This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

Low Latency Big Data Processing without Prior Information

Zhiming Hu, *Member, IEEE,* Baochun Li, *Fellow, IEEE,* Zheng Qin, *Member, IEEE* and Rick Siow Mong Goh, *Member, IEEE*

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 experiments in both real testbeds including Amazon EC2 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% and achieve better fairness at the same time.

Index Terms—job scheduling, big data processing, multilevel feedback queue

1 Introduction

Ranging from recommendation systems to business intelligence, the use of big data processing frameworks, such as Apache Hadoop [2] and Spark [3], 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 utilizations 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 can help them to obtain their results faster and thus have the potential to increase the financial revenue.

Simple scheduling policies, such as first-in-first-out (FIFO) and Fair scheduler [4], 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 the 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 in many cases.

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* [5], [6], [7], 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 [8], [9], [10], [11], where available capacities between data centers vary more significantly over time [12]. *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. 2). 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.

- *Zhiming Hu and Baochun Li are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, M5S 2E4.*
  E-mail: zhiming@ece.utoronto.ca, bli@ece.toronto.edu.
- *Zheng Qin and Rick Siow Mong Goh are with the Institute of High Performance Computing, A*STAR, Singapore, 138632.*
  E-mail: {qinz, gohsm}@ihpc.a-star.edu.sg.
- *Preliminary results were presented in Proceedings of the IEEE ICDCS, 2017 [1].*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

In this paper, we design and implement an efficient and practical job scheduler in big 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 that queue. To further improve the performance, we also carefully design efficient ways to schedule jobs in the same queue and across different queues in our system.

Our algorithm is partially motivated by the *least attained service* (LAS) policy [13]. 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 beforehand. 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 [13] 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. For this reason, 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. More specifically, we should avoid assigning resources to tasks that cannot be started at the time of scheduling in a job. 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 allocate slots for `reduce` tasks too early during the scheduling.

Though on the surface our proposed approach looks similar to a priority queue with aging, it is, in fact, remarkably different in the following three perspectives. *First*, our approach only reduces the priority of the jobs based on the amount of service they have received, while the aging method will increase the priority when the jobs wait for a longer period of time. *Second*, our approach is able to reduce the average job response time while the general priority queue with aging will not be able to. *Finally*, we have a completely different method to avoid the starvation problem by assigning resources to not only higher priority queues, but to lower priority queues as well.

Highlights of our original contributions in this paper are as follows. *First*, we 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 in existing systems [14], [15].

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

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.

2 Motivation

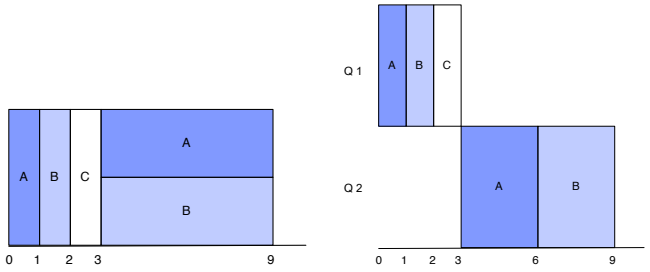
Information about job sizes is critical for superior job scheduling performance, and most previous works assume that such information is known or can be accurately estimated beforehand. However, as we have briefly discussed, information about job sizes may not be realistically available for the following 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 [16], which also implies that more than half of the jobs are ad hoc jobs. Ad hoc jobs would only run once; Thus either the programs or inputs are new, which makes it difficult to predict the running times of tasks or jobs. Some people may argue that if we cannot 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 before the stages are started.

Data skews are common in each stage. For Hadoop/Spark jobs, the ideal case is that the running times of 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 [17]. 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, which can make them 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 [17].

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 are completely different regarding task types and the numbers of tasks [3]. 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). Therefore, though we may know the completion times of stages that have finished, we still have no idea about the completion times of pending stages. In other words, we are still not able to predict the completion times of the entire job before it is complete.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing



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

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

Even for recurring jobs, resource sharing and unstable networks can also cause unpredictable running times. Normally, the network and I/O bandwidths are resources that are shared across concurrent jobs in the same cluster. Moreover, in some cases, the amount of available bandwidths 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 [12], the 95th percentile value could be several times larger than 5th percentile value within 35 hours. In these cases, unfortunately, even recurring jobs may have unpredictable performance regarding job completion times.

2.1 A Motivating 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 [13]. Even though some workloads of big data processing systems do follow heavy-tailed distribution in the long run [16], 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. If the job size is x, it means that the job will take x unit of time to finish when it can run alone in the cluster. 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[2] 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 where the jobs in the first queue have higher priorities than the jobs in the second queue. The scheduler will always schedule the jobs in the queue with the highest priority first. Therefore, in this case, A and B are demoted to the second queue after they used the cluster for one unit of time. Then C obtains the resources and starts to run. After C is complete, the first queue is now empty and we can now start to schedule the jobs one by one in the second queue. 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, which is reduced by 33%.

