Hawk: Hybrid Datacenter Scheduling

Pamela Delgado and Florin Dinu, École Polytechnique Fédérale de Lausanne (EPFL);
Anne-Marie Kermarrec, Inria; **Willy Zwaenepoel,**
École Polytechnique Fédérale de Lausanne (EPFL)

https://www.usenix.org/conference/atc15/technical-session/presentation/delgado

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Hawk: Hybrid Datacenter Scheduling

Pamela Delgado EPFL
Florin Dinu EPFL
Anne-Marie Kermarrec INRIA, Rennes
Willy Zwaenepoel EPFL

Abstract

This paper addresses the problem of efficient scheduling of large clusters under high load and heterogeneous workloads. A heterogeneous workload typically consists of many short jobs and a small number of large jobs that consume the bulk of the cluster's resources.

Recent work advocates distributed scheduling to overcome the limitations of centralized schedulers for large clusters with many competing jobs. Such distributed schedulers are inherently scalable, but may make poor scheduling decisions because of limited visibility into the overall resource usage in the cluster. In particular, we demonstrate that under high load, short jobs can fare poorly with such a distributed scheduler.

We propose instead a new hybrid centralized/distributed scheduler, called Hawk. In Hawk, long jobs are scheduled using a centralized scheduler, while short ones are scheduled in a fully distributed way. Moreover, a small portion of the cluster is reserved for the use of short jobs. In order to compensate for the occasional poor decisions made by the distributed scheduler, we propose a novel and efficient randomized work-stealing algorithm.

We evaluate Hawk using a trace-driven simulation and a prototype implementation in Spark. In particular, using a Google trace, we show that under high load, compared to the purely distributed Sparrow scheduler, Hawk improves the 50th and 90th percentile runtimes by 80% and 90% for short jobs and by 35% and 10% for long jobs, respectively. Measurements of a prototype implementation using Spark on a 100-node cluster confirm the results of the simulation.

1 Introduction

Large clusters have to deal with an increasing number of jobs, which can vary significantly in size and have very different requirements with respect to latency [4, 5]. Short jobs, due to their nature are latency sensitive, while longer jobs, such as graph analytics, can tolerate long latencies but suffer more from bad scheduling placement. Efficiently scheduling such heterogeneous workloads in a data center is therefore an increasingly important problem. At the same time, data center operators are seeking higher utilization of their servers to reduce capital expenditures and operational costs. A number of recent works [6, 7] have begun to address scheduling under high load. Obviously, scheduling in high load situations is harder, especially if the goal is to maintain good response times for short jobs.

The first-generation cluster schedulers, such as the one used in Hadoop [22], were centralized: all scheduling decisions were made in a single place. A centralized scheduler has near-perfect visibility into the utilization of each node and the demands in terms of jobs to be scheduled. In practice, however, the very large number of scheduling decisions and status reports from a large number of servers can overwhelm centralized schedulers, and in turn lead to long latencies before scheduling decisions are made. This latency is especially problematic for short jobs that are typically latency-bound, and for which any additional latency constitutes a serious degradation. For many of these reasons, there is a recent movement towards distributed schedulers [8, 14, 17]. The pros and cons of distributed schedulers are exactly the opposite of centralized ones: scheduling decisions can be made quickly, but by construction they rely on partial information and may therefore lead to inferior scheduling decisions.

In this paper, we propose Hawk, a hybrid scheduler, staking a middle ground between centralized and distributed schedulers. Attempting to achieve the best of both worlds, Hawk centralizes the scheduling of long jobs and schedules the short jobs in a distributed fashion. To compensate for the occasional poor choices made by distributed job scheduling, Hawk allows task stealing for short jobs. In addition, to prevent long jobs from monopolizing the cluster, Hawk reserves a (small) portion of the servers to run exclusively short jobs.

The rationale for our hybrid approach is as follows. First, the relatively small number of long jobs does not overwhelm a centralized scheduler. Hence, scheduling latencies remain modest, and even a moderate amount of scheduling latency does not significantly degrade the performance of long jobs, which are not latency-bound. Conversely, the large number of short jobs would overwhelm a centralized scheduler, and the scheduling latency added by a centralized scheduler would add to what is already a latency-bound job. Second, by scheduling long jobs centrally, and by the fact that these long jobs take up a large fraction of the cluster resources, the centralized scheduler has a good approximation of the occupancy of nodes in the cluster, even though it does not know where the large number of short jobs are scheduled. This accurate albeit imperfect knowledge allows the scheduler to make well-informed scheduling decisions for the long jobs. There is, of course, the question of where to draw the line between short and long jobs, but we found that benefits result for a large range of cutoff values.

The rationale for using randomized work stealing is based on the observation that, in a highly loaded cluster, choosing uniformly at random a loaded node from which to steal a task is very likely to succeed, while finding at random an idle node, as distributed schedulers attempt to do, is increasingly less likely to succeed as the slack in the cluster decreases.

We evaluate Hawk through trace-driven simulations with a Google trace [15] and workloads derived from [4, 5]. We compare our approach to a state-of-the-art fully distributed scheduler, namely Sparrow [14] and to a centralized one. Our experiments demonstrate that, in highly loaded clusters, Hawk significantly improves the performance of short jobs over Sparrow, while also improving or matching long job performance. Hawk is also competitive against the centralized scheduler.

Using the Google trace, we show that Hawk performs up to 80% better than Sparrow for the 50th percentile runtime for short jobs, and up to 90% for the 90th percentile. For long jobs, the improvements are up to 35% for the 50th percentile and up to 10% for the 90th percentile. The differences are most pronounced under high load but before saturation sets in. Under low load or overload, the results are similar to Sparrow. The results are similar for the other traces: Hawk sees the most improvements under high load, and in some cases the improvements are even higher than those seen for the Google trace.

We break down the benefits of the different components in Hawk. We show that both reserving a small part of the cluster and work stealing are essential to good performance for short jobs, with work stealing contributing the most to the overall improvement, especially for the 90th percentile runtimes. The centralized scheduler is a key component for obtaining good performance for the long jobs.

We implement Hawk as a scheduler plug-in for Spark [23], by augmenting the Sparrow plug-in with a centralized scheduler and work stealing. We evaluate the implementation on a cluster of 100 nodes, using a small sample of the Google trace. We demonstrate that the general trends seen in the simulation hold for the implementation.

