Job Scheduling without Prior Information in Big Data Processing Systems

Zhiming Hu, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
Email: zhiming@ece.utoronto.ca, bli@ece.toronto.edu

Zheng Qin, Rick Siow Mong Goh
Institute of High Performance Computing
*A*STAR*
Singapore, Singapore
Email: {qinz, gohsm}@ihpc.a-star.edu.sg

***Abstract*—Job scheduling plays an important role in improving the overall system performance in big data processing frameworks. Simple job scheduling policies, such as Fair and FIFO scheduling, do not consider job sizes and may degrade the performance when jobs of varying sizes arrive. More elaborate job scheduling policies make the convenient assumption that jobs are recurring, and complete information about their sizes is available from their prior runs. In this paper, we design and implement an efficient and practical job scheduler for big data processing systems to achieve better performance even without prior information about job sizes. The superior performance of our job scheduler originates from the design of multiple level priority queues, where jobs are demoted to lower priority queues if the amount of service consumed so far reaches a certain threshold. In this case, jobs in need of a small amount of service can finish in the topmost several levels of queues, while jobs that need a large amount of service to complete are moved to lower priority queues to avoid head-of-line blocking. Our new job scheduler can effectively mimic the shortest job first scheduling policy without knowing the job sizes in advance. To demonstrate its performance, we have implemented our new job scheduler in YARN, a popular resource manager used by Hadoop/Spark, and validated its performance with both experiments on real datasets and large-scale trace-driven simulations. Our experimental and simulation results have strongly confirmed the effectiveness of our design: our new job scheduler can reduce the average job response time of the Fair scheduler by up to 45%.**

***Keywords*-job scheduling, big data processing, multilevel feedback queue**

I. Introduction

Ranging from recommendation systems to business intelligence, the use of big data processing frameworks, such as Apache Hadoop [1] and Spark [2], to run data analytics jobs that process large volumes of data has become routine in both academia and the industry. As a large number of jobs are submitted on a real-time basis, it is important to schedule them efficiently to improve their overall performance and the utilization of cluster resources.

To achieve these objectives, one of the important performance metrics that job scheduling policies are designed to optimize is the average *job response time*, defined as the time elapsed from when a job is submitted till when it is complete. From the perspective of the overall system, minimizing job response times gives priorities to smaller jobs, and thus relieving them from the head-of-line blocking problem caused by long running jobs. From the perspective of users, lower job response times help them to obtain their results faster.

Simple scheduling policies, such as first-in-first-out (FIFO) and Fair scheduler [3], do not consider job sizes at all and may suffer from long job response times in many cases. With FIFO, small jobs would be delayed if there exist large jobs ahead of them, a common situation in a shared cluster. With Fair scheduling, the scheduler is downgraded to *processor sharing* when multiple long-running jobs are submitted together, and its performance becomes much worse than scheduling jobs one by one. The moral of this story is, schedulers that are oblivious to job sizes may not provide the best possible average job response times.

There has been a wide variety of existing job scheduling policies that are proposed to reduce the average job response time by assuming that the complete information on job sizes is known *a priori* [4], [5], [6], mostly for recurring jobs. If this assumption is valid, shortest job first (SJF) or shortest remaining time first (SRTF) becomes good candidates. However, we argue that this assumption is neither practical nor valid in many cases. *First*, in a shared cluster, resources such as network and I/O bandwidth are shared among applications, which makes the running times different across multiple runs even for the same job. This situation becomes even worse if jobs span across geo-distributed data centers [7], [8], [9], where available capacities between data centers vary more significantly over time [10]. *Second*, even if we can perfectly isolate resources, it is still difficult to estimate the running times of non-recurring jobs with low overhead (Sec. II). Without accurate estimates of job sizes, existing algorithms may degrade the overall performance by a substantial margin. For example, if the size of a long running job is under-estimated, it may be placed ahead of other smaller jobs and delay all of them.

In this paper, we design and implement an efficient and practical job scheduler in data processing frameworks, without knowing the job sizes ahead of time. The highlight of our design is a multi-level job queue, which is used to separate short jobs from large jobs effectively and to mimic

the shortest job first scheduling policy without assuming known job sizes. Our new job scheduler is also able to avoid fine-grained sharing for jobs with similar sizes because these jobs will enter the same queue eventually and will be scheduled one by one in each queue. To further improve the performance, we also carefully design efficient ways to schedule jobs in the same queue and across different queues.

Our algorithm is partially motivated by the *least attained service* (LAS) policy [11]. By serving the job that has received the least service so far, LAS is a preemptive scheduling policy that favors small jobs without knowing the job sizes. The implicit assumption is that, if a job receives the least service, it is likely to be small, and should be granted a higher priority. This assumption works remarkably well if the job sizes follow a heavy-tailed distribution [11] by mimicking the SJF policy. However, if the job sizes are similar, LAS would be downgraded to *processor sharing*, and suffer from longer average response times. Our new algorithm is designed to avoid such pitfalls.

To be realistically deployed in big data processing systems, our design takes *practicality* as one of the important objectives. We focus on two major concerns. *First*, we should always keep an eye on the real-time resource demands of each job, as it most likely does not need resources across the entire cluster, which is different from flow scheduling in network switches. *Second*, we should avoid assigning resources to tasks in a job that cannot be started at the time of scheduling. For example, `reduce` tasks depend on the output of `map` tasks, and will only start after the `map` tasks complete[1]. It would be a waste of resources to schedule slots for `reduce` tasks too early.

Highlights of our original contributions in this paper are as follows. *First*, to the best of our knowledge, we are the first to address the problem of job scheduling without prior information in data-parallel frameworks, such as Apache Hadoop and Spark. *Second*, we propose a new strategy to obtain the amount of service that each job would receive in the current stage, which can identify large jobs more quickly. *Finally*, we introduce a new mechanism for scheduling jobs in each queue, which outperforms the widely used FIFO.

