Cluster Fair Queueing: Speeding up Data-Parallel Jobs with Delay Guarantees

Chen Chen, Wei Wang, Shengkai Zhang, Bo Li
Hong Kong University of Science and Technology
{cchenam, weiwa}@cse.ust.hk, szhangaj@connect.ust.hk, bli@cse.ust.hk

***Abstract*—Cluster scheduler serves as a critical component to data-parallel systems in datacenters. Ideally, a scheduler should provide predictable performance with guarantees on the maximal job completion delay, while at the same time ensuring the minimal mean response time. Practically however, performance predictability and optimality are often conflicting with each other. The results often are a plethora of scheduling policies that either achieve predictable performance at the expense of long response times (e.g., max-min fairness), or run the risk of starving some jobs to obtain the minimal mean response time (e.g., Shortest Remaining Processing Time First). To address these problems, we develop a new scheduler, Cluster Fair Queueing (CFQ), which preferentially offers resources to jobs that complete the earliest under a fair sharing policy. We show that CFQ is able to minimize the mean response time while at the same time ensuring jobs to finish within a *constant time* after their completion under fair sharing. Our Spark deployment on a 100-node EC2 cluster demonstrates that compared to the built-in fair scheduler, CFQ can decrease the mean response time by 40%, which speeds up more than 40% of jobs by over 75% on average.**

I. Introduction

With the wide deployment of data-parallel frameworks like Spark [1] and Hadoop [2], it has become a norm to run data analytics applications in a large cluster of machines. Having different applications coexisting in a cluster, data analytics *jobs*, each consisting of many parallel *tasks*, expect *predictable performance* with guarantees on the *maximal completion delay*. Cluster operators, on the other hand, aim to minimize the *response times* of jobs, i.e., the time between the instants of job arrivals and completions.

Prevalent cluster schedulers deployed in today's datacenters rely on *fair sharing* to provide *predictable performance*, e.g., [3]–[7]. By seeking max-min fair allocations *at all times*, fair schedulers aim to assure that each job receives equal amounts of cluster resources (to the degree possible), regardless of the behaviors of the other jobs, therefore, achieving performance isolation from one another. However, it has been widely confirmed that fair schedulers can be inefficient, and may result in significantly long response times [8]–[10].

Given the inefficiency of fair schedulers, many recent proposals (e.g., [10]–[14]) turn to *performance-optimal heuristics* to minimize the job response time, ranging from the Shortest Remaining Processing Time First (SRPT) to first-in-first-out (FIFO) variants. These heuristics, while outperforming fair schedulers with shorter mean response time, fail to provide predictable job performance. For example, SRPT-based heuristics prioritize jobs whose remaining work requires the least processing efforts, which may starve "elephants" (big jobs) if "mice" (small jobs) keep arriving over time.

Fair sharing and performance-optimal heuristics represent the two extremes on the design spectrum of cluster scheduling. Unlike existing approaches that seek a tradeoff between fairness and performance, e.g., [10], [15], we ask a bold question: *is it possible to achieve the near-optimal response times of jobs without compromising the performance predictability provided by fair sharing?*

In this paper, we give an affirmative answer to this question. We propose to preferentially serve jobs that complete the *earliest* under fair sharing. This approach achieves the best of both worlds. First, it minimizes the mean response time by mimicking the behavior of SRPT: under fair sharing, jobs receive an equal share of the cluster resources, and those having the least remaining work likely complete the earliest and are hence prioritized to schedule. Second, because jobs are served in ascending order of completion times under fair sharing, it is *less likely* to have one delayed long after its completion using a fair scheduler, nor would it be starved. As a result, jobs can expect similar predictable performance as under fair sharing.

We develop a new cluster scheduler, Cluster Fair Queueing (CFQ), to implement this idea. CFQ builds upon the classical model of Generalized Processor Sharing (GPS) [16], [17] and extends that from a single network router to a cluster of machines. Specifically, CFQ maintains the cluster GPS—an idealized fair scheduler in clusters—as a reference system, and preferentially offers resources to jobs that complete the earliest under the cluster GPS. This is technically different from the traditional fair queueing where the algorithm tracks GPS in the unit of *packets* [16]–[19]—the equivalent of *tasks* in datacenter environments. In contrast, CFQ focuses on the completion times of *jobs* (i.e., the equivalent of *flows* in fair queueing), and requires new techniques in its implementation.