In summary, in this paper we make the following contributions:

1. We propose a novel hybrid scheduler, Hawk, combining centralized and distributed schedulers, in which the centralized entity is responsible for scheduling long jobs, and short jobs are scheduled in a distributed fashion.
2. We introduce the notion of randomized task stealing as part of scheduling data-parallel jobs on large clusters to "rescue" short tasks queued behind long ones.
3. Using extensive simulations and implementation measurements we evaluate Hawk's benefits on a variety of workloads and parameter settings.

2 Motivation

2.1 Prevalent workload heterogeneity

Workload heterogeneity is the norm in current data centers [4, 15]. Typical workloads are dominated by short jobs. Long jobs are considerably fewer, but dominate in terms of resource usage. In this paper, we precisely address scheduling for such heterogeneous workloads.

To showcase the degree of heterogeneity in real workloads, we analyze the publicly available Google trace [1, 15]. We order the jobs by average task duration. The top 10% jobs account for 83.65% of the task-seconds (i.e., the product of the number of tasks and the average task duration). Moreover, they are responsible for 28% of the total number of tasks, and their average task duration is 7.34 times larger than the average task duration of the remaining 90% of jobs.

Workload	% Long Jobs	% Task-Seconds
Google 2011	10.00%	83.65%
Cloudera-b 2011	7.67%	99.65%
Cloudera-c 2011	5.02%	92.79%
Cloudera-d 2011	4.12%	89.72%
Facebook 2010	2.01%	99.79%
Yahoo 2011	9.41%	98.31%

Table 1: Long jobs in heterogeneous workloads form a small fraction of the total number of jobs, but use a large amount of resources.

We also analyzed additional workloads described in [4, 5]. Table 1 shows the percentage of long jobs among all jobs, and the percentage of task-seconds contributed by the long jobs. The same pattern emerges in all cases, even for different providers: the long jobs account for a disproportionate amount of resource usage.

The numbers we provided also corroborate previous findings from several other researchers [2, 16, 22].

2.2 High utilization in data centers

Understanding how to run data centers at high utilization is becoming increasingly important. Resource-efficiency reduces provisioning and operational costs as the same amount of work can be performed with fewer resources [12]. Moreover, data center operators need to be ready to maintain acceptable levels of performance even during peak request rates, which may overwhelm the data center.

Related work has approached the problem from the point of view of a single data center server [6, 7]. For a single server, the challenge is to maximize resource utilization by collocating workloads without the danger of decreased performance due to contention. As a result, several isolation and resource allocation mechanisms have been proposed, ensuring that resources on servers are well and safely utilized [19, 20].

Running highly utilized data centers presents additional, orthogonal challenges beyond a single server. The problem we are targeting consists of scheduling jobs to servers in a scalable fashion such that all resources in the cluster are efficiently used.

2.3 Challenges in performing distributed scheduling at high load

We next highlight by means of simulation why a heterogeneous workload in a loaded cluster is a challenge for a distributed scheduler. The main insight is that with few idle servers available at high load, distributed schedulers may not have enough information to match incoming jobs to the idle servers. As a result, unnecessary queueing will occur. The impact of the unnecessary queueing increases dramatically for heterogeneous workloads.

We illustrate this insight in more detail using the Sparrow scheduler, a state-of-the-art distributed cluster scheduler [14]. In Sparrow, each job has its own scheduler. To schedule a job with t tasks, the scheduler sends probes to $2t$ servers. When a probe comes to the head of the queue at a server, the server requests a task from the scheduler. If the scheduler has not given out the t tasks to other servers, it responds to the server with a task. This technique is called “batch probing”. More details can be found in the Sparrow paper [14], but the above suffices for our purposes. Sparrow is extremely scalable and efficient in lightly and moderately loaded clusters, but under high load, few servers are idle, and $2t$ probes are unlikely to find them. More probes could be sent, but the paper found that this is counterproductive because of messaging overhead.

We use the same simulator employed by the Sparrow paper [14] to investigate the following scenario: 1000 jobs need to be scheduled in a cluster of 15000 servers. 95% of the jobs are considered short. Each short job has 100 tasks, and each task takes 100s to complete. 5% of the jobs are long. Each has 1000 tasks, and each task takes 20000s. The job submission times are derived from a Poisson distribution with a mean of 50s. We measure the cluster utilization (i.e., percentage of used servers) every 100s. The median utilization is 86%, and the maximum is 97.8%. This suggests that at least 300 servers (2%) are free at any time, enough to accommodate all tasks of any incoming short job.

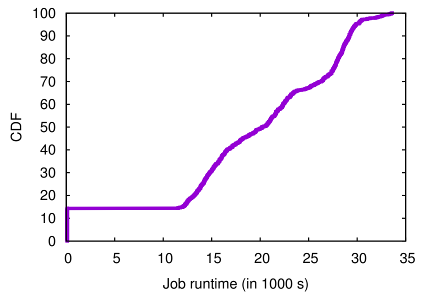


Figure 1: CDF of runtime for short jobs, in a loaded cluster, using Sparrow.

Figure 1 presents the cumulative frequency distribution (CDF) of the runtimes of short jobs. A large fraction of short jobs exhibit runtimes of more than 15000 seconds, far in excess of their execution time, which clearly indicates a large amount of queuing, mostly behind long jobs. Given that enough servers are free, an omniscient scheduler would yield job runtimes of 100s for the majority of the short jobs. With Sparrow, if all tasks are 100s long, the impact of queueing is less severe. However, a heterogeneous workload coupled with high cluster load has a strong negative impact on the performance of short jobs.

3 The Hawk Scheduler

3.1 System model

We consider a cluster composed of server (worker) nodes. A job is composed of a set of tasks that can run in parallel on different servers. Scheduling a job consists of assigning every task of that job to some server. We use the terms long task and short tasks to refer to tasks belonging to long jobs or short jobs respectively. A job completes only after all its tasks finish. Each server has one queue of tasks. When a new task is scheduled on a server that is already running a task, the task is added to the end of the queue. The server queue management policy is FIFO.