To demonstrate the performance of our new scheduler, we have implemented it in YARN, a popular open-source resource allocation framework for modern big data processing systems (*e.g.*, Hadoop and Spark). Both our experimental and trace-driven simulation results have strongly validated the effectiveness of our design. More specifically, our experimental results on real-world datasets have shown that the average job response time can be reduced by up to 45%, as compared to the Fair scheduler in YARN.

II. Motivation

Information about job sizes is critical for superior job scheduling performance, and most previous works assumed that such information is known or can be accurately estimated beforehand. However, as we have briefly discussed, information about job sizes may not be realistically available for the following practical reasons.

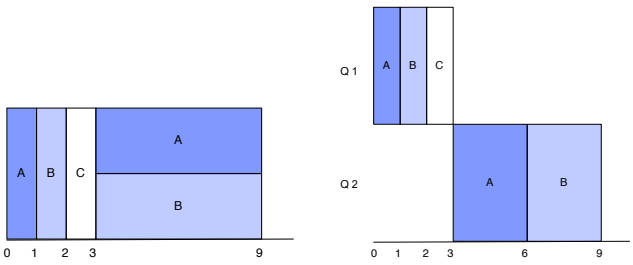
Many jobs are ad hoc jobs. It is shown that the percentage of recurring jobs is around 40% in the production workload at Microsoft [12], which also implies that more than half of the jobs are ad hoc jobs. Ad hoc jobs would only run once; Thus the programs or inputs are new, which makes it difficult to predict the running times of tasks or jobs. If we are not able to estimate the running time of the job before it is started, can we estimate the job size after it finishes a part of its tasks? Although it is conceivably feasible to estimate the completion times of tasks in the same stage if straggler tasks in each stage are properly handled, it is almost impossible to predict the completion times of tasks in future stages, as they have not yet started.

Data skews are common in each stage. For Hadoop/Spark jobs, the ideal case is that the running times for the tasks in the same stage are similar. However, this is not the case in the real workloads. In the `map` phase, even if the input sizes for each `map` tasks do not vary a lot, some records are just "larger" or "more expensive" than others [13]. For example, in graph processing applications, nodes with higher degrees are more computational and network intensive than other nodes. In the `reduce` phase, skews are more common because of the partition algorithms, where the intermediate results are distributed to the `reduce` tasks by hashing the keys and could be unevenly distributed. Therefore, in the `reduce` stage, different input sizes could be the main reason for skews, and it may also suffer from the type of skews as in the `map` phase even if the input sizes are the same [13].

Different stages have distinctly different running characteristics. It is hard to estimate the completion times of the tasks in later stages based on the completion times of completed stages because stages could be completely different regarding task types and the numbers of tasks [2]. For instance, in Hadoop, the `map` stage is entirely different from the `reduce` stage. In Spark, stages usually have completely different operations on their *Resilient Distributed Datasets* (RDDs). Thus, though we may know the completion times of stages that are finished, we still have no idea about the completion times of pending stages. Therefore, we are still not able to predict the completion time of the entire job before it is complete.

Even for recurring jobs, resource sharing and unstable networks can also cause unpredictable running times. Normally, the network and I/O bandwidth are resources that are shared across concurrent jobs in the same cluster. Moreover, in some cases, the amount of available bandwidth could vary with time significantly. For instance, in geo-distributed data analytics, the capacities between data centers vary quite significantly with time — as shown in [10], the 95th percentile value could be several times larger

[1]We do not consider stage overlaps in this paper.



(a) LAS. (b) LAS with 2 queues.

Figure 1. Multilevel queue for LAS: a motivating example.

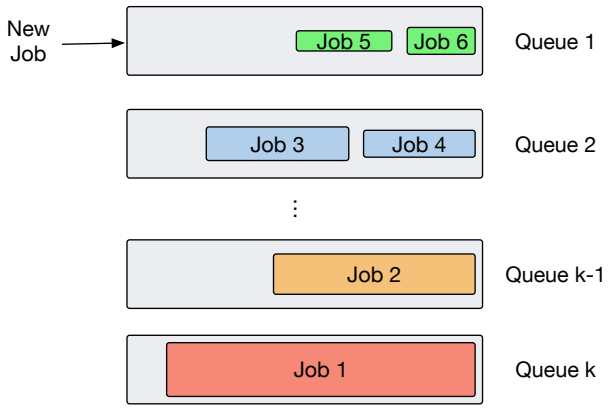


Figure 2. The design of LAS_MQ: an overview.

than 5th percentile value within 35 hours. In these cases, unfortunately, even recurring jobs may have unpredictable performance regarding job completion times.

A. A Motivation Example

As job size information is not known at the time of scheduling, we wish to schedule jobs without prior information about job sizes. Least attained service (LAS) is known as one of the best scheduling policies for this case, especially for jobs following heavy-tailed distributions [11]. Even though some workloads of big data processing systems do follow heavy-tailed distribution in the long run [12], we cannot directly apply LAS in the real systems. The most important reason is that workloads in the short term are dynamic and multiple large jobs may be in the system concurrently, which may greatly deteriorate the performance of LAS.

In this paper, we propose an approach based on multilevel queues. A motivating example is shown in Fig. 1. In this figure, there are three jobs (A, B, C), whose sizes are 4, 4, 1, respectively. Fig. 1(a) shows the scheduling results of LAS where A is admitted into the system first. At time 1, B arrives and A is thus preempted. A similar case also happens when C comes at time 2. At time 3, C is complete. A and B start to share the resources evenly as they have received the same amount of service at this time. From this figure, we can see that small jobs can still finish before large jobs through preemption (job C). However, when there are several large jobs, LAS is downgraded to *processor sharing*. Instead, if we design a multilevel queue to separate large jobs from small jobs and schedule these jobs one by one in each queue, we can mitigate this issue as we can see in Fig. 1(b). In this figure, there are two queues. A and B are demoted to the second queue after they used the cluster for one time slot. After C is complete, the first queue is now empty and we can start to schedule the jobs one by one in the second queue, which contains A and B. In the results, the response times of B and C are the same, while the response time of job A has been shortened from 9 to 6 (reduced by 33%).

III. System Design

We are now ready to present more details of our system design.