We show that CFQ provides the predictable performance with the job completion delay *no more than a small constant time* beyond that under the cluster GPS. We further evaluate CFQ against different schedulers using a Spark deployment on a 100-node Amazon EC2 cluster, as well as in simulations over the workload traces from a Google's cluster [20]. Compared to the fair scheduler, CFQ reduces the mean response time by 40% in our Spark deployment, speeding up more than 40% of jobs by over 75% on average. The improvement over the fair scheduler becomes more salient at a larger scale. In our trace-driven simulations, jobs with more than 10 tasks can expect an order of magnitude shorter response time using CFQ. Both Spark deployment and trace-driven simulations show that CFQ achieves the near-optimal response time, close to SRPT.

II. Related Work

Cluster scheduling has been extensively studied in both theory and systems with different objectives. In this section, we briefly examine the existing works by illustrating how they could suffer from long response time or run the risk of violating performance guarantees. We broadly categorize existing cluster schedulers into three approaches: fairness, performance-optimal heuristics, and fairness-performance tradeoffs.

Fairness. Max-min fairness is perhaps the most widely-adopted cluster scheduling approaches in today's datacenters. In a nutshell, it allocates an equal amount of cluster resources to jobs to run their tasks [3]–[7], [9]. This type of scheduler usually provides predictable performance by ensuring each job to receive the deserved fair share throughout its execution, irrespective of the demand of another. This has been shown, however, that it does not necessarily lead to job speed up, yet can suffer from long response times [8]–[10]. For instance, fair schedulers may slow down jobs by 30-50% on average compared to a performance-optimal heuristic [10].

Performance-optimal heuristics. Unlike fair schedulers, the performance-optimal heuristics aim to minimize the mean response time by completing as many jobs as possible. It has been shown in theory that the optimal strategy is to preferentially serve jobs whose remaining processing time is the shortest [21], [22], known as the Shortest Remaining Processing Time First (SRPT). The optimality of SRPT motivates many recent proposals to prioritize the execution of "mice" jobs over "elephants" [10]–[14]. However, SRPT-based schedulers can potentially result in unpredictable performance of "elephant" jobs, whose response time may suffer as a result of the bias towards "mice".

Fairness-performance tradeoffs. Given the deficiencies of fair schedulers and performance-optimal heuristics, recent works have made attempt to strike a balance between fairness and performance. For example, Grandl et al. [10] incorporate fairness into SRPT and propose a tunable fairness knob, with which a flexible tradeoff between fairness and performance can be achieved. HSPF [15] aims to approximate SRPT without job starvation. It employs an aging model that tracks the remaining work of each job under a *virtual fair sharing* system. These works retain better fairness than SRPT. However, they do not provide any guarantees on the maximal job completion delay.

While performance and fairness do not align with each other in the existing work, Grandl et al. [10] show by experiments that optimal performance does not mandate a significant fairness loss. This encouraging result motivates us to investigate whether it is feasible and how in a realistic scenario that both performance guarantee and optimality can be achieved at the same time.

III. Objectives and Motivation

In this section, we first outline our design objectives and identify the root cause behind the inefficiency of fair schedulers. Through a simple experiment, we show that achieving the predictable performance does not necessarily incur long response times. We provide our key intuitions and discuss the challenges.

A. Objectives

Performance and fairness are the two common requirements for a cluster scheduler. The performance usually aims to minimize the mean response time of jobs. The *response time* of a job is measured as the time between the moment the job arrives and the moment it completes. For job-j, let a_j be its arrival time and f_j the completion time, then its response time is defined as $f_j - a_j$. It is straightforward to see that minimizing the mean response time is equivalent to minimizing the total response time of all jobs, i.e., minimize $\sum_j f_j - a_j$.

Fairness, on the other hand, is an essential measurement to dictate predictable performance across different jobs. Specifically, here we say a scheduler achieves the predictable performance if the completion time of each job is within a *constant* of that achieved by a fair sharing policy. Formally, given a scheduler, let f_j be the completion time of job-j and $\bar{f}_j$ the completion time of the same job under fair sharing, then the scheduler ensures the *predictable performance* if for any job-j, there is $f_j - \bar{f}_j \leq C$, where C is a small constant.

