue at scheduler s ∈ 𝕊
4  t_cpu, t_mem, t_estTime := t's CPU, memory, estimated
     runtime
5  /* task scheduling at each s ∈ 𝕊                    */
6  while (tasks in s_q) do
7      t = task from the earliest job in s_q
8      t_N = nodes in cache with resources ≥ t_cpu, t_mem
9      t_dst = random (nodes with smallest 𝔼_n in t_N)
10     send t to t_dst
11     UpdateLocCache(t_dst, t_estTime)
12     /* Asynchronously send update message */
13     UpdateRemCache[t_dst, t_estTime] to (𝕊 \ s)
14 /* Asynchronously process new message    */
15 forall (s ∈ 𝕊) do
16     when new UpdateRemCache[t_dst, t_estTime]
17     UpdateLocCache(t_dst, t_estTime)
18 /* task finishing at node n ∈ ℕ           */
19 forall (n ∈ ℕ) do
20     when task t finishes at node n
21     UpdateRemCache[t_dst, t_estTime] to (𝕊)
22 procedure UpdateLocCache (node, x)
23     if (task placement at node) then
24         𝔼_node = 𝔼_node + x
25     else if (task finished at node) then
26         𝔼_node = 𝔼_node - x
```

in its queue, it picks task t with the earliest creation timestamp to schedule (line 7). It selects all nodes from its cache that can satisfy t's CPU and memory requirements (line 8). The node t_{dst} with the smallest estimated wait time is found (line 9), breaking ties at random. t is sent to this node for execution (line 10). Node managers queue newly scheduled tasks when they receive them. As soon as enough resources become available on the node, the first task in the queue is de-queued for execution.

The next steps refer to updating the resource caches of schedulers given the placement of a new task t at the selected t_{dst}. First, t_{dst}'s estimated wait time $\mathbb{E}$ is incremented in the local scheduler s's cache by the estimated running time of the job ($UpdateLocCache()$, line 11). s sends a resource update message to all other remote schedulers about the placement of t at t_{dst} ($UpdateRemCache()$, line 13). When a scheduler receives such a message it updates the wait time of the designated node t_{dst} in its local resource cache (lines

16 - 17). Finally, when a task finishes executing, an update message is sent by the node manager to all remote schedulers to decrement the estimated waiting time of the task's node in their respective local caches (lines 20 - 21).

Figure 7 shows an example flow between two schedulers ($s1$ and $s2$) hosted on two nodes ($n1$ and $n2$). A single incoming job of two tasks ($t1$ and $t2$) and an estimated job running time of 3sec is submitted to the cluster and assigned to $s1$ (1). For scheduling $t1$, $s1$ selects $n1$ which has the least estimated wait time in its cache and sends $t1$ to $n1$ (2) (we assume that $n1$ and $n2$ have the resources to run $t1$ and $t2$). Once received, $n1$ queues $t1$ in its worker-side queue and processes $t1$ immediately since no other tasks are currently running on this node. In (3), $s1$ increments the estimated wait time of $n1$ by 3sec in its local cache and notifies $s2$ of $t1$'s placement on $n1$. When $s2$ receives the update message it increments the value of $n1$'s estimated wait time by 3sec in its local cache. Similarly, (4) and (5) show the scheduling and placement of $t2$ on $n2$ and the notification of the placement to $s2$ respectively. Finally, in (6) and (7) $t1$ and $t2$'s completion notifications are sent respectively by $n1$ and $n2$ to both schedulers $s1$ and $s2$. $s1$ and $s2$ reduce the estimated wait times for the nodes in their respective caches by 3sec.

Size of resource update message. A resource update message contains a node identifier and the estimated runtime of the newly placed or completed task. Assuming that the node identifier is a 4B integer and the estimated runtime is a 4B floating point number, a task generates 8B × 2 = 16B of network overhead for placement and completion notifications. Our workload (§6) has a median fanout of 15 tasks per job, so the median total bytes exchanged per pair of schedulers per job is 240B. Our typical deployments have 10 schedulers, so the network overhead per job is 2.4KB.

Number of Update Messages. The number of update messages exchanged is proportional to the fanout of the workload processed (f) and the number of schedulers deployed in the cluster (S). Schedulers send placement information to all schedulers except themselves, while nodes send task completion notifications to all schedulers. The average number of messages exchanged per job is $f \times (2 \times S - 1)$. Since our workload has a median fanout of $f = 15$ tasks per job (§2.2) and our typical deployment has $S = 10$ (§6), an average of 285 messages are generated per job.

Batched Resource Updates. Murmuration generates a large number of messages when scheduling jobs with high fanout. This creates a large backlog of messages to be asynchronously processed by schedulers. Therefore, we modify Murmuration's design to generate a single resource update message for placing a job instead of as many as its tasks, leading to fewer messages exchanged between schedulers. This message contains the adjusted wait times of all nodes

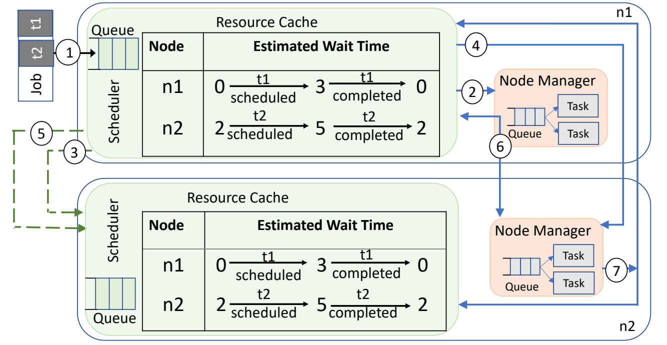


Figure 7: Murmuration example scheduling.

that tasks are placed on. Typically, the adjustment to a node's value is the estimated job running time, but can be larger if multiple tasks are placed on it. This caps the size of update message to the number of cluster nodes. The number of messages that are generated per job is $S - 1$ for placement and $S \times f$ for completion notifications, for a total of $(1 + f) \times S - 1$. This brings it down from 285 messages in the non-batched version to 159 when using batching.

Batching update messages incurs additional delays as the message is sent after all tasks of a job are scheduled. In Murmuration the scheduling time is small owing to the simplicity of its design. Therefore, batching messages does not add significant delay in relaying information to other schedulers. We implement Murmuration's prototype without batched messages (§5), and our simulator with batching (§6.2).