A. Design Overview

Referred to as LAS_MQ, our new scheduling policy is based on a multilevel queue. An overview of our design is shown in Fig. 2. There are k queues in our scheduler, and a job is demoted to a lower priority queue if the amount of service that it has received so far exceeds the threshold of the job's current queue. Thus, there would be $k-1$ thresholds for the first $k-1$ queues: α_1, $\alpha_2 \ldots \alpha_{k-1}$.

In this context, LAS_MQ works as follows. All the new jobs are submitted to the first queue (Queue 1), which has the highest priority. After that, jobs whose received service exceeds i-th threshold α_i ($i < k$) would be moved to the queue that has the threshold larger than the amount of received service. Those jobs, whose received service is larger than the threshold of $(k-1)$-th queue, would be placed to the last queue. With this strategy, small jobs can obtain enough service in the topmost several levels of queues and finish faster, while large jobs would be moved to lower priority queues to free up the resources for small jobs. Therefore, our design can effectively reduce the average job response time by mimicking a shortest job first (SJF) approach without knowing the job sizes beforehand.

Besides the rules of moving jobs across queues, we also need to decide the scheduling policies across queues and in each queue. Across queues, we adopt weighted sharing to avoid starvation in lower priority queues. This is because in our design, new jobs first go to the highest priority queue. If new jobs keep coming and we do not allocate resources to lower priority queues, it would cause job starvation in these queues. With weighted sharing among queues, small jobs can still finish very rapidly, while large jobs can also obtain a certain amount of resources and keep progressing. The scheduling policy in each queue is also very important

for the performance of our scheduler. We will present more details in Sec. III-C.

B. Obtaining the Amount of Service Received So Far

As we have stated previously, jobs are moved across queues based on the amount of service that they have received so far. In this section, we will discuss how we calculate the amount of service received by each job so far.

In YARN, resources are organized into containers and the completion times of jobs will be different if there are more (or less) containers allocated. Thus we cannot simply take the job completion time as the job size, and should also take the number of containers used into consideration. For this reason, in our case, if x containers are allocated to the job for t seconds, the amount of service that job j received j_s in that period is defined as follows:

$$j_s = x \cdot t. \tag{1}$$

When it comes to the amount of service received by the jobs so far, we would aggregate the products of duration and the number of containers in each scheduling period. For instance, if in the first round, the job is allocated 1 container for 5 units of time, and it is then allocated 2 containers for 3 units of time in the second round, the amount of service received by the job is $1 \cdot 5 + 2 \cdot 3 = 11$ container time. If in the scheduling period, the number of containers allocated to the job is 0, then the amount of service received by the job would not increase with time either.

However, as we have found out, we do not necessarily have to wait for the completion of the current stage to know the amount of service that a job would receive in this stage. We can *estimate* the amount of service that the job would receive in the stage, using the received amount of service so far in this stage divided by the progress of the stage. This method is also referred to as *stage awareness* in this paper.

There are several reasons for this strategy. First, if we can identify large jobs more quickly and thus move these jobs to lower priority queues faster, we can free up the resources sooner for small jobs instead of waiting for the threshold of the queue to be reached. Second, stage progresses, which indicate the percentage of data that has been processed so far in the stage, are available for both Hadoop and Spark, and can be used for estimating the amount of service that the job would receive in that stage. This strategy assumes that the progress rate of the stage is stable. In realistic cases, the progress rate of one stage may become faster in applications like Hadoop or Spark [14]. Therefore, we would sometimes over-estimate the amount of service that the job would receive in the stage.

The good news is, over-estimates have little impact on the scheduling results compared with under-estimates, because over-estimates only affect the completion time of the job itself [15]. The intuitive reasons are as follows. For shortest job first like strategies, if we under-estimate the job size, we may give it higher priority than it should have, which will delay a lot of jobs with smaller job sizes than this job. However, if we over-estimate the job size, the job is just scheduled in the later part of the job queue, which would mostly affect the response time of the job itself.

In sum, *stage awareness* works as follows. For example, in the `map` stage of Hadoop, if the stage has received 10 container time, and the stage progress is 10%, the estimated amount of service that the job would receive in the `map` stage is $\frac{10}{10\%} = 100$ container time. Using this approach, we can move the jobs to proper queues more quickly.

The amount of service received so far can be calculated by adding the amount of service that the job would receive in the current stage (estimated value) and the amount of service received in previous stages (precise value) together, which is then used to decide which queue the job would be moved to. Later in this paper, we will show that stage awareness is remarkably effective for improving the performance.

C. Job Ordering in Each Queue

After we know how jobs are moved across queues, we also need to decide the scheduling policy in each queue. For this issue, the most important requirement for our design is that the ordering in each queue should not change frequently, which may cause fine-grained sharing as what LAS does. Thus FIFO is a good start for this task, however we can do better by incorporating application-specific characteristics.

We propose to order the jobs by the number of containers that would be used by the remaining tasks of the job including running tasks. There are two reasons for this strategy. First, it is similar to FIFO and schedules jobs one by one. The order would not change too much because the number of remaining tasks of those jobs at the front of the queue would become smaller, and those jobs will remain at the front of the queue. Second, as the amount of resources allocated to each queue is fixed, assigning resources to jobs with smaller resource requirements can allow more jobs to finish their remaining tasks faster.