There are two lessons behind this simple motivating example. First, multilevel priority queue can effectively separate large jobs from small jobs and place them in different queues. Like the example in Fig. 1, A and B are identified as larger jobs and thus are demoted to the second queue. Therefore, it can mimic the shortest job first strategy effectively. Second, we can avoid the *processor sharing* problem for large jobs if we schedule the jobs one by one in each queue. For example, in Fig. 1(b), we schedule job A first followed by job B and we can mitigate the *processor sharing* issue in Fig. 1(a). In sum, we can reduce the average job response times because we can effectively mimic the shortest job first strategy and we can avoid the *processor sharing* problem existed in algorithms like FAIR and LAS.

2.2 Desirable Features

Given the motivating example, we can know that we can improve the performance by adopting the multilevel queue and a wise scheduling strategy in each queue. Before we introduce our design, here we list a few desirable features for our system.

Practical: Our system is first designed to be practical. To achieve practicality, we should not make assumptions about the job size information and our system should be ready to be applied in popular resource allocation frameworks such as YARN [18] and Mesos [19]. To this end, it should only depend on the inputs that are readily available in those frameworks, which means that we should not use the information that are not available.

Efficient: Our scheduling system should also be efficient as the number of jobs running at the same time could be large. Therefore, the scheduling algorithm should be able to gather the inputs and delivery the scheduling results quickly. This property is also the main reason why simple heuristic scheduling algorithms instead of sophisticated ones are adopted in big data processing frameworks like Hadoop and Spark.

Robust: Robust means that our system need to meet the following requirements. First, it should be adaptable for a variety of workloads. Second, robust means that the scheduler should be able to react to the current job progresses instead of static scheduling that does not make any changes to the scheduling results until the job finishes. This property is very important because running times of small jobs could become large for stragglers or high resource contention. The desired scheduler should be able to adjust its scheduling results adaptively for these cases.

Job response time: Minimizing the average job response time is the ultimate goal of our system, which is an important factor to achieve users' satisfaction. Our system should achieve lower job response times under different workloads

2. The amount of service is calculated by the product of the amount of resources and the amount of time in using the resources.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

compared to classical scheduling algorithms like first-in-first-out (FIFO) and Fair scheduling. Moreover, we also aim at reducing the long tail job response time as well.

2.3 Problem Formulation

After we have talked about the motivations and desired properties, now let us have a look at the problem itself.

Generally, it is a resource allocation problem. Let us assume that there are m jobs and the completion time of i-th job is represented by f_i. There are n types of resources in the cluster, and the resources in the cluster are specified by $\mathcal{C} = \{C_1, C_2 \dots C_n\}$, where C_r is the amount of the r-th resource in the cluster. We use d_{ir} to show the resource demand of the i-th job on the r-th resource. The total number of time slots is τ and amount of r-th resource allocated to i-th job at t-th slot is x_{ir}^t. More formally, the problem can be formulated as follows:

$$\min_{\boldsymbol{x}} \quad \sum_{i}^{m} f_i \tag{1}$$

$$\text{s.t.} \quad f_i = \{\max t \,|x_{ir}^t > 0\}, \ \forall i \tag{2}$$

$$\sum_{i=1}^{m} x_{ir}^t \leq C_r, \qquad \forall r, \forall t \tag{3}$$

$$\sum_{t=1}^{\tau} x_{ir}^t = d_{ir}, \qquad \forall i, \forall r \tag{4}$$

$$x_{ir}^t \in Z^0. \qquad \forall i, \forall r, \forall t \tag{5}$$

Job response time is calculated by job completion time minus job submission time. In the objective function, we only minimize the summation of job completion times because job submission times are fixed in the scheduling. The first constraint (2) is to define the job completion time, after which, it would not receive any resource from the scheduler. The second constraint (3) is to guarantee that the resource limits of the cluster are met for all the resource types at any time. The resource requirements of the job is listed in (4). The last constraints shows that each allocation in each time slot is an integer because we can only use integers to represent some resources like CPU cores. As we can see, the problem here is an **integer programming problem with a non-linear constraint in Eq. (2)**, which cannot be solved efficiently. What's worse, we do not know the job sizes d_{ir} in Eq. (4) in the cases if we do not know the prior information about job sizes beforehand, which makes it far from practical to solve this problem. Therefore, we focus on practical and efficient heuristic approaches.

3 System Design

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

3.1 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: $\alpha_1, \alpha_2 \dots \alpha_{k-1}$.

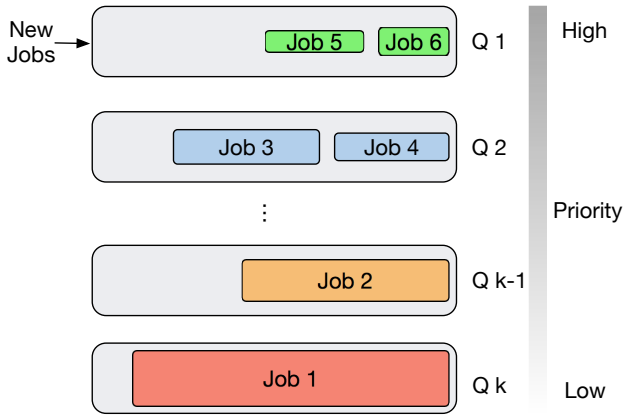


Fig. 2. The design of LAS_MQ: an overview.

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. 3.3.

3.2 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

job for t seconds, the amount of service that job j received j_s in that period is defined as follows:

$$j_s = x \cdot t. \tag{6}$$

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 [20]. 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 [21]. 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. In the implementation, we only apply the stage awareness when the stage progress is no less than 10% because the estimations are more accurate after the stages have been executed for more than 10%.