3.2 Hawk in a nutshell

The previous section demonstrated that *(i)* many cluster workloads consist of a short number of long jobs that take up the bulk of the resources and a large number of short jobs that take up only a small amount of the total resources, and *(ii)* existing distributed cluster scheduling systems, exemplified by Sparrow, do not provide good

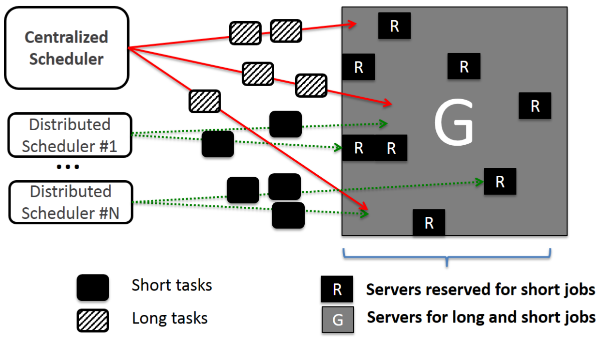


Figure 2: Overview of job scheduling in Hawk.

performance for short jobs in such an environment, due to head-of-line blocking.

In this context, Hawk's goals are:

1. to run the cluster at high utilization,
2. to improve performance for short jobs, which are the most penalized ones in highly loaded clusters,
3. to sustain or improve the performance for long jobs.

To meet these challenges, Hawk relies on the following mechanisms. To improve performance for short jobs, head-of-line blocking must be avoided. To this end, Hawk uses a combination of three techniques. First, it reserves a small part of the cluster for short jobs. In other words, short jobs can run anywhere in the cluster, but long jobs can only run on a (large) subset of the cluster. Second, to maintain low latency scheduling decisions, Hawk uses distributed scheduling of short jobs, similar to Sparrow. Third, Hawk uses randomized work stealing, allowing idle nodes to steal short tasks that are queued behind long tasks.

Finally, Hawk uses centralized scheduling for long jobs to maintain good performance for them, even in the face of reserving a part of the cluster for short jobs. The rationale for this choice is to obtain better scheduling decisions for long jobs. Since there are few long jobs, they do not overwhelm a centralized scheduler, and since they use a large fraction of the cluster resources, this centralized scheduler has an accurate view of the resource utilization at various nodes in the cluster, even if it does not know the location of the many short jobs. Figure 2 presents an overview of the Hawk scheduler.

3.3 Differentiating long and short jobs

The main idea behind Hawk is to process long jobs and short jobs differently. Two important questions are 1) how to compute a per-job runtime estimate, and 2) where to draw the line between the two categories.

Hawk uses an estimated task runtime for a job and computes it as the average task runtime for all the tasks in that job. This allows Hawk to easily classify jobs with variations in task runtime [13] without having to deal with per-task estimates. Moreover, the average task runtime is relatively robust in the face of a few outlier tasks.

Hawk compares the estimated task runtime against a cutoff (threshold). The value of the cutoff is based on statistics about past jobs because the relative proportion of short and long jobs in a cluster is expected to remain stable over time. Jobs for which the estimated task runtime is smaller than the cutoff are scheduled in a distributed fashion. This estimation-based approach is grounded in the fact that many jobs are recurring [9] and compute on similar input data. Thus, task runtimes from a previous execution of a job can inform a future run of the same job [9].

3.4 Splitting the cluster

Hawk reserves a portion of the servers to run exclusively short tasks. Long tasks are scheduled on the remaining (large) part. Short tasks may be scheduled on the whole set of servers. This allows short tasks to take advantage of any idle servers in the entire cluster. Henceforth we use the term *short partition* to refer to the set of servers reserved for short jobs and the term *general partition* to refer to the set of servers that can run both types of tasks.

If long tasks were scheduled on any server in the cluster, this may severely impact short jobs when short tasks end up queued after long tasks. A particularly detrimental case occurs when a long job has more tasks than servers or when several long jobs are being scheduled in rapid succession. In this case, every server in the cluster ends up executing a long task, and short tasks have no choice but to queue after them.

Hawk sizes the general partition based on the proportion of time that cluster resources are used by long jobs. For example, from Table 1 Hawk uses the percentage of task-seconds.

3.5 Scheduling short jobs

Hawk maintains low-latency scheduling for short jobs by relying on a distributed approach. Typically, each short job is scheduled by a different scheduler. For scalability reasons, these distributed schedulers have no knowledge of the current cluster state and do not interact with other schedulers or with the centralized component.

Distributed schedulers schedule tasks on the entire cluster. The first scheduling step is achieved as in Sparrow. To schedule a job with t tasks, a distributed scheduler sends probes to $2t$ servers. When a probe comes to the head of a server's queue, the server requests a task from the scheduler. If the scheduler has not given out the t tasks to other servers, it responds to the server with a task. Otherwise, a cancel is sent.

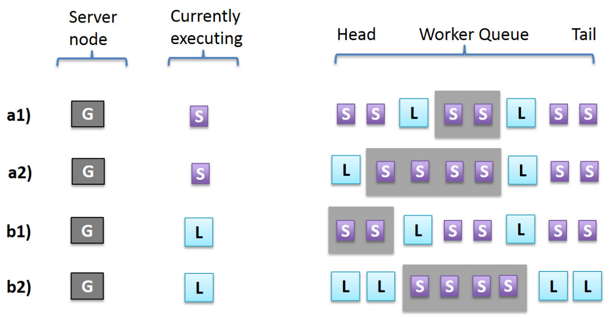


Figure 3: Task stealing in Hawk. L = Long task, S = Short task. Stolen tasks are on the dark background.

3.6 Randomized task stealing

Hawk uses task stealing as a run-time mechanism aimed at mitigating some of the delays caused by the occasionally suboptimal, distributed scheduling decisions. Since the distributed schedulers are not aware of the content of the server queues, they may end up scheduling short tasks behind long tasks. In a highly loaded cluster, the probability of this event happening is fairly high. Even if a short job is scheduled using twice as many probes as tasks, if more than half of the probes experience head-of-line blocking, then the completion time of the short job takes a big hit.

Hawk implements a randomized task stealing mechanism, that leverages the fact that the benefit of stealing arises in highly loaded clusters. In such a cluster a random selection very likely returns an overloaded server. Indeed, if 90% of the servers are overloaded, a uniform random probe has 90% probability of returning an overloaded server from which tasks are stolen.