To demonstrate the benefits of stage awareness and job ordering in each queue, we compare the performance of different cases with Fair scheduling and show the results in Fig. 3. We use 100 Hadoop jobs in this case and the experiments are conducted multiple times. The job arrivals follow *Poisson distribution* and the mean interval of job arrival is 50 seconds. The normalized average job response time is calculated by dividing the average job response time of Fair scheduler by the average job response times of our algorithms. In this figure, we can see that, without these two features, the scheduler only slightly outperforms the Fair scheduler as seen in Case 1. With stage awareness, we can improve the performance by around 10% in the best case, as we can see in Case 2. Moreover, we can increase the performance by a wide margin with our job ordering strategy in each queue as shown in Case 3. If we only examine Case

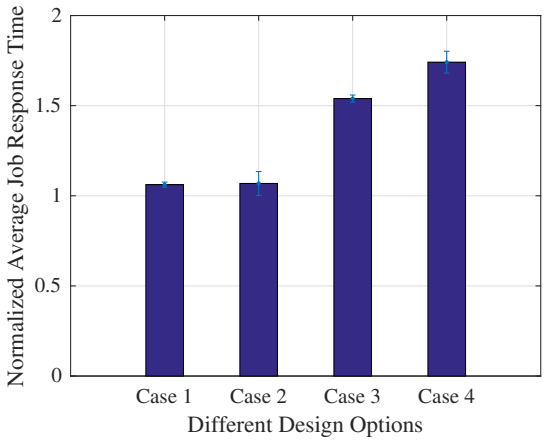


Figure 3. Case 1 represents the traditional design without either feature. Case 2 shows the result with stage-awareness only. Case 3 shows the result only with job ordering in each queue. Case 4 shows the result with both features. All the results are normalized over the Fair scheduler in YARN.

1 and Case 2, it seems that stage awareness only provides marginal improvements. However, in Case 4, we can see that stage awareness can further improve the performance compared with Case 3. Case 4 is also our current design with both of these features.

D. Dealing with Dependencies among Tasks

We have discussed how jobs are moved across queues and in each queue, but have not yet addressed the amount of resources we should allocate to the job that would be scheduled next.

There are two factors that need to be considered for scheduling jobs. First, most jobs do not need to be allocated all the resources in the entire cluster. Thus, we need to focus on the real demand of the job when it is to be scheduled. Second, jobs have dependencies among stages, therefore we should not allocate resources to tasks that are not ready. For instance, Hadoop jobs have dependencies among `map` tasks and `reduce` tasks. We would allocate resources to `map` tasks first and only allocate resources to `reduce` tasks after the `map` stage is complete.

Based on these principles, for Hadoop/Spark jobs, we calculate the demand and allocate the resources to the jobs stage by stage. We only consider the number of remaining tasks in the current stage when calculating its demand. After that, we assign the number of containers according to the number of remaining tasks to the job if there is a sufficient number of containers available.

E. Number of Queues and Thresholds of Queues

As our scheduler employs multiple queues, we need to determine the optimal number of queues k and the thresholds of queues, while ensuring that the average job response time is minimized.

If strict priority is enforced among queues, which means that jobs in the i-th queue can only start after all the jobs from the first queue to the $(i-1)$-th queue are complete, and FIFO is used in each queue, a theoretical way to derive the optimal thresholds and the number of queues is proposed in [16]. However, we propose to adopt weighted fair sharing among queues, with specific ordering in each queue. The methods in [16] cannot be directly applied in our case. Instead, we propose a simpler approach as stated in [17] where the thresholds for different queues increase exponentially. The reason for the exponentially increased thresholds of queues is because we can separate the jobs better with a much smaller number of queues if the job sizes follow heavy-tailed distribution compared with linearly increased thresholds. In other words, if the size of the largest job is s, then the number of queues $k = \lceil log(s) \rceil$. If the threshold of the first queue α_0 and the step p are decided, the threshold of each queue can then be computed using the formula $\alpha_{i+1} = p \cdot \alpha_i$.

In realistic cases, our algorithm works very well in a variety of settings. In our experiments, we simply set the number of queues as 10 and the threshold of the first queue as 100. We will also show the performance of our approach with a varying number of queues and different thresholds for the first queue in our evaluations.

Algorithm 1: Update Job Orders

```
for i = 1 to k do
    The set of jobs in i-th queue is denoted by set Q_i;
    for job j ∈ Q_i do
        Update the amount of service j_m that job j
          received so far, optimized by stage awareness;
        if j_m > α_i then
            Delete the job from Q_i;
            Add the job to the queue that has larger
              threshold than j_m;
        end
    end
    Sort jobs in Q_i according to the number of
      containers that would be used by their remaining
      tasks.
end
```

F. Summary

Overall, our scheduling algorithm works as follows. First, new jobs would be admitted to the end of the first queue, which is also the highest priority queue. After that, for new events like completions of map/reduce tasks or jobs, we would update the amount of service received by jobs and move the jobs across queues if necessary. During the update, we would also sort the jobs in each queue by the number of containers that are needed for the remaining tasks. The

Algorithm 2: Job Scheduling

Allocate all the containers to the queues according to weights of queues. The number of containers allocated to the i-th queue is denoted by r_i;
The number of containers needed by the remaining tasks of job j in the current stage is represented by j_{rt};
for $i = 1$ **to** k **do**
  for *job* $j \in \mathcal{Q}_i$ **do**
    if $r_i > 0$ **then**
      Allocate $x = \min(r_i, j_{rt})$ slots to the job j;
      $r_i = r_i - x$;
    else
      break;
    end
  end
end
Share the remaining containers to all the jobs if there is any.

algorithm of updating job orders is shown in **Algorithm 1**. The complexity of the algorithm is $\mathrm{O}(kn \log n)$ if the total number of jobs is n.

Second, in the scheduling part, we retrieve the number of ready tasks of the next job in the first non-empty queue and allocate the number of containers according to the number of ready tasks. After that, if there are any remaining resources, we would allocate the resources to other jobs in that queue.

Third, our scheduling policy also meets the requirements for work conservation. After we finish assigning containers to all the running jobs, if there are remaining containers, we would assign those containers evenly to all the running jobs. One of the advantages for this is that each job can have more resources than it needs, and it can then launch a few speculative tasks that may further improve the performance. Our job scheduling algorithm is shown in **Algorithm 2**. The complexity of the algorithm is $\mathrm{O}(kn)$.

IV. Implementation

We have implemented our algorithm as a plug-in scheduler in YARN, in the context of Apache Hadoop 2.4.0. The overall design of our implementation is shown in Fig. 4. In our overall design, we can see that when a new job is submitted to the system, it would first go through the job admission module. In our case, we only control the total number of running jobs because too many running jobs may cause hanging. If the total number of running jobs is smaller than the limit, then the job is admitted and queued for its share of resources.