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.

3.3 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.

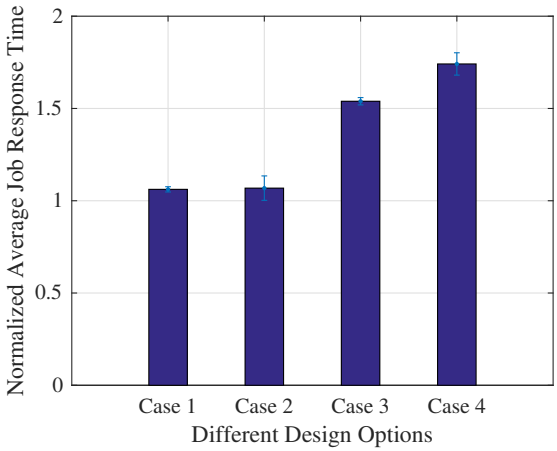


Fig. 3. Case 1 represents the traditional design without either feature. Case 2 shows the result with stage-awareness only. Case 3 shows the result only with job ordering in each queue. Case 4 shows the result with both features. All the results are normalized over the Fair scheduler in YARN.

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 1 and Case 2, it seems that stage awareness

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

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.

3.4 Resource Allocation with Dependencies

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.

3.5 DAG Awareness and Work Conservation

Even though we allocate the resources stage by stage, there are some cases that the later stages are hanged by a few pending tasks in the previous stage. This phenomenon is also identified in [5], where the jobs that have finished more than 85% or 95% of the tasks in a stage will be given higher priorities. In our case, we find out that our job ordering strategy in each queue can handle this situation well. For instance, if there are only a few reduces tasks are left for a job, there is a high chance that the job is placed in the front part of the queue. If there is only a few map tasks pending instead, the situation is similar because the number of reduce tasks is normally much smaller than the number of map tasks.

Besides DAG awareness, work conservation also plays an important role in our algorithm. By enabling work conservation, we can increase the utilizations of computing resources in the cluster. Thus it can effectively improve the overall performance.