Our objective is to design a cluster scheduler that minimizes the mean response time of jobs while achieving the predictable performance comparable to fair sharing at the same time, i.e.,

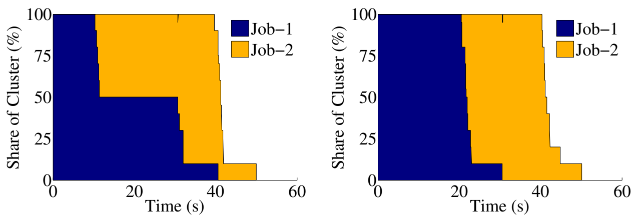
$$\begin{aligned} &\text{minimize} \quad && \sum_j f_j - a_j, \\ &\text{subject to} \quad && f_j - \bar{f}_j \leq C, \ \text{ for all job-}j. \end{aligned} \tag{1}$$

B. Inefficiency of Fair Sharing

Prevalent fair schedulers (e.g., [3]–[7]) maintain *instantaneous fair allocations* at all time, thus the predictable job performance. However, ensuring instantaneous fairness is not relevant to shortening the response time. Making things worse, fair allocations can enforce all jobs to share the cluster with others throughout the execution—those that could have been sped up by taking more cluster resources are now limited to only a smaller share, inevitably delaying their response times.

To illustrate these problems, we run two sample Spark jobs in a 10-node Amazon EC2 cluster, where job-1 is submitted first, followed by job-2. Fig. 1a shows the cluster shares of the two jobs over time using Spark Fair Scheduler [4]. We see that upon the submission of job-2, the scheduler enforces fair sharing by taking half of the compute slots from job-1 and using them to serve job-2. This is unnecessary, as it slows down job-1 without speeding up job-2. If we do not share the cluster but instead prioritize job-1 over job-2, as shown in Fig. 1b, we could speed up job-1 without slowing down job-2.

Observing from this, first, maintaining the instantaneous fairness at all time is not necessary. After all, a job cares *little* about the instantaneous resource allocations, but the completion time of *all* its tasks. Second, enforcing resource sharing is *inefficient* in that it slows down job completion (e.g., job-1 in Fig. 1a). Recognizing the inefficiency of instantaneous fairness,



(a) Fair sharing. (b) Prioritizing job-1 over job-2.

Fig. 1: Illustration of the inefficiency of fair sharing. Two sample Spark jobs, each consisting of 40 tasks, run in a 10-node EC2 cluster where each node (`m4.large`) can run 2 tasks. (a) Cluster shares over time using the fair scheduler. (b) Cluster shares over time where job-1 is prioritized over job-2.

an earlier work [23] defines that an allocation scheme P, from a practical point of view, is *fair* as long as it *weakly dominates* instantaneous fair sharing, i.e., no job completes in P later than it would when instantaneous fair sharing is enforced. This suggests that good performance and fairness can be achieved at the same time.

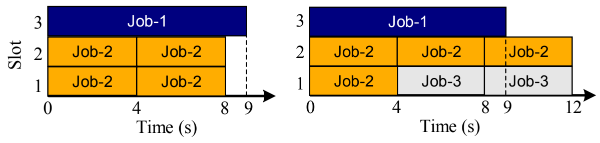
C. Key Intuition

Based on the observations from the experiment, we see that, in order to minimize the job response times while achieving the predictable performance, a scheduler should prioritize the job execution subject to the constraint that each job should complete not long after its completion under fair sharing, i.e., satisfying the constraint of optimization problem (1).

One possible way to satisfy the constraint is to interpret the completion time of each job under fair sharing (i.e., $\bar{f}_j$) as the "*deadline*" of the job, and seek a scheduling discipline to minimize the maximal lateness. A commonly used optimal discipline is to finish the most urgent jobs first, i.e., use Earliest Deadline First (EDF) [24]. This EDF-like algorithm not only ensures the predictable performance comparable to fair sharing, but also shortens the response times by prioritizing the job executions, rather than enforcing fair allocations at all time.

The question is how this EDF-like algorithm performs compared to the optimal Shortest Remaining Time First (SRPT) in terms of minimizing the mean response time. Interestingly, we find that this EDF-like algorithm *mimics* the behavior of SRPT to some extent. To see this, we start to consider fair sharing, with which jobs are allocated approximately the same amounts of cluster resources, and are processed at the similar progress. As a result, those having the least remaining work likely complete the first, meaning that they are likely assigned the earliest deadline. These jobs are then prioritized under EDF, which resembles Shortest Remaining Time First. Therefore, the mean response time of this EDF-like algorithm should be near-optimal, close to SRPT.