The capacity scheduler [18] can change the capacities of queues by updating the configuration file on a real-time basis. In our implementation, each application is assigned

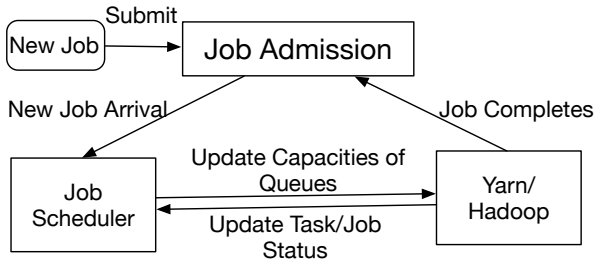


Figure 4. Implementing the LAS_MQ scheduler: overall design.

to a unique queue. Thus, we can control the amount of resources for each application by setting the capacities of queues. After scheduling, if the number of containers allocated to the application is above zero, the new job will be submitted to the cluster and start running. During the lifetime of the job, we keep monitoring the job's running status such as task completion events and stage progresses, which are part of the inputs for our scheduler. When a job completes, the job admission module will be notified, and it will check whether all the jobs have completed. If not, it will submit more jobs to the scheduler.

In the scheduler, the unit of allocation is a container with one *vcore* and 2 GB of main memory. In our case, the total number of containers in the cluster is fixed and bounded by the total amount of memory. The scheduling problem now becomes how to place the jobs to those containers. We keep track of the number of remaining tasks in the current stage of the job during the scheduling process to assign a proper number of containers to it.

Obtaining the number of remaining tasks in the current stage of the job is thus very important. In our implementation, we calculate the remaining number of tasks by using the total number of tasks minus the number of successfully completed tasks. Therefore, the first problem is how to obtain the total number of map/reduce tasks for a job. For Hadoop jobs, we can set the number of map/reduce tasks, however, the real total number of tasks could be different from the values in the configurations. We solve this issue by examining the number of splits of the inputs after the job is submitted to the cluster, looking for the total number of maps and reduces. The second issue is how we get to know the number of successful map tasks or reduce tasks. To the best of our knowledge, there are no available counters that we can directly use for this piece of information. We propose to store all the incoming task events of the job, filter out those unsuccessfully finished tasks and count the number of successful tasks accordingly.

When a job is to be scheduled, we check the number of remaining tasks first and allocate the calculated number of containers or all the remaining containers, whichever is smaller. Here, the calculated number of containers might be different from the number of remaining tasks, because

we usually allocate two containers for each reduce task as reduce tasks need more memory than map tasks.

V. Performance Evaluation

In this section, we present the experimental and simulation results of our scheduler.

A. Experimental and Simulation Setup

Testbed: Our testbed consists of 4 workstations, each with 120 GB of main memory and 56 Intel Xeon CPU E5-2683 v3 @ 2.00 GHz. The capacities between those workstations are 10 Gbps.

In this testbed, we have four slave nodes, one of which is also the master node of YARN and Hadoop Distributed File System (HDFS) [1]. The size of main memory allocated for each *NodeManager* is 60 GB in each node and the sizes of memory for map tasks and reduce tasks are 2 GB and 4 GB, respectively. Thus we can start up to 120 containers at the same time. In HDFS, the block size is 128 MB and the replication factor is 2.

Workload and Inputs: Our workload contains 100 jobs and each job is randomly selected from *TeraGen*, *SelfJoin*, *Classification*, *HistogramMovies*, *HistogramRatings*, *SequenceCount*, *InvertedIndex* and *WordCount*, all of which are popular jobs for benchmarking Hadoop. The job arrivals follow the *Poisson distribution*. We have tried different intervals for job arrivals, which will be stated in our experimental results.

The inputs are from the PUMA datasets [19]. For *SelfJoin* we use synthetic data and we use real datasets for all the other jobs. More specifically, for *SequenceCount*, *InvertedIndex* and *WordCount*, the inputs are from Wikipedia datasets. In *Classification*, *HistogramMovies*, *HistogramRatings*, the movie dataset is used. The workload is divided into four bins according to the sizes of inputs.[2] More details about the workloads and inputs are available in Table I.

Simulator: We have implemented an event-driven simulator for the simulations and we use both workloads following heavy-tailed and uniform distribution to evaluate the performance of our scheduler. For the heavy-tailed case, the trace was collected from a Facebook cluster in 2010 [20], which consists of 24,443 jobs. We calculate the job sizes by summing up the amount of data processed by each job including input data, intermediate data and output data. The job sizes are further normalized based on the loads of the system. In this case, the load is set to be 0.9. The final job sizes follow the heavy-tailed distribution. For the case of light-tailed distribution, we generate 10,000 jobs, all with the size of 10,000.

Baselines: We compare our approach, LAS_MQ with the LAS, FAIR and FIFO scheduler. For the Fair scheduler, it allocates resources to jobs according to the priorities of jobs. In our workload, the priorities of jobs are randomly generated integers ranging from 1 to 5.

Metrics: The most important metric is the average job response time. We also measured *slowdown*, which is defined as the job response time divided by the time it takes to finish when the job is scheduled to the cluster alone. The slowdown is widely used to evaluate the fairness of scheduling algorithms.

We also use the notion of the *Normalized Average Job Response Time*, defined as follows:

$$\text{Normalized Resp. Time} = \frac{\text{Result of Fair Scheduling}}{\text{Result of Our Algorithm}}$$

Table I
THE WORKLOAD USED IN THE EXPERIMENTS.