The cluster might reach a point where many servers in the general partition are occupied by long tasks and also have short tasks in their queues, while other servers lie idle. Hawk allows such idle servers to steal tasks from the over-subscribed ones. This works as follows: whenever a server is out of tasks to execute, it randomly contacts a number of other servers to select one from which to steal short tasks. Both the servers from the general partition and the servers from the short partition can steal, but they can only steal from servers in the general partition, because that is where the head-of-line blocking is caused by long jobs.

Task stealing in Hawk proceeds as follow: The first consecutive group of short tasks that come after a long task is stolen. To see this in more detail, consider Figure 3. In cases a1) and a2) a server currently is executing a short job. The short tasks that it provides for stealing come after the first long job in the queue. In cases b1) and b2) the server is executing a long task. The short tasks stolen come immediately after that long task. Even though that long task is being executed already and has made some progress to completion, it is still likely that it will delay the short tasks queued behind it.

With our design we want to increase the chance that stealing actually leads not only to an improvement in task runtime but also in job runtime. Consider a job that has completed all but two of its tasks. Stealing just one of these tasks improves that task's runtime, but the job runtime is still determined by the completion time of the last task (the one not stolen). As shown in Figure 3, Hawk steals a limited number of tasks and starts from the head of the queue when deciding what to steal. Thus, stealing focuses on a few short jobs, increasing the chance that the runtime of those jobs benefits. If short tasks were stolen from random positions in server queues that would likely end up focusing on too many jobs at the same time while failing to improve most.

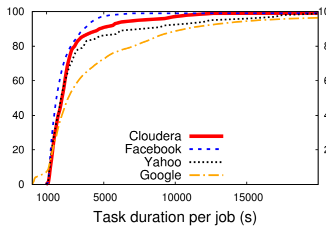
3.7 Scheduling long jobs

The final technique used in Hawk is to schedule long jobs in a centralized manner. Long jobs are only scheduled in the general partition, and the centralized component has no knowledge of where the short tasks are scheduled. This centralized approach ensures good performance for long jobs for three reasons. First, the number of long jobs is small, so the centralized component is unlikely to become a bottleneck. Second, long jobs are not latency-bound, so they are largely unaffected even if a moderate amount of scheduling latency occurs. Third, by scheduling long jobs centrally and by the fact that these long jobs take up a large fraction of the cluster resources, the centralized component has a timely and fairly accurate view of the per-node queueing times regardless of the presence of short tasks.

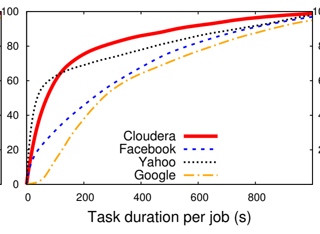
The centralized component keeps a priority queue of tuples of the form $< server, waiting\ time >$. The priority queue is kept sorted according to the waiting time. The waiting time is the sum of the estimated execution time for all long tasks in that server's queue plus the remaining estimated execution time of any long task that currently may be executing. When a new job is scheduled, for every task, the centralized allocation algorithm puts the task on the node that is at the head of the priority queue (the one with the smallest waiting time). After every task assignment, the priority queue is updated to reflect the waiting time increase caused by the job that is being scheduled. The goal of this algorithm is to minimize the job completion time for long jobs.

3.8 Implementation

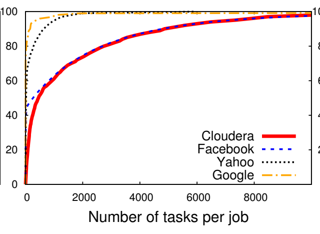
We implement Hawk as a scheduler plug-in for Spark [23], by augmenting the Sparrow scheduler with a centralized scheduler and work stealing. To realize work stealing we enable the Sparrow node monitors to com-



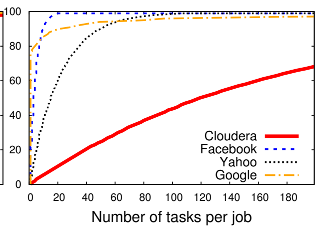
(a) Long jobs.



(b) Short jobs.



(c) Long jobs.



(d) Short jobs.

Figure 4: Workload properties. CDFs of average task duration and number of tasks per job.

municate and send tasks to each other. The node monitors communicate via the Thrift RPC library.

4 Evaluation

We compare Hawk with Sparrow, a state-of-the-art fully distributed scheduler. We show that in loaded clusters Hawk outperforms Sparrow for both long and short jobs. The benefits hold across all workloads. We also show that Hawk compares well to a centralized scheduler.

4.1 Methodology

Workloads We use the publicly available Google trace [1, 15]. After removing invalid or failed jobs and tasks we are left with 506460 jobs. Task durations vary within a given job. The estimated task execution time for a job is the average of its task durations.

We create additional traces using the description of the Cloudera C and Facebook 2010 workloads from [4] and Yahoo 2011 workload from [5]. We only consider the mapper tasks from these workloads, since many jobs do not have reducers. In [4, 5] the workloads are described as k-means clusters, and the first cluster is deemed composed of short jobs. We consider the rest of the clusters to be long jobs. For each cluster we derive the centroid values for the average number of tasks per job and the duration of the tasks by combining the information on task-seconds from [4, 5] with the job to mapper duration ratios in [22]. We then use the derived centroid values as the scale parameter in an exponential distribution in order to obtain the number of tasks and the mean task duration for each job. Given the mean task duration we derive task runtimes using a Gaussian distribution with standard deviation twice the mean, excluding negative values.

Figures 4a, 4b, 4c and 4d show the CDFs of the duration of tasks and the number of tasks per job for both long and short jobs. Table 2 shows additional trace properties. The trace properties differ from trace to trace. This is expected, as workload properties are known to vary depending on the provider [2, 4, 5].

Simulator We augment the event-based simulator used to evaluate Sparrow [14]. The input traces contain tuples of the form: (jobID, job submission time, number of tasks in the job, duration of each task). Network delay is assumed to be 0.5ms. The scheduling decisions and the task stealing do not incur additional costs.