4.2 Discussion

Placement Conflicts and Resource Cache Staleness. Placement conflicts occur when two or more schedulers select the same node simultaneously from their local resource caches as the one with the smallest estimated wait time to schedule their tasks. Murmuration employs a *relaxed* approach towards placement conflicts that arise due to reasons like network and processing delays; once tasks are scheduled they are always sent to the selected node without resolving any potential conflicts. Tasks are placed in worker-side queues for execution in the order they arrive. Eventually, despite thousands of nodes and high fanout jobs, Murmuration's resource caches have a near up-to-date view of wait times although the actual order in node queues may vary from the one assumed by the schedulers. Murmuration is designed to reduce $w2x$ across jobs by balancing out the *total* estimated wait time at each node.

Figure 8 shows an example where both $s1$ and $s2$ select w (with estimated wait time of $10s$ in both schedulers) to place their tasks t_x (estimated running time of $3s$) and t_y (estimated running time of $4s$) respectively. Initially, each

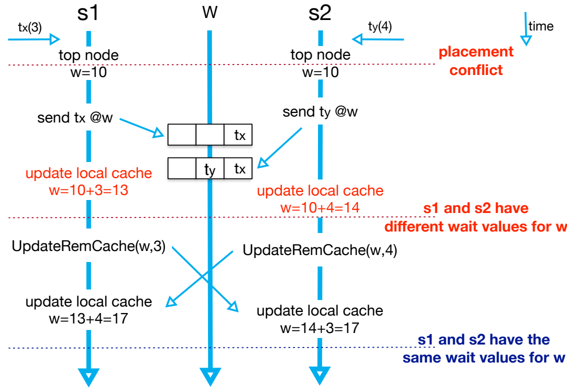


Figure 8: Example placement conflict of two tasks t_x and t_y both placed on node w by schedulers $s1$ and $s2$.

scheduler updates its own local cache assuming that its task is appended to the end of w's queue, and *temporarily* have different wait times for w. However, after the two schedulers exchange *UpdateRemCache* messages they both are updated with the true wait time of w despite conflicting placement. Note that the arrival order of the two tasks at w is random.

Message Update Delays. The above example also demonstrates the lag in schedulers' cache updates because of inherent delays in network propagation and message processing at sending and receiving schedulers. In such cases, schedulers may select nodes which might not have the true smallest estimated wait time. The degree to which the caches are out-of-sync depends on job inter-arrival time and update message delays, and we evaluate these effects later (§6.2.3).

Fault Tolerance. In Murmuration, schedulers maintain two states, namely, schedulers' queues containing unscheduled tasks, and resource caches that store expected wait times at nodes. When schedulers crash or restart they lose both states. Upon scheduler failures, Murmuration's underlying scheduling framework re-creates affected tasks, if required, and assigns them to active schedulers [75].

A newly (re-)started Murmuration scheduler initializes its resource cache with an approximate state. For this, it requests the cache from two randomly chosen schedulers. When such a request is received, a scheduler only responds if its resource cache is already initialized. The requesting scheduler initializes its cache with the first response it receives from a scheduler. If no such response is received within a configured timeout period, it requests two other schedulers and so on, till it receives a response or exhausts all schedulers in its list. In the latter case, the scheduler proceeds with an empty cache, similar to the case when the cluster is first initialized. Our evaluation shows that Murmuration is able to schedule tasks well using such *approximate states* (§6.2.4).

Queue Re-ordering Techniques. In Murmuration, schedulers schedule tasks of jobs, and node managers dequeue tasks for execution, using FCFS to process their queues (§4). We choose FCFS owing to its simplicity (§5). However, in strict FCFS, long tasks executing before shorter ones cause head-of-line (HoL) blocking, leading to increased wait times for tasks queued behind them [17, 63, 79]. Murmuration's performance can be improved using job and task-related information available to both schedulers and node managers to make better decisions on processing their queues. We aim to use queue re-ordering to further improve the average JCT by increasing the throughput of completed jobs in the system (goal **G0**). Although we do not explicitly evaluate Murmuration with starvation avoidance techniques, they can be easily plugged in [63].

To increase the throughput of jobs, we aim to schedule the "smallest" of jobs at the scheduler first. We use the **Shortest Job First (SJF)** policy that selects the job with the smallest cumulative estimated running time of its tasks. The intuition is that more number of jobs will finish in a given time, improving the average JCT. Note that since jobs get distributed among multiple schedulers, re-ordering jobs at a scheduler provides only a partial ordering of jobs in the cluster.

At the node manager, we use the **Shortest Task First (STF)** policy that selects the task with the shortest estimated running time to execute first. This helps avoid the problem of HoL blocking and leads to faster average task completion time. We use the notations Murmuration-FCFS to mean FCFS at the scheduler and FCFS at the node manager, and Murmuration-SRJF (Murmuration-Shortest Remaining Job First) to mean SJF at the scheduler and STF at the node manager. We evaluate the performance of Murmuration-FCFS and Murmuration-SRJF with respect to other well-known schedulers (§6.2.1).

Data Locality. Data locality is an important consideration for schedulers [81]. As future work, data locality can be integrated into Murmuration by incorporating it as a metric along with estimated wait time (Algorithm 1, line 9) when scoring nodes for task placement.

5 IMPLEMENTATION

We now describe the implementation of Murmuration on Kubernetes' stable release version 1.23. A Kubernetes cluster consists of one or more machines (nodes) hosting application containers and control plane services. The control plane contains a set of core Kubernetes services responsible for managing the cluster and executing jobs. These services include etcd, kube-apiserver, kube-controller-manager and kube-scheduler. The kubelet service runs on each node in the cluster and acts as a node agent. The smallest deployable unit is a pod; a logical object representing a collection of one

or more containers. In our implementation, we have a one-to-one mapping between a task and a pod, used interchangeably hereafter.

etcd [26] is the centralized datastore that stores user and cluster data. kube-controller-manager provides various controllers, like the node-controller and the job-controller, that monitor and react to changes in the states of objects, like nodes and jobs, so that active jobs complete. kube-scheduler schedules all newly created pods (§2.3). Finally, kubelet collects the required data and images and starts containers for pods assigned to the node.

5.1 Murmuration in Kubernetes

Murmuration is a decentralized scheduler where multiple scheduler instances schedule pods in parallel. We implement Murmuration in Kubernetes by using the underlying framework wherever possible. In this section, we focus on the main modifications in the kube-scheduler and kubelet services.