Bin	Job Name	Dataset Size	# of maps	# of reduces	# of jobs
1	TeraGen	1 GB	100	10	3
1	SelfJoin	1 GB	102	10	15
2	Classification	10 GB	102	20	17
2	HistogramMovies	10 GB	102	20	12
2	HistogramRatings	10 GB	102	20	8
3	SequenceCount	30 GB	234	60	16
3	InvertedIndex	30 GB	234	60	19
4	WordCount	100 GB	721	80	10

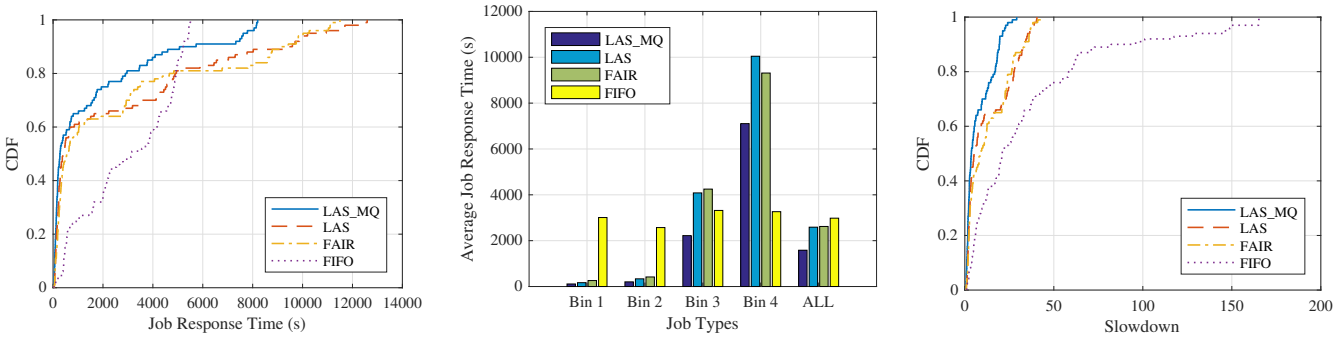
B. Experimental Results

We aim to answer the following questions in our experiments. (1) What is the performance of our approach regarding average job response times and fairness? (2) How does the performance vary with different loads?

1) Average Job Response Times: In Fig. 5, the mean interval of job arrivals is set to be 80 seconds. The number of queues and step are 10. The threshold of the first queue is 100. In Fig. 5(a), we can see that our solution outperforms LAS, Fair scheduler and FIFO scheduler for job response times. FIFO has the worst performance among the four algorithms. The reason is that small jobs can be severely delayed by large jobs in FIFO. LAS and Fair scheduling have similar performance. For these two schedulers, small jobs can also get their shares. Thus small jobs finish rapidly, however, large jobs will be delayed for fine-grained sharing. To better illustrate the performance for jobs with different input sizes, we divide the workload into four bins according to the sizes of inputs and we can see the performance of different bins in Fig. 5(b).

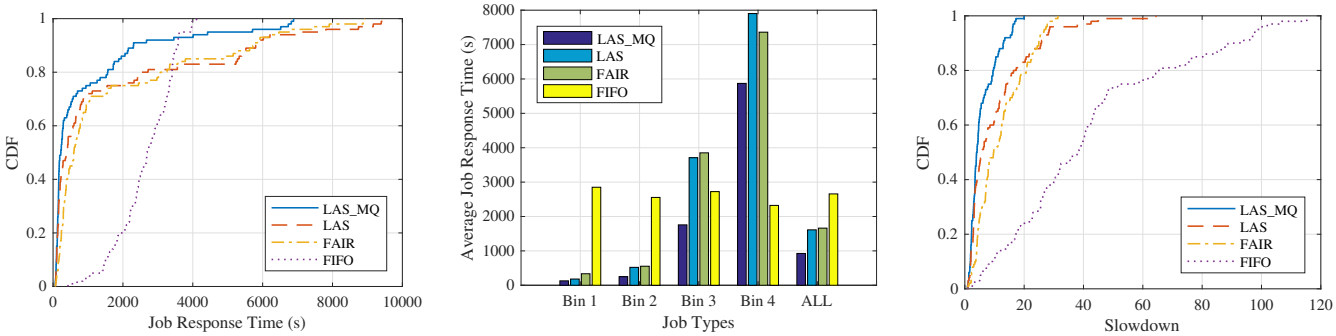
In Fig. 5(b), we can see that our approach outperforms LAS and the Fair scheduler in all cases, and FIFO except Bin 4. In the average job response time of all the jobs, we can reduce the average job response time of LAS and Fair scheduler by nearly 40%, and the average job response time of FIFO by 46%. There are several highlights in our results. First, our solution can effectively find out and give higher priorities to small jobs, which can be seen in Bin

[2]For TeraGen, we do not need to define the input size and we set the output size instead.



(a) The CDF of job response times (b) The average job response time in different bins. (c) The slowdown of all the jobs.

Figure 5. The performance of the workload with the mean arrival interval of 80 seconds.



(a) The CDF of job response times (b) The average job response time in different bins. (c) The slowdown of all the jobs.

Figure 6. The performance of the workload with the mean arrival interval of 50 seconds.

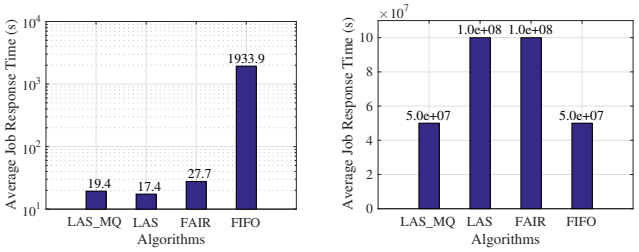
1-3. Second, a small portion of jobs in FIFO has better performance than ours as shown in Fig. 5(a), because of large jobs as we can see in Bin 4. This is because for large jobs in FIFO, it would be executed once the jobs before it are finished. However, in our case, we would give priorities to newly arrived smaller jobs instead and run large jobs in the final stage of the workload. Thus in our scheduling policy, large jobs would free up the resources for smaller jobs and wait longer to obtain the resources. Third, FIFO has similar average job response times in all the bins, which is because no matter the job is large or not, it would wait for the completion of 29 jobs before it when the maximum number of running jobs is 30 in the job admission module. Finally, LAS and Fair scheduler have nearly the same performance. They have good performance for small jobs while suffering from fine-grained sharing for large jobs.