3.6 The 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 [14]. However, we propose to adopt weighted fair sharing among queues, with specific ordering in each queue. The methods in [14] cannot be directly applied in our case. Instead, we propose a simpler approach as stated in [15] 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
```

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 ∈ 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.
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

3.7 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 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 $\mathrm{O}(kn)$.

4 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 design, it contains three major components: job admission, job scheduler and resource manager (YARN). 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. If not, the job will still be in the queue of pending jobs.

Our scheduler is built on top of capacity scheduler [22]. In capacity scheduler, it can change the capacities of queues by updating the configuration file on a real-time basis. In our implementation, each application is assigned to a unique queue. Thus, we can control the amount of resources for each application by setting the capacities of queues. After the scheduling process, 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 admit more jobs to the scheduler. Here we also record the job response time as the gap between the job completion time and the time when the job is admitted into 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 resource in the cluster. The scheduling problem now becomes how to decide the number of containers that should be allocated to each job. We keep track of the number of remaining tasks in the current stage of jobs during the scheduling process to calculate a proper number of containers to it.

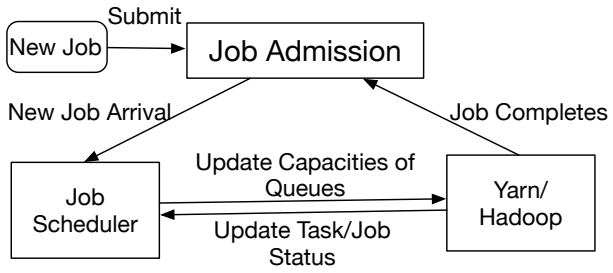


Fig. 4. Implementing the LAS_MQ scheduler: overall design.

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 especially the map 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.

After we know the number of remaining tasks in the stage, we allocate the calculated number of containers or all the remaining containers in the cluster, 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. Note that the number of remaining tasks does not count the number of tasks in other remaining stages because the tasks in remaining stages can only start after the current stage completes.

Currently, our prototype is implemented with Hadoop applications. But our design is also compatible with other big data processing frameworks like Spark [3]. The reasons are twofold. First, we allocate resources stage by stage, which is initially supported in Spark. Second, It is also very convenient to know the number of tasks in each stage and whether a stage is shuffle dependent.

5 Performance Evaluation

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

5.1 Experimental and Simulation Setup

Private 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) [2]. 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.

Amazon EC2: As the cluster size of our private testbed is limited, we also conduct extensive experiments in Amazon EC2. Our cluster consists of 16 m4.xlarge instances, of which 15 nodes are used as slave nodes. Each instance has 4 vCPU cores and 15 GB of main memory. The master node and each slave node have 200 GB and 50 GB of solid state disk (SSD) as the storage, respectively. In the settings for YARN, the size of main memory allocated for each *NodeManager* is 8 GB. Thus we can start 4 containers in each node and 60 containers at most for the whole cluster. In HDFS, the replication factor is 3. All the other settings are the same as the ones in private testbed.

Workload and Inputs: Our workload contains 100 jobs for the experiments in the private testbed and 30 jobs in the EC2. All the jobs are randomly selected from *TeraGen*, *SelfJoin*, *Classification*, *HistogramMovies*, *HistogramRatings*, *SequenceCount*, *InvertedIndex* and *WordCount*, all of which are popular jobs for benchmarking Hadoop in PUMA benchmarks [23]. 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 [23]. 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.[3] More details about the workloads and inputs are available in Table 1 and Table 2, respectively. The main differences are that the workloads used in the Amazon EC2 has smaller input sizes for the WordCount and the total number of jobs is changed to 30 because of the resource limitations.