Kubernetes has a centralized scheduler although it supports the deployment of multiple instances. Each scheduler instance has its own unique name. If an incoming pod specifies a scheduler's name then it is sent to that scheduler. If no scheduler is specified in a pod's specification, then it is scheduled by the default kube-scheduler. Murmuration's deployment uses multiple instances of the same kube-scheduler binary, however, each of these instances is assigned a unique name. A new job creation request (§2.3.1) specifies a scheduler name for its pods. When these pods are created, they are sent to the specified scheduler.

Scheduling. We modify the existing cache of kube-scheduler to store the estimated wait times of nodes. During a scheduling cycle, the scheduler first finds feasible nodes that have the required resources (Algorithm 1 line 8), sorts them according to their estimated wait times, and schedules a pod on the node with the smallest wait time (lines 9-10). Murmuration scores all nodes unlike a subset of nodes that Kubernetes uses. The modified kubelet service manages the multiplexing of running tasks at nodes.

Resource update messages. kube-scheduler uses the *bind* API call to commit the placement of a task on a node in etcd. Murmuration additionally uses this API for schedulers to communicate placement information with each other. Every pod's specification contains the estimated job running time as an environment variable. When a scheduler completes a placement or receives a *bind* notification from other schedulers, it extracts the pod's estimated running time to update the node's wait time in its cache (Algorithm 1, lines 12 - 17). Similarly, schedulers update a node's wait time in their caches when they receive pod completion notification from kubelet (Algorithm 1, lines 18 - 21).

Worker-side queues. We implement worker-side queues in kubelet for queuing pods assigned to a node. All pods are queued according to their creation timestamps. kubelet is notified of available resources when a task finishes execution and releases its resources. kubelet allocates these resources to as many queued pods as possible, until the node's resources are fully occupied and no further pods can be accommodated. Our current implementation supports unbounded queues, although the length of the worker-side queue can be bound depending on the CPU and memory resources available to kubelet. All evaluations for Murmuration completed successfully without implementing bounds on worker-side queues (§6.1).

Though we describe only the default implementation of Murmuration, we made additional changes to support queue re-ordering and fault tolerance. We implemented SJF re-ordering in kube-scheduler, and STF in kubelet (§4.2). To support fault tolerance we made changes to kube-controller-manager. These changes include implementing a constantly updated list of active cluster schedulers and nodes, detecting scheduler failures and assigning other active schedulers to jobs affected by these failures, and managing a job's lifecycle to handle node failures.

6 EVALUATION

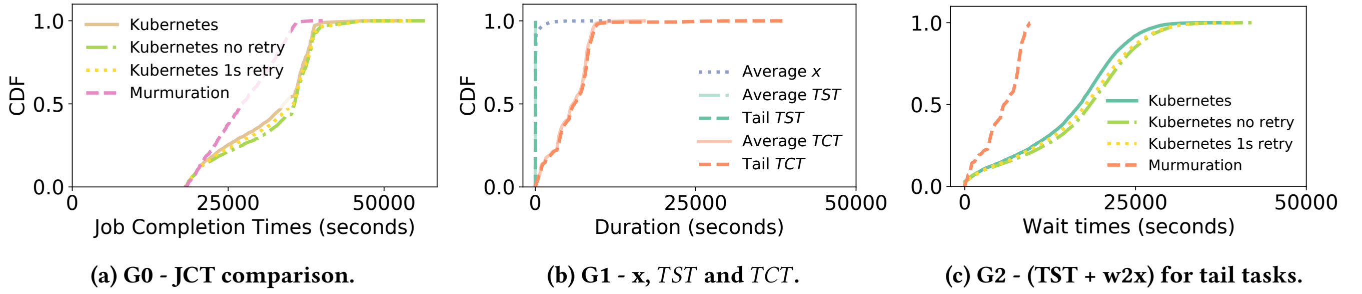
Murmuration is designed to schedule jobs comprising many parallel tasks, as found in modern workloads [69]. We evaluate Murmuration's performance using industry workloads (§2.2) both as a prototype evaluation on CloudLab cluster (§6.1) and as trace-driven simulations (§6.2). In particular, we evaluate our design goals as follows:

G0: Low Job Completion Times: We run the workloads for different job arrival rates (§2.2) and evaluate Murmuration's performance with respect to existing schedulers (§6.1 and §6.2). Our evaluation shows that Murmuration has a 25% smaller median job completion time JCT as compared to the default Kubernetes' scheduler for different request arrival rates. Murmuration-SRJF re-ordering has a JCT that is $100X$ better than that of Sparrow, Yaq-d and Eagle.

G1: Low Scheduling Times: We evaluate task scheduling time TST, and compare the average and tail TST in Murmuration against our evaluation of centralized and distributed schedulers (§2). We show that Murmuration has a small TST for all tasks, including tail tasks (§6.1).

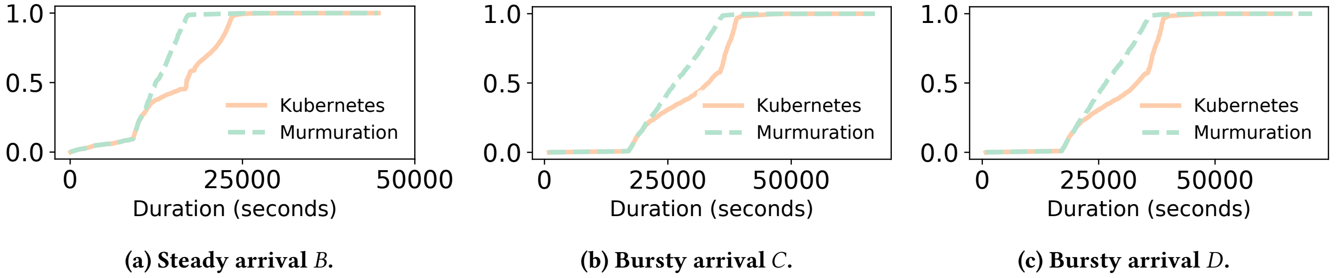
G2: Low Total Waiting Times: We evaluate the total task wait time ($TST + w2x$) for Murmuration, and show that the variance in this value is small among tasks of the same job as compared to our evaluation of centralized schedulers (§6.1).

G3: Scalable Scheduling: We show that Murmuration achieves high throughput by scheduling millions of pods in minutes.



(a) G0 - JCT comparison. **(b) G1 - x, *TST* and *TCT*.** **(c) G2 - (TST + w2x) for tail tasks.**

Figure 9: Comparison of Murmuration against Kubernetes for Figure 3 for arrival A.