2) Fairness: Even though fairness is not the main goal of our design, we also show the results as fairness is a very important metric for job scheduling. We show our results of the slowdown in Fig. 5(c), and we can see that again our solution has the smallest slowdowns followed by LAS, Fair scheduler and FIFO. The poor performance of FIFO is because of the seriously delayed small jobs. While for LAS and Fair scheduling, the bottlenecks are in large jobs as also indicated in Fig. 5(a) and Fig. 5(b). Instead, our solution achieves a good balance between small jobs and large jobs, thanks to our design using a multilevel queue and weighted fair sharing among queues.

3) Performance with Different Loads: The load is also an important factor for the performance of scheduling. Thus we change the mean arrival interval of the workload from 80 seconds in Fig. 5 to 50 seconds in Fig. 6. In this figure, we can see that the performance is better than the case of 80 seconds. More specifically, in all three figures, the performance gaps between our approach and baselines are larger. In Fig. 6(a), for our solution, around 90% of jobs have the average job response time of less than 2000 seconds while the corresponding values for LAS and the Fair scheduler are only around 70%. In Fig. 6(b), we reduce the average job completion time of LAS and Fair scheduling by around 45% and the one of FIFO by 65%. In Fig. 6(c), the slowdowns of our algorithm are also much smaller. All the figures show that our approach works even better for higher system loads. The explanation is that if the load is higher, there would be more running jobs in the system, which makes job scheduling more important.

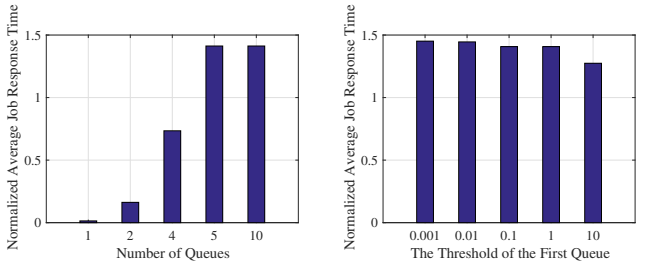
C. Simulation Results

With our simulations, we would like to investigate the performance of our scheduler with different distributions



(a) Heavy-tailed distribution (b) Uniform distribution

Figure 7. The average job response time of algorithms over different distributions.



(a) The threshold of the first queue is 1 and step is 10.

(b) The mumber of queues and step are 10.

Figure 8. The performance of our algorithm with different parameters. Improvements are over Fair Scheduling. (a) The number of queues. (b) The threshold of the first queue.

and the sensitivity of our approach to different parameter settings.

1) Workloads following Different Distributions: We compare the performance of algorithms for workloads under both heavy-tailed and light-tailed distributions. In these two cases, for LAS_MQ, the number of queues and steps are 10. The threshold of the first queue is 1. The results are shown in Fig. 7. In Fig. 7(a), we can see that, for the heavy-tailed distribution, the performance of LAS is the best, followed by LAS_MQ and Fair scheduling. LAS_MQ performs slightly worse than LAS, but it still outperforms Fair scheduling and reduces the average job completion time by around 30%. FIFO is much worse than the other three algorithms because small jobs can be severely delayed by large jobs. The reason for the good performance of our approach is that we can effectively separate large jobs from small jobs with the multilevel queue, thus we can obtain similar performance with LAS.

In the case of uniform distribution, FIFO and LAS_MQ have the best performance, and the average job response time is only half the ones of Fair scheduling and LAS. Because in this case, Fair scheduling and LAS would both be downgraded to *processor sharing*. The good performance of LAS_MQ is because that jobs with similar sizes will eventually go to the same queue, and we schedule these jobs one by one to avoid fine-grained sharing.

To summarize, we can see that LAS_MQ can provide stable and good performance for different workloads. In other words, our approach is quite adaptive, which is of particular importance in the case of cloud computing where the workloads in the future are dynamic and difficult to predict.

2) Different Number of Queues and Thresholds: We show the performance of our scheduler with different numbers of queues and different thresholds of the first queue in Fig. 8. In this figure, we can see that our scheduler with more queues can achieve better performance, and our scheduler outperforms the Fair scheduler if the number of queues is five or higher. In fact, it already obtains the best result when the number of queues is five because there are no jobs with sizes larger than the threshold of the fifth queue. With more queues, we can separate large jobs from small ones more effectively. In our system, adding more queues only has limited overhead because the movements of jobs only happen at the software level. In Fig. 8(b), we investigate the performance with different thresholds of the first queue. We can see that the performance is good for a variety of values. When the threshold is set to be 10, the performance is going down because the mean normalized size of jobs in the trace is around 20, which makes most of the jobs stay in the first queue until they complete. Therefore, it does not effectively separate large jobs from smaller ones in this case, which is the main reason for the deterioration of its performance. We can avoid this problem by using a relatively small number as the threshold for the first queue.

VI. Related Work

Scheduling without prior information. Multilevel feedback queue (MLFQ) [21] was a scheduling algorithm that prefers small jobs and I/O bound processes. It also does not require prior information about job sizes. Bai *et al.* [16], [22] implemented a MLFQ in commodity switches for flow scheduling in data center networks. Least attained service (LAS) [11] was also popular for job scheduling without prior information, yet it may suffer from performance degradations when several large jobs arrive. To resolve this issue, Discrete-LAS (DLAS) was introduced in [17] for coflow scheduling.

While they focused on similar problems, our approach in this paper differs significantly in the following ways. First, our approach focuses on job scheduling for big data processing systems, which is different from scheduling flows in switches. In our case, resources are organized into containers and tasks have dependencies. Second, we utilize more information available, such as stage progresses, and propose a practical way to calculate the amount of service that the job would receive in each stage, which can separate the large jobs from small jobs more quickly. Third, we propose to schedule the jobs in each queue based on the amount of resources required by the remaining tasks of jobs, which

can also significantly improve the performance.

Job scheduling. For non-network aware approaches, Hopper [23] took speculation into consideration as well and aimed to combine speculative mechanisms with job scheduling to improve the performance. Hung *et al.* proposed solutions for job scheduling in geo-distributed big data processing [24]. While the work mentioned above did not consider job utilities, Huang *et al.* [25] proposed to achieve max-min fairness across different jobs. For network-aware approaches, in [5], Jalaparti *et al.* proposed to coordinate data and task placements to improve the network locality and the overall performance. Grandl *et al.* took more types of resources, such as CPU, memory, and network, into consideration in its job scheduling policy, and aimed to improve the performance of job completion time, makespan and fairness at the same time [4].