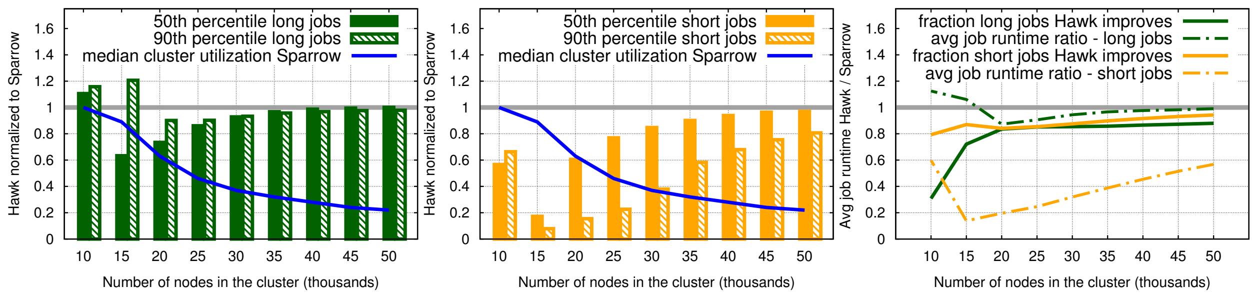
Workload	% Long Jobs	Total number jobs
Google 2011	10.00%	506460
Cloudera-c 2011	5.02%	21030
Facebook 2010	2.01%	1169184
Yahoo 2011	9.41%	24262

Table 2: Number of long jobs and total number of jobs.

Real cluster run We use a 100-node cluster with 1 centralized and 10 distributed schedulers. We use a subset of 3300 jobs from the Google trace. To obtain task runtimes proportional to the ones in the Google trace, we scale down task duration by 1000x (i.e., sec. to msec.) and use these durations in a sleep task. We also scale down the number of tasks per job by keeping constant the ratio between the cluster size and the largest number of tasks in a job. When we scale down the number of tasks in a job, we compensate by proportionally increasing the duration of the remaining tasks in order to keep the same task-seconds ratio as the original trace. We vary the cluster load by varying the mean job inter-arrival rate as a multiple of the mean task runtime. We use this mean to generate job inter-arrival times according to a Poisson distribution.

Parameters By default, in Hawk, a node performs task stealing by randomly contacting 10 other nodes and stealing from the first node that has short tasks eligible for stealing. We compare against Sparrow configured to send two probes per task because the authors of Sparrow [14] have found two to be the best probe ratio. Each simulated cluster node has 1 slot (i.e., can execute only one task at a time). This is analogous to having multi-slot nodes with each slot served by a different queue. Following the task-second proportion between long and short jobs, the short partition comprises 17% of the nodes for the Google trace and 9%, 2% and 2% for the Cloudera, Facebook and Yahoo traces, respectively.

Metrics When comparing Hawk to another approach X, we mostly take the ratio between the 50th (or



(a) Google trace long jobs. (b) Google trace short jobs. (c) Google trace additional metrics.

Figure 5: Google trace. Hawk normalized to Sparrow. Figure (c) shows two additional metrics: (1) percentage of jobs for which Hawk is equal or better to Sparrow and (2) average job runtime.

90th) percentile job runtime for Hawk and the 50th (or 90th) percentile job runtime time for X. Consequently, our results are normalized to 1. We do this separately for short and long jobs. Additional metrics are explained with the corresponding results. In all figures lower values are better.

Repeatability of results The results for the 50th and 90th percentiles are stable across multiple runs, and for this reason we do not show confidence intervals. We have seen variations in the maximum job runtime for short tasks. This is expected, as failing to steal one task can make a big difference in job runtime.

4.2 Overall results on the Google trace

We take the Google trace and vary the number of server nodes in order to vary cluster utilization. We find that Hawk consistently outperforms Sparrow, especially in a highly loaded cluster. Figures 5a and 5b illustrate the improvements in job runtime for long jobs and short jobs, respectively as a function of the number of machines in the cluster. The cluster utilization is based on snapshots taken every 100s.

Hawk shows significant improvements when the cluster is highly loaded but not overloaded (i.e., 15000 - 25000 nodes), since both the centralized scheduler and the task stealing algorithm make efficient use of any idle slots. In the best cases, Hawk improves the 50th and 90th percentile runtimes by 80% and 90% for short jobs and by 35% and 10% for long jobs. Hawk improves short job runtime at the 90th percentile more than at the 50th percentile, because these jobs are more affected by queueing. Stealing a few (even one) short tasks experiencing head-of-line blocking can greatly improve short job completion time.

Figure 5c presents additional metrics: the fraction of jobs for which Hawk provides performance better than or equal to Sparrow and the average job runtime for Hawk vs. Sparrow. The average job runtime for short jobs is significantly better for Hawk and is as low as a factor of 7. For 15000 nodes we present additional details, not all pictured: Hawk improves the runtime of 68% of short jobs, while for 59% of short jobs the improvement is more than 50%. Overall, for 86% of short jobs, Hawk is better or equal to Sparrow. For long jobs, Hawk improves 51% of jobs and is better or equal to Sparrow for 72% of jobs.

Small clusters (10000 nodes) tend to be overwhelmed by the high job submission rate in the trace. As a result, the node queues become progressively longer and waiting times keep increasing. We do not believe that any cluster should be run at this overload, but the case is nevertheless interesting to understand. Hawk is just slightly worse for long jobs, as the long jobs in Hawk are scheduled only in the general partition, while in Sparrow they can be scheduled across the entire cluster. Conversely, Hawk is better for short jobs because of the randomized stealing, but the improvement is small. The short partition is overloaded, and its nodes have few opportunities to steal short tasks experiencing head-of-line blocking in the general partition. As the cluster size increases (40000+ nodes), the benefits of Hawk decrease as the cluster becomes mostly idle. Any scheduler is likely to do well in that case.

4.3 Overall results on additional traces

Figures 6a, 6b and 6c show the results for the workloads derived from Facebook, Cloudera and Yahoo data. Hawk's benefits hold across all traces. At the median (not pictured), Hawk also improves on Sparrow across all simulated cluster sizes.

The most important difference compared to the Google trace is the larger improvement for short jobs. This can be traced back to the utilization of the short partition. In the Facebook, Cloudera and Yahoo traces the short partition is less utilized compared to the Google trace so there are more chances for stealing.