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 [24], 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 [21]. 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, FIFO scheduler, which is the default ordering policy in capacity scheduler, equally sharing scheduler (EQU) and the scheduler with complete job sizes (IDEAL), which serve as the optimal solution for the problem. 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. For the equally sharing scheduler, it equally shares the resources to all the running jobs in the system. The scheduler with complete job sizes needs to run each job once and gather the running information of jobs such as the running time of map tasks and reduce tasks as the input. Then it simply applies the shortest job first strategy to schedule all the running jobs. The remaining job sizes are calculated by the summation of the products of the resource demand of the task and the running time of the task.

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. Even though our main target is not improving the performance regarding makespan, we also evaluated the makespans in the results.

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 1
The workload used in the experiments in the private testbed.

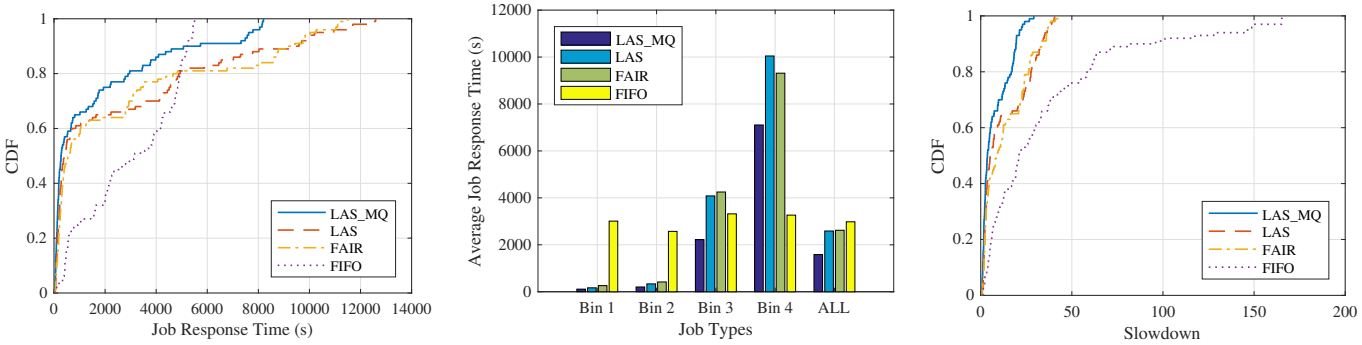
Bin	Job Name	Dataset Size	# of maps	# of reduces	# of jobs
1	TeraGen	1 GB	100	10	3
1	SelfJoin	1 GB	102	10	15
2	Classification	10 GB	102	20	17
2	HistogramMovies	10 GB	102	20	12
2	HistogramRatings	10 GB	102	20	8
3	SequenceCount	30 GB	234	60	16
3	InvertedIndex	30 GB	234	60	19
4	WordCount	100 GB	721	80	10

TABLE 2
The workload used in the experiments in Amazon EC2.

Bin	Job Name	Dataset Size	# of maps	# of reduces	# of jobs
1	TeraGen	1 GB	100	10	3
1	SelfJoin	1 GB	102	10	4
2	Classification	10 GB	102	20	4
2	HistogramMovies	10 GB	102	20	3
2	HistogramRatings	10 GB	102	20	4
3	SequenceCount	30 GB	234	60	3
3	InvertedIndex	30 GB	234	60	3
4	WordCount	50 GB	396	80	6

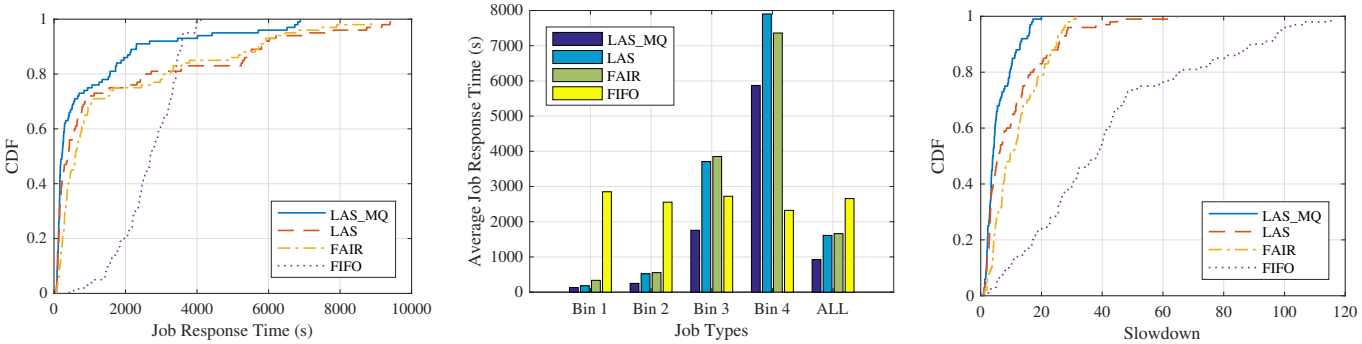
3. For TeraGen, we do not need to define the input size and we set the output size instead.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing



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

Fig. 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.

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

5.2 Experimental Results in Private Testbed

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?

5.2.1 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 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.

5.2.2 Fairness

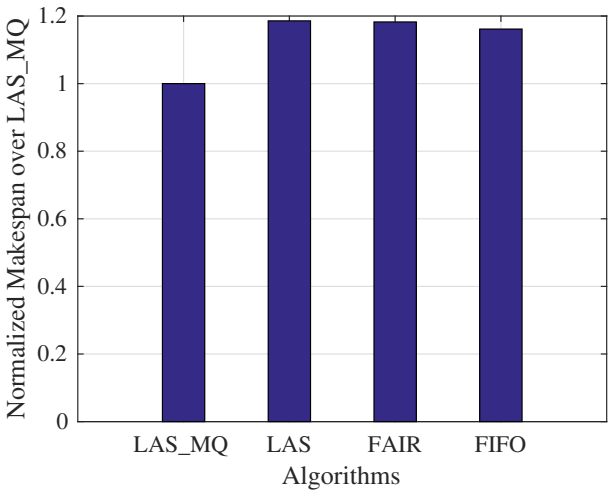
Even though fairness is not the main goal of our design, we also show the results as fairness is a very important metric for job scheduling. We show our results of the slowdown in Fig. 5(c), and we can see that again our solution has the smallest slowdowns followed by LAS, Fair scheduler and FIFO. The 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

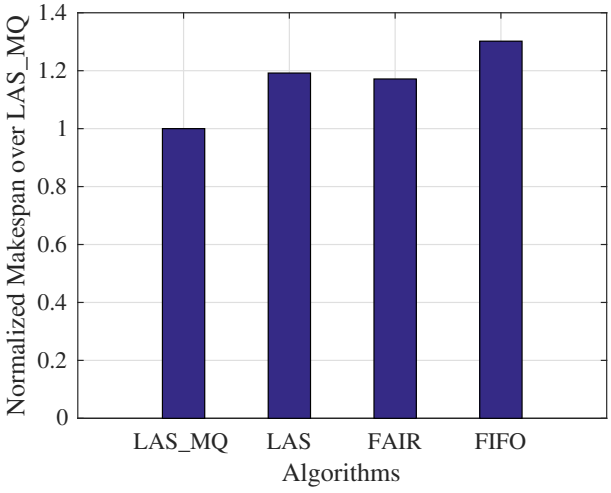
to our design using a multilevel queue and weighted fair sharing among queues.

5.2.3 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.



(a) Mean arrival interval of 80s



(b) Mean arrival interval of 50s

Fig. 7. The makespan of algorithms for different arrival intervals.

5.2.4 Makespan

Even though reducing the makespan is not the main objective of our system, we also measure the performance of our system regarding makespan. The results are normalized over the makespan of LAS_MQ. In Fig. 7(a), we draw the results of makespan when the mean arrival of the interval is 80 seconds. We can see that the makespans of the other three algorithms are similar and are around 1.2 times to the result of LAS_MQ. For LAS, when there are free resources left after the job with the least attained service is scheduled, the free resources are shared among all the other jobs. In this case, the sharing process might not be efficient enough as it may assign more resources to some jobs than they need, which can lead to low utilization of resources. Similar situations also happen to FAIR and FIFO. For FAIR, which directly shares all the resources to jobs according to priorities only, can also result in low utilizations of resources and larger makespans. FIFO will assign all the resources to the job that arrives the first and will incur the same results.