(a) Steady arrival B. **(b) Bursty arrival C.** **(c) Bursty arrival D.**

Figure 10: JCT comparison for different arrival rates.

Additionally, we evaluate the effect of a) inaccuracies in estimating job running times (§6.2.2); and b) delays in processing update messages due to network propagation and asynchronous communication among the schedulers (§6.2.3).

6.1 Prototype Evaluation

We evaluate the performance of Murmuration under varying input request characteristics of steady-state arrivals A and B, and bursty arrivals C and D (§2.3). In all prototype evaluations, *Murmuration incorporates baseline error of* ±9% *on the median and* ±100% *on the* 99th *percentile in estimated running times with regard to actual task runtimes.* Estimated job runtime is the average of actual task durations of a job, as specified in workload traces. Errors are the variance in the average and actual task durations taken over the entire workload. *Additional median error of* ±15% *and* 99th *percentile error of* ±50% *are incorporated in running time estimates* [8]. Murmuration's prototype and deployment scripts are available as open-source repositories [75, 77].

We evaluate our system's goals (§3) and compare it against Kubernetes. Figure 9 shows the comparison against three different versions of kube-scheduler: *(i)* the default version of kube-scheduler; *(ii)* a modified version where pods that fail scheduling due to unavailable resources wait for a maximum of 1sec in the retry queue before returning to the ready queue (Kubernetes 1s retry); and, *(iii)* a third version where pods are sent directly to the ready queue if they fail scheduling due to resources unavailability (Kubernetes no retry) (§2.3.2). This evaluation is the Murmuration equivalent of Figure 3, run with steady arrival pattern A (Figure 2).

Figure 9a shows how the different JCTs compare once the cluster reaches steady-state. Murmuration (pink line) reduces the median JCT by 20% — 25% as compared to the different versions of kube-scheduler, and by 13% — 18% for the 99th percentile JCT. This shows that Murmuration is able to achieve goal **G0** of low JCT in highly utilized clusters.

Figure 9b shows the cumulative distribution of x (dotted blue line), average and tail TST (dashed green lines), and average and tail TCT (orange and dashed red lines). We see that the average and tail TST overlap and show a near-constant value with a median of $5ms$, unlike Kubernetes where the tail TST is much larger than that of other tasks. This proves that Murmuration achieves goal **G1** of having a small scheduling time. Note that both the average and tail TCT curves overlap suggesting that all tasks belonging to a job, including tail tasks, complete at nearly the same time.

Figure 9c compares the total wait times for tasks as compared to the three versions of kube-scheduler. The wait times in Murmuration are 3× smaller than in Kubernetes for both the median and 99th percentile, proving it achieves goal **G2** of smaller total wait times ($TST + w2x$).

Figure 10 shows the cumulative distribution of JCT for both Murmuration and Kubernetes for steady arrival B, and

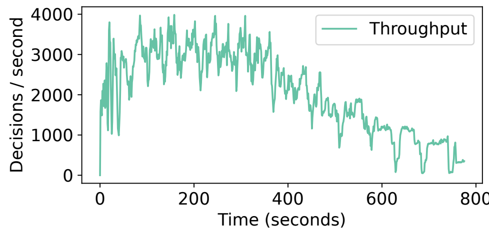


Figure 11: G3: Scheduling throughput in Murmuration.

bursty arrivals C and D. Murmuration (green line) has a better performance as compared to Kubernetes with a $16-20\%$ smaller median JCT in all three cases. Note that Murmuration's tail of JCT distribution for A, B and C is smaller or comparable to that of Kubernetes, while for D it is longer by 7%. As stated earlier (§3), Murmuration's goal **G0** is to reduce the *average* JCT and not the tail of its distribution.

To summarize, we evaluated Murmuration under conditions of normal and high utilization with a median arrival rate of $8-10$ tasks per second on our 50 node cluster (§2.3). We find that in all scenarios, Murmuration reduces the median JCT by $16-25\%$ as compared to default Kubernetes scheduler.

Finally, we present Murmuration's ability to scale to high incoming workloads. We use a CloudLab cluster with 100 nodes and 10 schedulers. 500 jobs arrive following pattern C sped up by $100X$, for a total of 1.6 million pods. Figure 11 shows that schedulers achieve a peak scheduling of 3900 decisions per second, and are able to place all 1.6 million pods in around 13 minutes. This shows that Murmuration is able to handle extremely large and bursty input request rates, achieving goal **G3**.

6.2 Evaluation using Simulation

We present the simulation of Murmuration using Eagle's simulator [20] which provides an implementation of Eagle hybrid scheduler, and Sparrow and Yaq-d distributed schedulers. In Yaq-d [63], schedulers build local cluster state using heartbeat messages sent by nodes periodically. It incorporates various local queue management and task reordering techniques at nodes with a goal to reduce fallow resources when tasks complete.

Eagle [20] is a hybrid scheduler that partitions a cluster to execute long and short jobs separately to avoid HoL. It uses various techniques to improve JCT such as executing long jobs in order of their cumulative task executions times, avoiding placing short tasks on nodes where long ones execute, executing multiple tasks of a job using a single probe, and using SRPT for executing tasks at nodes.

We extend the simulator by adding support for Murmuration (§5) in order to have a fair comparison against these schedulers. We simulate small datacenters with $1,000$ nodes and large ones up to $15,000$ nodes. We evaluate the systems using both Yahoo and Cloudera traces (§2.2) with a total of $10,000$ jobs arriving at a rate of $2,000$ tasks per second. We use a standard deployment of 10 schedulers, unless specified otherwise. The simulation assumes all machines are switched on for the entire duration of the experiment. We report the median JCT and utilization for each datacenter size. Utilization is computed as the ratio of the total time nodes in the cluster are busy executing tasks to the total node seconds elapsed in the cluster. The simulator extended with support for Murmuration is available as open-source [76].

For Yaq-d, nodes periodically advertise their estimated waiting times to the schedulers using heartbeats. Since we speed up workload arrival by $1,000$ times, hence we have used 8 millisecond as the heartbeat interval as compared to the recommended value of 8 seconds [20], so that Yaq-d accesses the current state of nodes. We note that this value is too small for real-world deployments. We use all other recommended parameter values for schedulers [19]. Note that the original simulator implements *SRPT* for both Eagle and Yaq-d, and additional job-level re-ordering in Eagle. Hence, we simulate two version of Murmuration, Murmuration-FCFS that has no re-ordering of jobs and tasks, and Murmuration-SRJF that incorporates SJF and STF re-ordering policies (§4.2).