Even though we both focus on job scheduling, those solutions assume known information on job sizes, which may make them infeasible for the cases that we mentioned previously.

Resource managers. YARN [26] and Mesos [27] are both open-source resource managers for scheduling jobs from different data processing frameworks in a shared cluster. In YARN, it is the application's responsibility to request resources with specifications from the cluster. While in Mesos, the cluster would offer available resources to the applications first. These applications then decide whether to accept the resource offers. YARN and Mesos both meet all the requirements of our project. We adopt YARN for simplicity.

VII. Discussion

In this section, we discuss a few limitations and potential directions of this work.

First, theoretical analysis regarding the number of queues and the thresholds of queues is not addressed in this paper. As mentioned before, adopting our new ordering approach and weighted fairness sharing increase the complexity of theoretical analysis a lot. Thus we adopt a simple but practical strategy to determine the number of queues and the thresholds of queues, which is validated through extensive experiments and simulations. In the future, we will try to conduct theoretical analysis to set these parameters and make the scheduler more adaptable for different workloads.

Second, we may take fairness as another major objective for the scheduler in the future work. In the current practice, using weighted fairness sharing can achieve better fairness than baselines even though fairness is not explicitly optimized in our algorithm. However, it will be interesting to investigate the tradeoff between fairness and job response times. We plan to design a tunable parameter to make the tradeoff and flexibly adjust the performance as needed.

Third, how to design the scheduling algorithm in cases with low and diverse network bandwidths like geo-distributed big data processing is another interesting potential direction. In these cases, the network transfer times could be comparable or even larger than the CPU times of the jobs. Thus we may need to figure out how to coupling both VMs and network resources in the scheduling process to increase the resource efficiency apart from reducing job response times.

VIII. Concluding Remarks

In this paper, we first show that there are plenty of cases that complete information on job sizes is not available in big data processing systems. To address this challenge, we have designed and implemented a new job scheduler, called *LAS_MQ*, to utilize a multilevel priority queue to mimic a shortest job first policy without complete prior information of jobs. We have also proposed a new way to obtain the amount of service that the job would receive in each stage, as well as a new policy for job scheduling in each queue to further improve the performance. Both our experimental and trace-driven simulation results have strongly confirmed the effectiveness of our scheduler.

Acknowledgment

The authors would like to thank Zengxiang Li for providing the testbed in the experiments and insightful discussions. This work is supported by a research contract with Huawei Corp. and an NSERC Collaborative Research and Development (CRD) grant.

References

[1] "Hadoop." [Online]. Available: https://hadoop.apache.org/

[2] 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. of USENIX NSDI*, 2012.

[3] "Fair scheduler." [Online]. Available: https://goo.gl/xjb3Yp

[4] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource Packing for Cluster Schedulers," in *Proc. of ACM SIGCOMM*, 2014, pp. 455–466.

[5] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware Scheduling for Data-parallel Jobs: Plan When You Can," in *Proc. of ACM SIGCOMM*, 2015, pp. 407–420.

[6] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient Queue Management for Cluster Scheduling," in *Proc. of ACM Eurosys*, 2016.

[7] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a Geo-distributed Data-intensive World," in *Proc. of CIDR*, 2015.

[8] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica, "Low Latency Geo-distributed Data Analytics," in *Proc. of ACM SIGCOMM*, 2015.

[9] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling Tasks Closer to Data across Geo-distributed Datacenters," in *Proc. of IEEE INFOCOM*, 2016.

[10] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang, "Guaranteeing Deadlines for Inter-datacenter Transfers," in *Proc. of ACM Eurosys*, 2015.

[11] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack, "Analysis of LAS Scheduling for Job Size Distributions with High Variance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 218–228, 2003.

[12] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Reoptimizing Data Parallel Computing," in *Proc. of USENIX NSDI*, 2012, pp. 281–294.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A Study of Skew in Mapreduce Applications," *Open Cirrus Summit*, 2011.

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments." in *Proc. of USENIX OSDI*, 2008.

[15] M. Dell'Amico, D. Carra, M. Pastorelli, and P. Michiardi, "Revisiting Size-based Scheduling with Estimated Job Sizes," in *IEEE International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, 2014, pp. 411–420.

[16] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic Flow Scheduling for Commodity Data Centers," in *Proc. of USENIX NSDI*, 2015, pp. 455–468.

[17] M. Chowdhury and I. Stoica, "Efficient Coflow Scheduling Without Prior Knowledge," in *Proc. of ACM SIGCOMM*, 2015, pp. 393–406.

[18] "Capacity scheduler." [Online]. Available: https://goo.gl/c9GS2p

[19] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue Mapreduce Benchmarks Suite," 2012.

[20] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of Mapreduce Workloads," *PVLDB*, vol. 5, no. 12, pp. 1802–1813, 2012.

[21] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An Experimental Time-sharing System," in *Proc. of ACM Spring Joint Computer Conference*, 1962, pp. 335–344.

[22] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling Mix-flows in Commodity Datacenters with Karuna," in *Proc. of ACM SIGCOMM*, 2016, pp. 174–187.

[23] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale," in *Proc. of ACM SIGCOMM*, 2015.

[24] C.-C. Hung, L. Golubchik, and M. Yu, "Scheduling Jobs Across Geo-distributed Datacenters," in *Proc. of ACM SoCC*, 2015, pp. 111–124.

[25] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Need for Speed: Cora Scheduler for Optimizing Completion-Times in the Cloud," in *Proc. of IEEE INFOCOM*, 2015, pp. 891–899.

[26] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop Yarn: Yet Another Resource Negotiator," in *Proc. of ACM SoCC*, 2013.

[27] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." in *Proc. of USENIX NSDI*, 2011.