We also depict the results when the mean arrival of the interval is 50 seconds in Fig. 7(b). In the figure, we can see that LAS_MQ still performs the best and the makespans of the other three algorithms are more than 1.2 times to the one of our algorithm. In this figure, the performance of FIFO is the worst. The reasons are as follows. First FIFO cannot fully utilize the resources in the cluster for many jobs. Second, LAS_MQ, LAS and FAIR have better performance in this case than the case with the mean arrival interval of 80 seconds, which can be also seen in Fig. 5 and Fig. 6.

5.3 Experimental Results in Amazon EC2

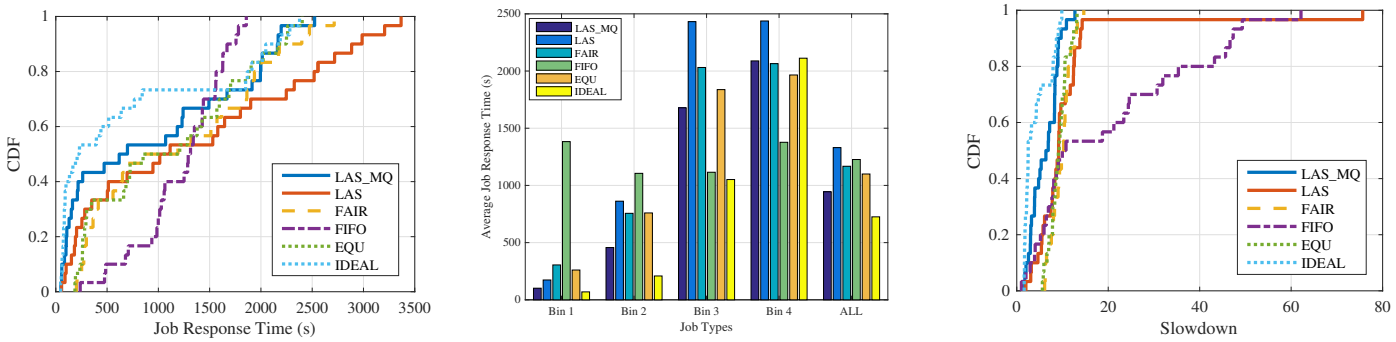
As there are only four large workstations in the testbed experiments, we further examine the performance of our algorithm with 16 m4.xlarge instances in Amazon EC2. We not only increase the number of nodes in the experiments but also compare our algorithm with two more algorithms, which are EQU and IDEAL. EQU is the algorithm that shares the resources equally to all the pending jobs, which is different from FAIR because FAIR shares the resource according to the priorities of jobs. The algorithm named IDEAL is the algorithm that has complete information about job sizes and adopts shortest remaining job first strategy. This algorithm can let us know the gap between our algorithm and the optimal solution with ideal information about job sizes. Here we also compare the performances regarding average job completion times, slowdowns and makespans.

5.3.1 Average Job Response Times

We show the CDF of the job response times in Fig. 8(a). In this figure, we can see that our algorithm is better than all the other algorithms except IDEAL. Again, the reason is that our algorithm can effectively separate large jobs from small jobs and schedule the large jobs one by one. LAS, FAIR and EQU both have similar performances. We can also see that around 20% of the jobs have better performance than LAS_MQ and IDEAL. To better understand the results, we also show the average job response times of jobs in different bins.

In Fig. 8(b), we can see that the reason is that FIFO has better performance for large jobs. The reason is the same

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing



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

Fig. 8. The performance of the workload with the mean arrival interval of 80 seconds in Amazon EC2.

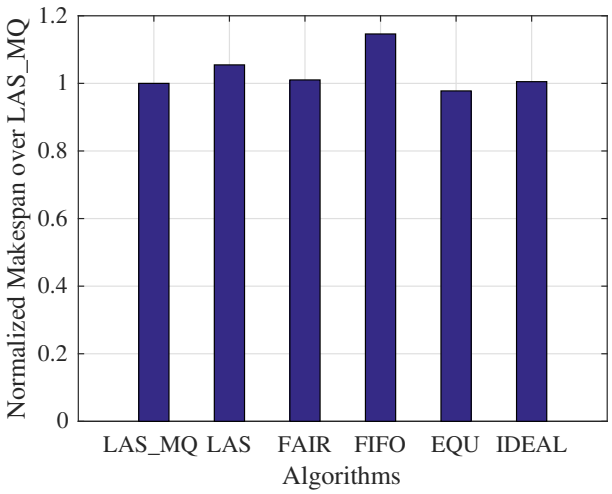


Fig. 9. The makespan of algorithms.

as we discussed before, which is FIFO will schedule the large jobs when they are available. However, both LAS_MQ and IDEAL would schedule smaller jobs first and delay those large jobs intentionally to minimize the average job completion time. Thus LAS_MQ and IDEAL have lower average job completion times for all the jobs. In the figure, by looking at the results of LAS_MQ and IDEAL, we can see that our algorithm has comparable performance with IDEAL in Bin 1 and Bin 4, which means our algorithm can actually identify large and small jobs. However, our multi-level queue structure has some extra overhead for other jobs to find their belonging position, which results in performance degradations in Bin 2 and Bin 3. Last but not the least, EQU has almost the same performance with the FAIR scheduler.

5.3.2 Fairness

We also show the results of slowdowns in Fig. 8(c). Again, IDEAL has the best performance, followed by our algorithm without knowing the information about job sizes. The result is aligned with the results in Fig. 5(c) and Fig. 6(c). In all the cases, FIFO has the worst performance because of the large response times of small jobs.

5.3.3 Makespan

We also depict the results regarding makespan in Fig. 9. In the figure, we can see that FIFO performs the worst and the other five algorithms including IDEAL have similar performances. FIFO has the worst performance because it always allocates 100% of the resources to the first available job, which results in low utilization of resources. It is also interesting to see that our algorithm LAS_MQ does not show improvements over LAS and FAIR as shown in Fig. 7(a) and Fig. 7(b). After careful analysis, we find out the main reason is that total amount of resource is two times less than the ones in the case of the private testbed. Thus LAS and FAIR will have higher resource utilizations and thus have better makespans compared with the case in the private testbed.