6.2.1 **Murmuration's Performance.** Figures 12 and 13 show results using the Yahoo and the Cloudera traces respectively. Labels atop every bar show the multiplier with respect to Murmuration-SRJF's JCT. Utilization ranges from near-saturation in clusters comprising $1,000$ nodes to lightly-loaded in $15,000$ node clusters.

For both Yahoo and Cloudera workloads, Murmuration-SRJF outperforms other schedulers across all datacenter sizes with the median JCT being two orders of magnitude smaller. Sparrow performs better than expected in our simulations since batch scheduling with late binding samples a large number of nodes for each of these high fanout jobs. Recall that Murmuration-FCFS uses no re-ordering while all other schedulers use one or both job-level and task-level re-ordering, leading to better performance. Therefore, we primarily compare their performance with that of Murmuration-SRJF.

Yaq-d has a much larger median JCT because its schedulers rely on snapshots of node statuses taken every 8 milliseconds, rather than maintaining up-to-date estimates themselves. For bursty arrivals, such snapshots quickly turn stale. Eagle is a hybrid scheduler that partitions a cluster into two sub-clusters for long and short jobs (§7). These sub-clusters have periods of under- and over-utilization for heterogeneous workloads, leading to increased median JCT.

Though Murmuration-SRJF performs the best, we note that its utilization is low. This is because the last job in

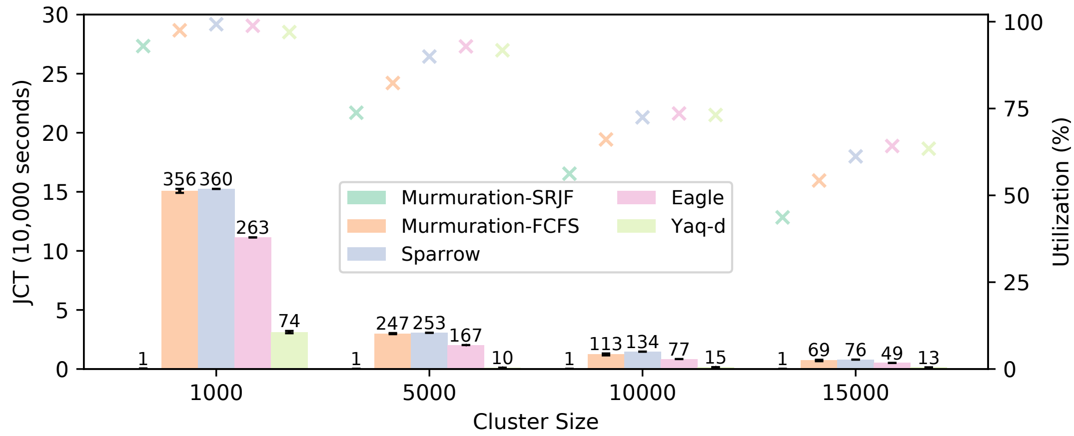


Figure 12: 50th percentile JCT comparison for the Yahoo trace. Numbers on bars represent JCT relative to Murmuration with SRJF. Dots indicate the cluster's utilization in the corresponding system.

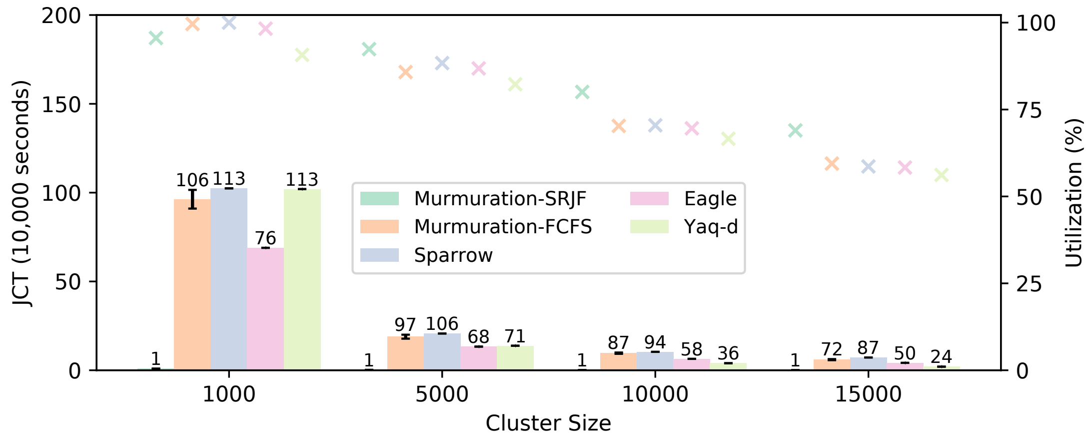


Figure 13: 50th percentile JCT comparison for the Cloudera trace. Numbers on bars represent JCT relative to Murmuration-SRJF. Dots indicate the cluster's utilization in the corresponding system.

Murmuration-SRJF takes longer to finish. Since we consider all machines to be switched on till the end of the simulation, some machines are idle towards the end, bringing down the overall utilization. As noted earlier, the focus of Murmuration is to reduce the median and not the tail JCT.

6.2.2 **Mis-estimations in Running Times.** As previously mentioned, estimated job runtimes are provided as a part of the workload traces. While both Kubernetes and Sparrow do not depend on runtime estimations, Murmuration uses them to calculate node wait times and to make placement decisions. However, these estimates are not always accurate [8, 14, 21], and analysis shows a median estimation error of 15% and a 99th percentile error of 50% [8].

To study the effect of such inaccuracies on Murmuration we artificially introduce mis-estimation in running times provided in the workload traces. We create two workloads where *all* jobs are mis-estimated. In the first workload, if r denotes the running time of a job, we consider job running times to be uniformly mis-estimated in the range $[0.85{*}r, 1.15 * r)$. In the second workload, all jobs are mis-estimated in the range $[0.5 * r, 1.5 * r)$. We consider these two workloads to approximate the median and tail of mis-estimations observed in the real-world.