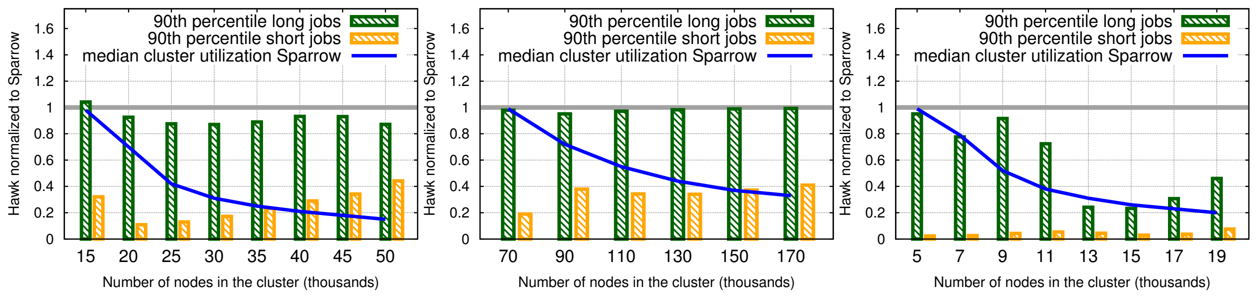

(a) Cloudera trace. (b) Facebook trace. (c) Yahoo trace.

Figure 6: Cloudera, Facebook and Yahoo traces. Long and short jobs. Hawk normalized to Sparrow.

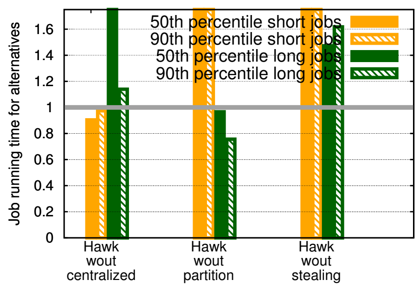


Figure 7: Break-down of Hawk's benefits normalized to Hawk. 15000 nodes. Google trace.

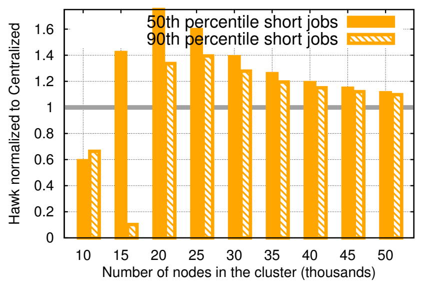


Figure 8: Hawk normalized to centralized approach, short jobs. Google trace.

4.4 Breaking down Hawk's benefits

This subsection analyzes the impact of each of the major components of Hawk: work stealing, reserving cluster space for short jobs and using centralized scheduling for the long jobs. We find that the absence of any of the components reduces the performance of Hawk for either long or short jobs.

Figure 7 shows the results of the Google trace normalized to Hawk with all components enabled. Without centralized scheduling for long jobs the performance of long jobs takes a significant hit, as tasks of different long jobs queue one after the other. The performance of short jobs improves due to the decrease in the performance for long jobs. As the placement of long jobs is not optimized in the general partition, fewer short tasks encounter queueing there.

Without partitioning the cluster, the short jobs are impacted, because they can be stuck behind long tasks on any node. For long jobs, the performance slightly increases, because they can be scheduled on more nodes. Without task stealing both short and long jobs suffer. The short jobs are greatly penalized, because some of their tasks are stuck behind long tasks. The long tasks are penalized, because they share the queues with more short tasks.

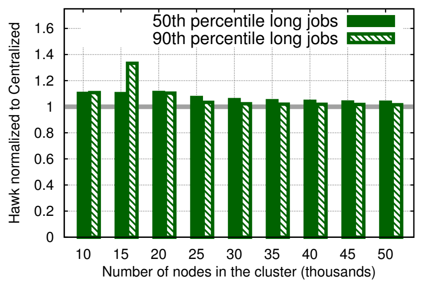


Figure 9: Hawk normalized to centralized approach, long jobs. Google trace.

4.5 Hawk vs. a fully centralized approach

We next look at the performance of Hawk compared to an approach that schedules all jobs (long and short) in a centralized manner. We find that Hawk is competitive, while not suffering from the scalability concerns that plague centralized schedulers.

This centralized scheduler does not reserve part of the cluster for short jobs and does not use work stealing. It uses the algorithm we presented in subsection 3.7 for all jobs. Figures 8 and 9 show Hawk normalized to the centralized scheduler's performance using the Google trace.

The centralized scheduler penalizes short jobs (Figure 8), when the cluster is heavily loaded (10000-15000 nodes). This is because in periods of overload the centralized scheduler does not have many options and

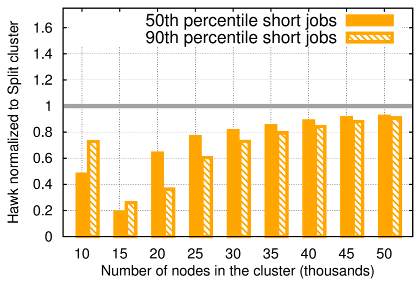


Figure 10: Hawk normalized to split cluster, short jobs. Google trace.

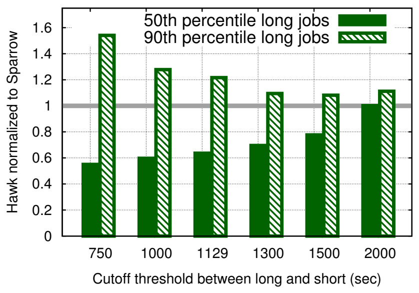


Figure 12: Effect of varying cutoff, Hawk normalized to Sparrow, long jobs. 15000 nodes. Google trace.

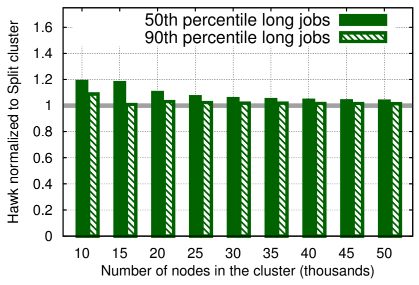


Figure 11: Hawk normalized to split cluster, long jobs. Google trace.