5.4 Simulation Results

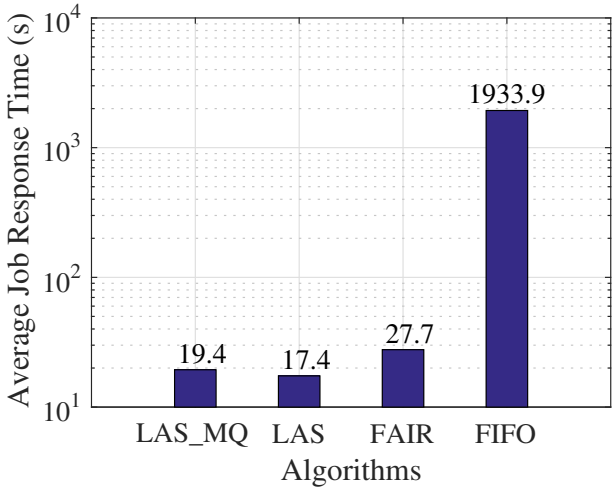
With our simulations, we would like to investigate the performance of our scheduler with different distributions and the sensitivity of our approach to different parameter settings.

5.4.1 Workloads following Different Distributions

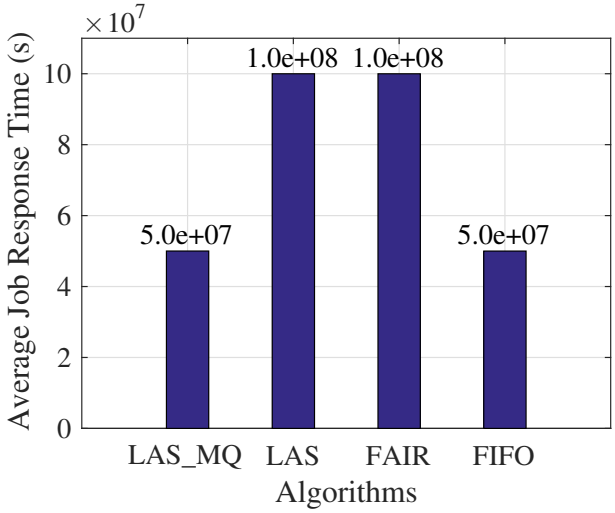
We compare the performance of algorithms for workloads under both heavy-tailed and light-tailed distributions. In these two cases, for LAS_MQ, the number of queues and steps are 10. The threshold of the first queue is 1. The results are shown in Fig. 10. In Fig. 10(a), we can see that, for the heavy-tailed 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

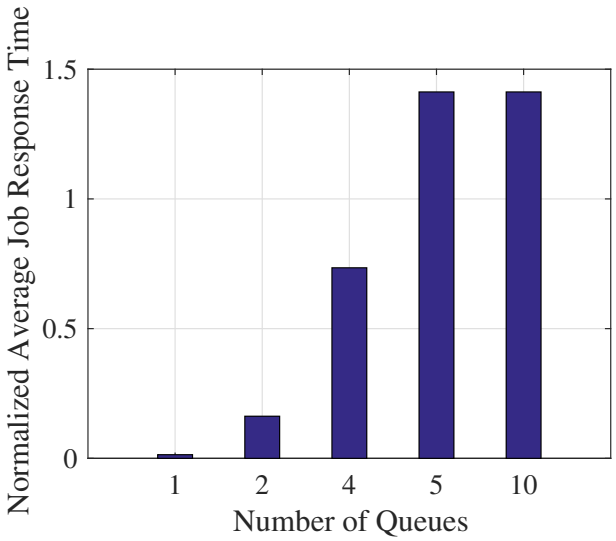


(a) Heavy-tailed distribution

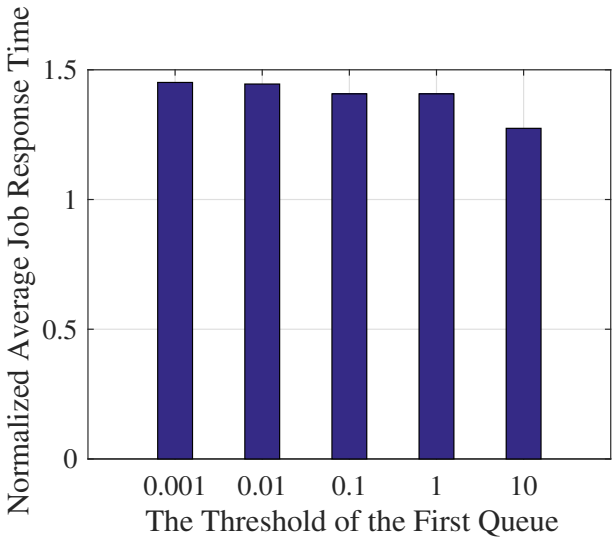


(b) Uniform distribution

Fig. 10. 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.

Fig. 11. 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.

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.

5.4.2 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. 11. 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. 11(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.

6 Related Work

Scheduling without prior information. Multilevel feedback queue (MLFQ) [25] was a scheduling algorithm that prefers small jobs and I/O bound processes. It also does not require

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

prior information about job sizes. Bai *et al.* [14], [26] implemented a MLFQ in commodity switches for flow scheduling in data center networks. Least attained service (LAS) [13] 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 [15] for coflow scheduling. Theoretical analysis on the thresholds of the queues for multilevel queue based coflow scheduling was proposed in [27].

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.

General job scheduling. For non-network aware approaches, Hopper [28] took speculation into consideration 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 [29]. While the work mentioned above did not consider job utilities, Huang *et al.* [30] proposed to achieve max-min fairness across different jobs. For network-aware approaches, in [6], 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 [5]. Tan *et al.* proposed to improve the job response times by coupling the progresses of `map` tasks and `reduce` tasks [31] in Hadoop jobs.

Even though we both focus on job scheduling, the solutions that assume known information on job sizes may be infeasible for the cases that we mentioned before.

Resource allocation and planing. YARN [18] and Mesos [19] 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. However, the flexibility and parallelism of Mesos are limited by conservative resource visibility. To resolve these issues, Omega [32] proposed a shared state and optimistic concurrency control approach. Matrix [33] proposed automatic VM configurations and recommendations to achieve predictable performance of workloads by applying machine learning algorithms.

Rayon [34] proposes an efficient heuristic to reserve the resource demands for deadline aware jobs while minimizing the latencies for small jobs. While Morpheus [35] starts from deriving the Service Level Objectives (SLOs) and presents a low-cost based resource allocation scheme. Instead of placing the pending jobs one by one, TetriSched [36] can make global decisions for all the pending jobs.

7 Discussions and Insights

There exist several limitations in our proposed approach, which may lead to potential future directions of research.

First, a theoretical analysis regarding the number of queues and the thresholds of queues has not been included in this paper. As we previously mentioned, adopting a new ordering approach and weighted fair sharing would increase the complexity of theoretical analysis substantially. Thus, we have chosen to adopt a simple but practical strategy to determine the number of queues and the thresholds of queues, and the effectiveness of such a strategy is validated through our extensive experiments and simulations. As future work, it would be interesting to conduct a more detailed theoretical analysis to set the parameters and make the scheduler more adaptive to 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 trade-off between fairness and job response times. We plan to design a tunable parameter to make the trade-off 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.