Figure 14 shows the effect of mis-estimations on quality of scheduling. For the first workload with jobs mis-estimated uniformly in $[0.85{*}r, 1.15 * r)$ (orange line), Murmuration shows a 3% increase in JCT over all percentiles. For the

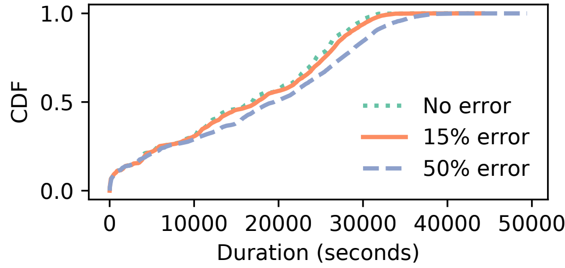


Figure 14: Impact of running time mis-estimations.

second workload (dashed blue line), the increase in JCT is between 15 — 18%; however such large errors in mis-estimations account for only 1% of mis-estimations observed. Therefore, Murmuration's performance does not degrade too much under observed inaccuracies in job running times. In Murmuration, the cumulative mis-estimation of job running times gets spread uniformly across different nodes. This enables schedulers to select lightly-loaded nodes despite inaccuracies in the actual node wait times.

6.2.3 **Delayed Scheduler Updates.** Schedulers experience delays in processing update messages which are handled asynchronously in the background. These messages are also optionally batched and sent out periodically, or configured to be sent upon a minimum number of updates. We study the effect of such delays over large scheduler deployments.

In Figure 15, we vary the number of schedulers deployed from 25% to 100% in a 5, 000 node cluster. We simulate delays of 100 milliseconds to 100 seconds in update message processing, typical of applications delays in background processing [47]. Since the input arrival rate is 2, 000tps, these delays translate to respectively 200 to 200, 000 messages in transit per second. The figure shows the median JCT relative to the ideal case of no delay.

We see that the JCT increases as the update delays increase due to schedulers working on increasingly stale resource caches. For example, at 1 second delay, the JCT is 1.08 times that at 100 milliseconds. At 10 second delay, the JCT rise to 1.85 times. Further, we observe that JCT increases as the number of schedulers increases; when the number of schedulers deployed is small, every scheduler handles more number of jobs, and hence, their resource caches are more up-to-date. As the number of deployed schedulers increases, the arrival of jobs at each individual scheduler decreases, resulting in greater discrepancies among scheduler caches.

6.2.4 **Fault Tolerance.** We evaluate the effect of restarting a scheduler that is operational and actively processing jobs. The aim is to assess the quality of resource caches that new schedulers are initialized with. For this, we modify our

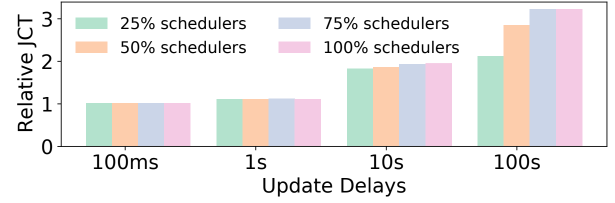


Figure 15: Effect of update delays on median JCT.

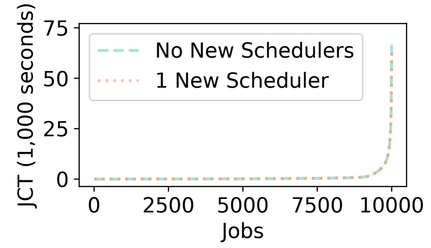


Figure 16: Performance on addition of a new scheduler.

simulator to randomly select a scheduler from among 10 schedulers at the start of the evaluation, and delay its start by $t = 10$s. We chose this value because this is the time taken by a scheduler to finish initializing in a Kubernetes cluster running Murmuration's prototype. Upon start, the scheduler requests two randomly selected schedulers for their caches (§4.2) which return a deepcopy of their caches.

All scheduler-to-scheduler communications incur a network delay of 5 milliseconds [19]. Scheduling time is given by $t_{decision} = t_{job} + t_{task} \times tasks_per_job$, where $t_{job} = 0.1$ seconds and $t_{task} = 5$ milliseconds [64]. To simulate real-world schedulers that delay background processing, responses sent by schedulers incur an additional overhead uniformly distributed in (0, 3) seconds. The newly restarted scheduler accepts the first cache response, and gets added to the list of active schedulers and starts scheduling jobs. We simulate only a single scheduler failure.

Evaluation. We run our simulation multiple times to identify how jobs fare when placed by a newly added scheduler versus when handled by existing active schedulers. Figure 16 shows that the performance is very similar and that the new scheduler shows a slight improvement in the tail of the JCT. This is because the queue of the newly added scheduler has smaller number of jobs waiting, leading to a relatively smaller $w2s$. When scheduled by already active schedulers, these jobs wait longer behind already queued jobs which reflects in larger tail JCT. This further proves the effectiveness of Murmuration in reducing scheduler wait time $w2s$.

7 RELATED WORK

Scheduling Designs. A large body of work exists on schedulers, broadly centralized [29, 38, 41, 42, 68, 72, 74], distributed [8, 24, 46, 59] and hybrid [20, 22, 27, 44, 79, 83]. We compare Murmuration with Kubernetes, Yaq-d and Sparrow as representative systems and show its effectiveness in highly utilized clusters. Though Murmuration relies on job estimates, it performs well even with inaccurate estimates. Recently, there has been a surge in using machine learning techniques for scheduling [54, 58, 82]. Though promising, these techniques depend on the quality of training data which need to be representative of the workloads. Murmuration can be extended to include such learning techniques in a decentralized context, similar to Pronto [31] that uses an aggregate of local models to build a global view of the cluster.

Overloaded Clusters. Bronson and Huang et al. [9, 40] refer to "the state of a permanent overload with an ultra-low goodput (throughput of useful work)". They describe different reasons beyond increased arrival rates that cause permanent overloads, analyzing several cases from Google, AWS, Azure and others. In our paper, we show that certain scheduling designs also cause increased completion times. Murmuration addresses this performance degradation, and could be used in conjunction with other load limiting policies.

Estimated Job Runtime. Some schedulers use runtime estimates to perform scheduling decisions [8, 33, 60, 63, 71] while others [21, 39] argue against using them. Yaq [63] and Apollo [8] are closely related with Murmuration and also use runtime estimates for scheduling. Yaq [63] implements both a centralized (Yaq-c) and a distributed (Yaq-d) scheduler with worker-side queues. It explores placement quality using various local and global queue management and job reordering techniques, which is complementary to Murmuration.

Apollo [8] is a loosely coordinated framework with multiple schedulers and a central resource monitor to aggregate load information. It uses a wait-time matrix to schedule tasks on nodes that have sizable task inputs, are located in the same rack, or are lightly loaded. Apollo adopts a token-based execution of jobs that subjects it to long tail TCT. In contrast, Murmuration is decentralized and every scheduler instance maintains its own resource view based on job estimates. It focuses on improving the average *tail* TCT in busy clusters.