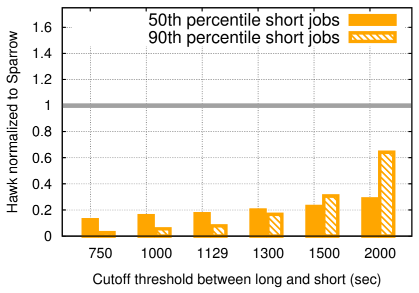


Figure 13: Effect of varying cutoff, Hawk normalized to Sparrow, short jobs. 15000 nodes. Google trace.

queues short tasks behind long ones. This is especially the case when long jobs are present in every node in the cluster. In Hawk short tasks benefit from stealing and from running on reserved nodes. As the cluster utilization decreases, the centralized scheduler does an increasingly better job for short jobs. When the cluster becomes lightly loaded (50000 nodes), the results for both approaches begin to converge.

For long jobs the centralized approach performs slightly better (Figure 9), because they can use the entire cluster. In Hawk they only use the general partition.

4.6 Hawk compared to a split cluster

We now compare Hawk to a split cluster, in which a long partition only runs long jobs and a short partition only runs short jobs. In other words, there is no general partition, in which both short and long jobs can execute. Hawk fares significantly better for short jobs, while being competitive for long jobs.

We use the Google trace. The split cluster uses 17% of the cluster for the short partition, and the remaining 83% is reserved for long jobs (long partition). The split cluster uses centralized scheduling for the long partition and distributed scheduling for the small one.

Figures 10 and 11 show the results. For long jobs, the split cluster performs slightly better, because the short jobs do not take up the space in the general partition. However, this comes at the cost of greatly increasing runtime for short jobs. For short jobs, for small cluster sizes, the relative degradation for the split cluster is smaller, because both approaches suffer from significant queueing delays. In the other extreme, for a large cluster, both approaches do well. In between, the split cluster shows extreme degradation, because short tasks cannot leverage the general partition nodes.

4.7 Sensitivity to the cutoff threshold

Next we vary the cutoff point between short and long jobs. Hawk yields benefits for a range of cutoff values, showing that it does not depend on the precise cutoff chosen.

The cluster size is 15000 nodes in this experiment, and we use the Google trace. Figures 12 and 13 show the results for long and short jobs, respectively. The percentage of short jobs increases as the cutoff increases. Thus, for the smaller cutoffs, Hawk improves the most on Sparrow because the short partition is underloaded and can steal more tasks. The percentage of long jobs increases as the cutoff decreases. For the smaller cutoffs the 90th percentile long job runtime is affected more for Hawk compared to Sparrow, because Sparrow is able to relieve some of the queueing among long jobs by scheduling them over the entire cluster.

4.8 Sensitivity to task runtime estimation

Hawk's centralized component schedules long jobs according to an estimate of the average task runtime for that job. We next analyze how inaccuracies in estimat-

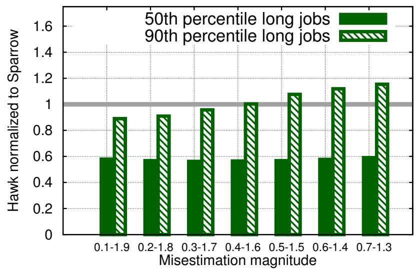


Figure 14: Hawk with varying mis-estimation magnitude normalized to Sparrow, long jobs. 15000 nodes. Google trace.

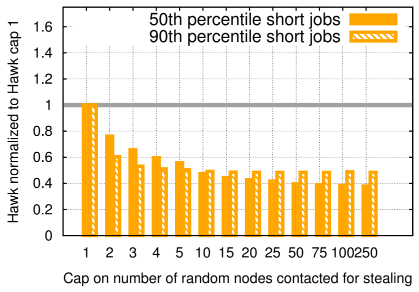


Figure 15: Hawk with varying number of stealing attempts normalized to Hawk capped at 1 attempt, short jobs. 15000 nodes. Google trace.

ing the average affect the results. For each job, to obtain the inaccurate estimate, we multiply the correct estimate with a random value, chosen uniformly within a range given as a parameter (e.g., 0.1-1.9). Figure 14 shows the job runtimes normalized to Sparrow for the set of jobs classified as long when no mis-estimations are present. These results are averaged over ten runs.

Hawk is robust to mis-estimations. The mis-estimation results in some long jobs being classified as short and vice-versa. This is more likely to happen for long and short jobs for which the estimation is comparable to the cutoff. Since these jobs are fairly similar in nature, the two opposing mis-classifications (long as short and short as long) tend to cancel each other. Moreover, most jobs are not mis-classified, because their estimation significantly differs compared to the cutoff. In Figure 14, long jobs perform better at the 90th percentile as the mis-estimation magnitude increases because more long jobs are classified as short. At 15000 nodes the short partition is less loaded than the general partition so the long jobs classified as short benefit from the additional, less-loaded nodes in the short partition.

Short jobs are not directly impacted by mis-estimations, since their scheduling does not rely on estimations. Short jobs can be indirectly impacted by the changes in the scheduling of the long jobs. In the experiments, we only see minute variations for the results for short jobs (not pictured).

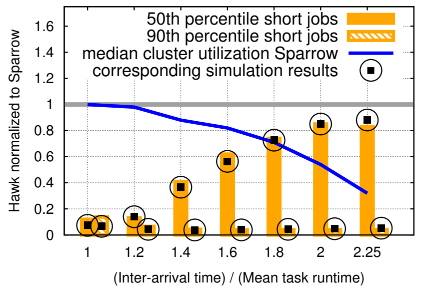


Figure 16: Implementation vs simulation, short jobs. 3300 job sample from the Google trace.

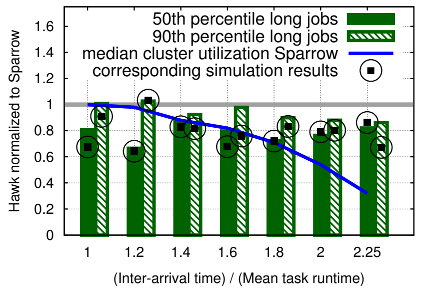


Figure 17: Implementation vs simulation, long jobs. 3300 job sample from the Google trace.

4.9 Sensitivity to stealing attempts

We now vary the maximum number of nodes that an idle node can contact for stealing. We find that performance increases with an increase in the cap value, but even a low value (e.g., 10) gives significant benefit.