8 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.

References

[1] Z. Hu, B. Li, Z. Qin, and R. S. M. Goh, "Job scheduling without prior information in big data processing systems," in *Proc. of IEEE ICDCS*, 2017, pp. 572–582.
[2] "Hadoop." [Online]. Available: https://hadoop.apache.org/
[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] ——, "Time-and Cost-Efficient Task Scheduling across Geo-Distributed Data Centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 705–718, 2018.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] Y. Gao, H. Yu, S. Luo, and S. Yu, "Information-Agnostic Coflow Scheduling with Optimal Demotion Thresholds," in *Proc. of IEEE ICC*, 2016, pp. 1–6.

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

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

[30] 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.

[31] J. Tan, X. Meng, and L. Zhang, "Coupling Task Progress for Mapreduce Resource-aware Scheduling," in *Proc. of IEEE INFOCOM*, 2013, pp. 1618–1626.

[32] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proc. of ACM Eurosys*, 2013, pp. 351–364.

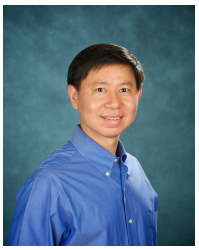
[33] R. C.-L. Chiang, J. Hwang, H. H. Huang, and T. Wood, "Matrix: Achieving Predictable Virtual Machine Performance in the Clouds." in *Proc. of USENIX ICAC*, 2014, pp. 45–56.

[34] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based Scheduling: If You're Late Don't Blame Us!" in *Proc. of the ACM SoCC*, 2014, pp. 1–14.

[35] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards Automated SLOs for Enterprise Clusters," in *Proc. of USENIX OSDI*, 2016.

[36] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters," in *Proc. of ACM Eurosys*, 2016.

Zhiming Hu received his BS degree in computer science from Zhejiang University, China, in 2011 and his Ph.D. degree from the School of Computer Science and Engineering, Nanyang Technological University, Singapore, in 2016. He is now a postdoctoral fellow in the Department of Electrical and Computer Engineering, University of Toronto, Canada. His research interests include big data processing, data center networking, and cloud computing.



Baochun Li received his B.Engr. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and his M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000. Since 2000, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a Professor. He holds the Bell Canada Endowed Chair in Computer Engineering since August 2005. His research interests include cloud computing, multimedia systems, applications of network coding, and wireless networks. He was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems in 2000, the Multimedia Communications Best Paper Award from the IEEE Communications Society in 2009, and the University of Toronto McLean Award in 2009. He is a member of ACM and a Fellow of IEEE.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2910251, IEEE Transactions on Cloud Computing

Zheng Qin is a Senior Scientist and leads the Distributed Computing Capability Group at the Institute of High Performance Computing (IHPC), Agency for Science, Technology and Research (A*STAR). He graduated with BEng in Information Engineering from Xi'an JiaoTong University in 2001 and PhD in Electrical and Computer Engineering from National University of Singapore in 2006. He has been with IHPC Since November 2007. His research interest includes scheduling for distributed systems, large-scale spatial-temporal data processing, and data-driven urban simulation.

Rick Siow Mong Goh is the Director of the Computing Science (CS) Department at the A*STAR's Institute of High Performance Computing (IHPC). At IHPC, he leads a team of more than 60 scientists in performing world-leading scientific research, developing technology to commercialization, and engaging and collaborating with industry. The research focus areas include high performance computing (HPC), distributed computing, artificial intelligence, and complex systems. His expertise is in discrete event simulation, parallel and distributed computing, and performance optimization and tuning of applications on large-scale computing platforms. Dr. Goh received his Ph.D. in Electrical and Computer Engineering from the National University of Singapore. More details on Rick can be found at: http://www.a-star.edu.sg/ihpc/People/tid/2/Rick-Goh-Siow-Mong.aspx.