Prior work propose backfill and its variants [34, 57, 70] for batch workloads. These allow short jobs to utilize idle resources out-of-turn by selecting appropriate batches of jobs to occupy nodes. In Murmuration's online scheduling, jobs arrive at different times. Worker-side queues help prevent idle resources by queuing tasks ready for execution as they arrive. Queue re-ordering (§4.2) can be applied to achieve better utilization, as in backfill scheduling.

Tail Latencies. Reducing tail latency has been studied extensively. Plenz et al. [61] show queuing is one of the primary reasons for increased tail latencies. Li et al. [51] show that non-FIFO scheduling and multi-core operations with multiple queues increase application tail latencies. They propose multiple processors process a single common queue to help reduce tail latency, as designed in Murmuration. Kernel and cache level optimizations are further suggested for improving tail latencies [23, 45]. Others [35, 43] target small tail latency for short interactive services and microsecond scale applications. Reducing latency using straggler replication is also well-studied [2, 4, 36, 78]. These approaches replicate straggler tasks on machines for reducing the execution times of tasks, and are orthogonal to our work.

Kubernetes. Senjab et al. [65] identify the need to evaluate Kubernetes on large clusters and on dynamic workloads across heterogeneous environments to identify scalability bottlenecks. Further, Larsson et al. [50] find Kubernetes to be unsuitable for edge computing scenarios, attributing its single-point failure and lack of scale to its centralized scheduling. In this work, we show that the performance of Kubernetes degrades under heavily-loaded conditions and that Murmuration's scheduling approach is better designed to handle such challenging conditions. Senjab et al. also propose that scheduling should take into account application characteristics, as is done in Murmuration.

8 CONCLUSION

Murmuration is a decentralized scheduler which is effective in handling scheduling under conditions of both normal and bursty arrivals in datacenters. Murmuration reduces the total wait time tasks face when waiting to be scheduled in scheduler queues and in lengthy worker-side queues. It achieves this by having scheduler instances communicate their placement decisions to build their own nearly-updated view of cluster resources. Our evaluations show that Murmuration reduces the tail task completion times as well as the average job completion times by 25%. Murmuration, its simulator and deployment scripts are open-source [75–77].

9 ACKNOWLEDGMENTS

We would like to thank Richard Mortier, Jon Crowcroft, Keshav and the rest of the Cambridge Systems Research Group for their valuable feedback on this work. We would also like to thank our shepherd, Matthias Boehm, and SoCC 2024 reviewers for their comments which helped improve this paper. Finally, we are grateful to Huawei Technologies for funding this work as a PhD studentship.

REFERENCES

[1] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. 2012. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 61–74. https://doi.org/10.1145/2150976.2150984

[2] Mehmet Fatih Aktas, Pei Peng, and Emina Soljanin. 2018. Straggler Mitigation by Delayed Relaunch of Tasks. *SIGMETRICS Perform. Eval. Rev.* 45, 3 (Mar 2018), 224–231. https://doi.org/10.1145/3199524.3199564

[3] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. 2018. On the Diversity of Cluster Workloads and Its Impact on Research Results. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 533–546.

[4] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(NSDI'13)*. USENIX Association, USA, 185–198.

[5] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. 2020. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 782–797. https://doi.org/10.1145/3387514.3406221

[6] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 556–573. https://doi.org/10.1145/3447786.3456259

[7] Angel M. Beltre, Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E. Grant. 2019. Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 11–20. https://doi.org/10.1109/CANOPIE-HPC49598.2019.00007

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 285–300.

[9] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. 2021. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 221–227. https://doi.org/10.1145/3458336.3465286

[10] Jeferson R. Brunetta and Edson Borin. 2019. Selecting Efficient Cloud Resources for HPC Workloads. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing* (Auckland, New Zealand) *(UCC'19)*. Association for Computing Machinery, New York, NY, USA, 155–164. https://doi.org/10.1145/3344341.3368798

[11] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. 2014. Long-term SLOs for reclaimed cloud computing resources. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670999

[12] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proc. VLDB Endow.* 5, 12 (Aug 2012), 1802–1813. https://doi.org/10.14778/2367502.2367519

[13] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. 2011. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '11)*. IEEE Computer Society, USA, 390–399. https://doi.org/10.1109/MASCOTS.2011.12

[14] Emilio Coppa and Irene Finocchi. 2015. On Data Skewness, Stragglers, and MapReduce Progress Indicators. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) *(SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 139–152. https://doi.org/10.1145/2806777.2806843

[15] M.E. Crovella and A. Bestavros. 1997. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking* 5, 6 (1997), 835–846. https://doi.org/10.1109/90.650143

[16] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, and et al. 2019. Hydra: A Federated Resource Manager for Data-Center Scale Analytics. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 177–191.

[17] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb 2013), 74–80. https://doi.org/10.1145/2408776.2408794

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

[19] Pamela Delgado. 2015. Eagle Simulator. https://github.com/epfl-labos/eagle

[20] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-Aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 497–509. https://doi.org/10.1145/2987550.2987563

[21] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2018. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 135–148. https://doi.org/10.1145/3267809.3267838

[22] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid Datacenter Scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 499–510. https://www.usenix.org/conference/atc15/technical-session/presentation/delgado

[23] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PerséPhone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 621–637. https://doi.org/10.1145/3477132.3483571

[24] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. 2014. Decentralized Task-Aware Scheduling for Data Center

Networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 431–442. https://doi.org/10.1145/2619239.2626322

[25] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[26] etcd. 2024. A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io

[27] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3190508.3190549

[28] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 323–336.

[29] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 99–115.

[30] Google. 2017. Addressing Cascading Failures, Site Reliability Engineering. https://sre.google/sre-book/addressing-cascading-failures/

[31] Andreas Grammenos, Evangelia Kalyvianaki, and Peter Pietzuch. 2021. Pronto: Federated Task Scheduling. arXiv:2104.13429 [cs.DC]

[32] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466. https://doi.org/10.1145/2740070.2626334

[33] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 65–80.