Figure 15 shows the results normalized to Hawk using a cap of 1. As expected, increasing the cap also increases performance, as it increases the chance for successful stealing. At high cap values there is also a slight increase in the performance of long jobs (not pictured), because they wait behind fewer short tasks. The improvement for long jobs is small, because of the large relative difference between the resource usage of long jobs compared to short jobs.

4.10 Implementation vs. simulation

Figures 16 and 17 show the results for a 3300-job sample of the Google trace. In the implementation, Hawk schedules 3000 short jobs in a distributed way (300 per each of the 10 distributed schedulers) and 300 long jobs in a centralized fashion. The simulation and implementation experiments agree and show similar trends. Hawk is best at high loads, when it significantly improves on Sparrow for short jobs, while maintaining good performance for long jobs. As load decreases, the 50th percentiles for Hawk and Sparrow become similar, as fewer jobs suffer from queueing. Even at medium load, the 90th percentile is still considerably better for Hawk for short jobs, since those jobs suffer from queueing in Sparrow but not in Hawk.

The simulation and implementation results do not perfectly match, because the simulation does not model overheads for scheduling or stealing. Moreover, some Spark tasks sleep very little (a few msec) and are sensitive to slight inaccuracies in sleeping time and to various system overheads (message exchanges, network delays).

5 Related Work

The first data center schedulers had a monolithic design [22], which lead to scalability concerns [20]. Second generation schedulers (YARN [20], Mesos [10]) use a two-level architecture, which decouple resource allocation from application-specific logic such as task scheduling, speculative execution or failure handling. However, the two-level architecture relies on a centralized resource allocator, which can still become a scalability bottleneck in large clusters. In contrast, Hawk schedules most jobs in a distributed manner minimizing the scalability concerns.

We compared against Sparrow [14] in this paper. Sparrow is a fully distributed scheduler that performs well for lightly and medium loaded clusters. However, it is challenged in highly loaded clusters, especially for heterogeneous workloads, because tasks experience unnecessary queueing. This is due to Sparrow's design, which is geared at extreme scalability and cannot fully benefit from load information when making scheduling decisions. Moreover, Sparrow does not have runtime mechanisms to compensate in case the initial assignment of tasks to nodes is suboptimal.

In Apollo [3], distributed schedulers utilize global cluster information via a loosely coordinated mechanism. Apollo does not differentiate between long and short jobs and uses the same mechanisms to schedule both types of jobs. Apollo has built-in, node-level correction mechanisms to compensate for inaccurate scheduling decisions. If a task is queued longer than estimated at scheduling time, then Apollo starts duplicate copies of the task on other nodes. In contrast, work stealing in Hawk works at the level of the entire cluster. Even if the queueing time for a task has been correctly predicted, the task can be stolen by another server that becomes idle.

Mercury [11] is parallel work on designing a hybrid scheduler. In Mercury, jobs can choose between guaranteed (non-preemtable, non-queueable, centrally-allocated) containers and queueable containers (preemptable, allocated in a distributed way). However, it is not clear whether jobs have the information necessary to make an informed choice with respect to the appropriate container type. In Mercury, distributed schedulers loosely coordinate with a coordinator to obtain per-node load information. In Hawk, the distributed schedulers make completely independent decisions.

Omega [17] supports multiple concurrent schedulers which have full access to the entire cluster. The schedulers compete in a free-for-all manner, and use optimistic concurrency control to handle conflicts when they update the cluster state. Omega is designed to support at most tens of schedulers and this may prove insufficient to ensure low latency scheduling for very short jobs. Borg [21] uses a logically centralized controller but employs replication to improve availability and scalability. Borg's scheduling design is similar to Omega's optimistic concurrency control.

HPC and Grid schedulers [18] use centralized scheduling and do not have the same latency requirements. The jobs they schedule are usually compute-intensive and often long running. These jobs come with several constraints as they are tightly coupled in nature, requiring periodic message passing and barriers.

6 Conclusions and Future Work

In this paper we address the problem of efficient scheduling in the context of highly loaded clusters and heterogeneous workloads composed of a majority of short jobs and a minority of long jobs that use the bulk of the resources. We propose Hawk, a hybrid scheduling architecture. Hawk schedules only the long jobs in a centralized manner, while performing distributed scheduling for the short jobs. To compensate for the occasional poor choices made by distributed job scheduling, Hawk uses a novel randomized task stealing approach. With a Spark-based implementation and with large scale simulations using realistic workloads we show that Hawk outperforms Sparrow, a state-of-the-art fully distributed scheduler, especially in the challenging scenario of highly loaded clusters.

Acknowledgement

We thank the anonymous reviewers and our shepherd David Shue for their feedback. We also thank Laurent Bindschaedler, Christos Gkantsidis, Calin Iorgulescu, Konstantinos Karanasos, Sergey Legtchenko, Amitabha Roy, Malte Schwarzkopf and John Wilkes for the discussions, feedback and help. Additional thanks go to the Sparrow authors for making their simulator available. This research has been supported in part by a grant from Microsoft Research Cambridge.

References

[1] J. Wilkes - More Google cluster data. http://googleresearch.blogspot.ch/2011/11/more-google-cluster-data.html.

[2] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: coordinated memory caching for parallel jobs. In *Proc. NSDI 2012*.

[3] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proc. OSDI 2014*.

[4] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. In *Proc. VLDB 2012*.

[5] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. MASCOTS 2011*.

[6] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. ASPLOS 2013*.

[7] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proc. ASPLOS 2014*.

[8] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *Proc. SIGCOMM 2014*.

[9] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. EuroSys 2012*.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proc. NSDI 2011*.

[11] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. Usenix ATC 2015*.

[12] C. Kozyrakis. Resource efficient computing for warehouse-scale datacenters. In *Proc. DATE 2013*.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in mapreduce applications. In *Proc. OpenCirrus Summit 2011*.

[14] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proc. SOSP 2013*.

[15] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. SoCC 2012*.

[16] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. In *Proc. VLDB 2013*.

[17] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. EuroSys 2013*.

[18] G. Staples. Torque resource manager. In *Proc. SuperComputing 2006*.

[19] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: a software-defined storage architecture. In *Proc. SOSP 2013*.

[20] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. SOCC 2013*.

[21] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proc. EuroSys 2015*.

[22] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. EuroSys 2010*.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI 2012*.