[34] Isaac Grosof, Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. 2022. Optimal Scheduling in the Multiserver-Job Model under Heavy Traffic. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 3, Article 51 (dec 2022), 32 pages. https://doi.org/10.1145/3570612

[35] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 161–175. https://doi.org/10.1145/2694344.2694384

[36] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 161–175. https://doi.org/10.1145/2694344.2694384

[37] Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action* (1st ed.). Cambridge University Press, USA.

[38] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 295–308.

[39] Zhiming Hu, Baochun Li, Zheng Qin, and Rick Siow Mong Goh. 2017. Job Scheduling without Prior Information in Big Data Processing Systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 572–582. https://doi.org/10.1109/ICDCS.2017.105

[40] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 73–90. https://www.usenix.org/conference/osdi22/presentation/huang-lexiang

[41] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 261–276. https://doi.org/10.1145/1629575.1629601

[42] Morris A. Jette and Tim Wickberg. 2023. Architecture of the Slurm Workload Manager. In *Job Scheduling Strategies for Parallel Processing*, Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo (Eds.). Springer Nature Switzerland, Cham, 3–23.

[43] Kaffes, Kostis and Chong, Timothy and Humphries, Jack Tigar and Belay, Adam and Mazières, David and Kozyrakis, Christos. 2019. Shinjuku: Preemptive Scheduling for Msecond-Scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 345–359.

[44] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 485–497.

[45] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient Cache Sharing with Strict Qos for Latency-Critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 729–742. https://doi.org/10.1145/2541940.2541944

[46] Mansour Khelghatdoust and Vincent Gramoli. 2018. Peacock: Probe-Based Scheduling of Jobs by Rotating Between Elastic Queues. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11014)*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer, 178–191. https://doi.org/10.1007/978-3-319-96983-1_13

[47] Kubernetes. 2021. https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scheduling/scheduling_code_hierarchy_overview.md.

[48] Kubernetes. 2023. Why the number of pods per node should not exceed 110? https://github.com/kubernetes/kubernetes/issues/119391

[49] Kubernetes. 2024. An open-source system for automating deployment, scaling, and management of containerized applications. https://kubernetes.io/docs/home

[50] Lars Larsson, Harald Gustafsson, Cristian Klein, and Erik Elmroth. 2020. Decentralized Kubernetes Federation Control Plane. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 354–359. https://doi.org/10.1109/UCC48980.2020.00056

[51] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670988

[52] The Amazon Builders' Library. 2024. Using load shedding to avoid overload. https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/.

[53] L. Mai, E. Kalyvianaki, and P. Costa. 2013. Exploiting Time-Malleability in Cloud-based Batch Processing Systems. (2013). https://openaccess.city.ac.uk/id/eprint/8179/ Unpublished.

[54] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[55] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A Large Scale Study of Data Center Network Reliability. In *Proceedings of the Internet Measurement Conference 2018* (Boston, MA, USA) *(IMC '18)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3278532.3278566

[56] Justin J. Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, David Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard. 2023. Defcon: Preventing Overload with Graceful Feature Degradation. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 607–622. https://www.usenix.org/conference/osdi23/presentation/meza

[57] A.W. Mu'alem and D.G. Feitelson. 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems* 12, 6 (2001), 529–543. https://doi.org/10.1109/71.932708

[58] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation.* 481–498.

[59] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[60] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 2018. 3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 17 pages. https://doi.org/10.1145/3190508.3190515

[61] Julius Plenz. 2019. How to Trade off Server Utilization and Tail Latency. USENIX Association, Singapore.

[62] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 66–85. https://doi.org/10.1145/3544216.3544265

[63] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient Queue Management for Cluster Scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 36, 15 pages. https://doi.org/10.1145/2901318.2901354

[64] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 351–364. https://doi.org/10.1145/2465351.2465386

[65] Khaldoun Senjab, Sohail Abbas, Naveed Ahmed, and Atta ur Rehman Khan. 2023. A survey of Kubernetes scheduling algorithms. *Journal of Cloud Computing* 12 (06 2023). https://doi.org/10.1186/s13677-023-00471-1

[66] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 183–197. https://doi.org/10.1145/2829988.2787508

[67] Apache Spark. 2024. A unified engine for large-scale data analytics. https://spark.apache.org/docs/latest/job-scheduling.html

[68] The Apache Software Foundation. 2022. The Hadoop Fair Scheduler. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[69] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[70] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. 2007. Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems* 18, 6 (2007), 789–803. https://doi.org/10.1109/TPDS.2007.70606

[71] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 35, 16 pages. https://doi.org/10.1145/2901318.2901355

[72] Oana-Mihaela Ungureanu, Cundefinedlin Vlundefineddeanu, and Robert Kooij. 2019. Kubernetes Cluster Optimization Using Hybrid Shared-State Scheduling Framework. In *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems* (Paris, France) *(ICFNDS '19)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/3341325.3341992

[73] Abhishek Verma, Madhukar Korupolu, and John Wilkes. 2014. Evaluating job packing in warehouse-scale computing. *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014* (11 2014), 48–56. https://doi.org/10.1109/CLUSTER.2014.6968735

[74] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European*

Conference on Computer Systems (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[75] Smita Vijayakumar. 2023. Mumuration's Prototype. https://github.com/csesmita/murmuration_prototype.

[76] Smita Vijayakumar. 2023. Mumuration's Simulator. https://github.com/csesmita/murmuration_simulator.

[77] Smita Vijayakumar. 2024. Mumuration's Helper Scripts. https://github.com/csesmita/kubernetes-helper.

[78] Da Wang, Gauri Joshi, and Gregory Wornell. 2015. Using Straggler Replication to Reduce Latency in Large-Scale Parallel Computing. *SIGMETRICS Perform. Eval. Rev.* 43, 3 (nov 2015), 7–11. https://doi.org/10.1145/2847220.2847223

[79] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. 2019. Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019.* ACM, 246–258. https://doi.org/10.1145/3357223.3362728

[80] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. 2011. Overdriver: handling memory overload in an oversubscribed cloud. *SIGPLAN Not.* 46, 7 (Mar 2011), 205–216. https://doi.org/10.1145/2007477.1952709

[81] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 265–278. https://doi.org/10.1145/1755913.1755940

[82] Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Chi Xu. 2020. Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 137, 12 pages.

[83] Wei Zhou, K. Preston White, and Hongfeng Yu. 2019. Improving Short Job Latency Performance in Hybrid Job Schedulers with Dice. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 56, 10 pages. https://doi.org/10.1145/3337821.3337851