To summarize, our intuition suggests a promising approach that likely minimizes the response time while achieving the predictable performance: *preferentially offer resources to jobs in ascending order of their completion times under fair sharing*.



(a) Job-2 completes before job-1. (b) Job-1 completes before job-2.

Fig. 2: Frequent predictions are needed to maintain the job completion order under a fair scheduler. Each block corresponds to a task running on a slot. (a) At time 0, job-2 receives 2 slots and is predicted to complete first. (b) The arrival of job-3 at time 4 s reverses the completion orders of job-1 and job-2.

D. Challenges

To implement this approach in real systems, a cluster scheduler needs to maintain the order of job completion times under fair sharing. This requires a scheduler to simulate fair sharing as a *reference system*: upon a job arrival or completion, the scheduler updates the reference system with a new resource allocation, based on which it predicts the completion time of each job.

While the state-of-the-art profiling techniques are fairly accurate in predicting the job completion time given an allocation [12], [25]–[28], frequent predictions are expensive. Yet, this seems to be inevitable if the reference system naively simulates the behaviors of prevalent fair schedulers [3]–[7], where max-min fairness is maintained at all time. To see this, we refer to the example in Fig. 2a, where two jobs arrive to a 3-slot cluster at time 0. Job-1 has one task running 9 s on a slot; job-2 has four tasks, each running 4 s. As shown in Fig. 2a, to achieve max-min fairness, job-1 receives one slot while job-2 receives two. Job-2 is predicted to complete before job-1. Now suppose that job-3 arrives at time 4 s, with two tasks each running 4 s. As shown in Fig. 2b, to maintain max-min fairness, each job is allocated one slot, and this time job-1 is predicted to complete first. In general, to maintain max-min fairness, resources are reallocated upon a job arrival or completion, which may result in a different job completion order. To keep track of the latest completion order, predictions must be performed to each job frequently.

Frequent predictions can also increase the sensitivity in the prediction errors. For example, consider two jobs consisting of an equal number of tasks with the same runtime. Under fair sharing, both jobs receive an equal share of cluster resources *regardless* of the arrivals or completions of the other job, and hence finish at the same time. Ideally, our approach can prioritize one job over the other compared to fair sharing, i.e., the one that is prioritized can be sped up without slowing down the other. However, due to the prediction errors, it is possible that each time a new prediction is made, the job that was previously believed to complete earlier than the other is now predicted to finish later, and is assigned a lower service priority. Thus, frequent predictions may result in frequent *priority reverses*, forcing two jobs to *time-multiplex* the cluster, which is inefficient as no one can get speedup.

We see from the discussions above that the key challenge to

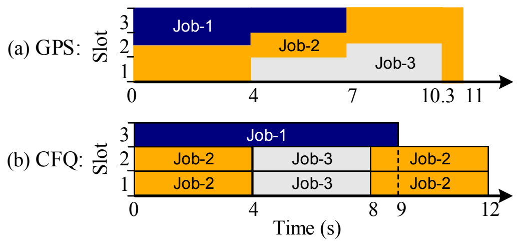


Fig. 3: Illustration of CFQ in the example of Fig. 2.

implement our approach is to maintain the job completion order under fair sharing without frequent job predictions or profiling. We shall answer this challenge in the following sections.

IV. Cluster Fair Queueing

In this section, we develop a new scheduling algorithm, Cluster Fair Queueing (CFQ), that prioritizes the execution of jobs that complete the earliest under an idealized fair sharing policy. CFQ is efficient to implement in that each job is profiled once, only upon its arrival. We show that CFQ achieves the predictable performance comparable to fair sharing.

A. Cluster Fair Queueing

Idealized model. We have shown in the previous section that maintaining the job completion orders under the prevalent fair sharing policies [3]–[7] is expensive. This motivates us to turn to an *idealized fair sharing* in a simplified, hypothetical model in which a task is assumed to be *infinitely divisible* and can be sped up using more than one slot. That is, if we use n slots to service a task that runs l time units on one slot, we can finish the task in l/n time units. Under the idealized model, fair sharing simply allocates each job the *same* number of slots at all time. Specifically, suppose that N jobs run in an M-slot cluster. Under the idealized fair scheduler, each job is allocated M/N slots, and is serviced at the same progress.

Cluster Fair Queueing. Our algorithm simulates the idealized fair sharing as a reference system in the background. Whenever a compute slot becomes available in the *real system*, the algorithm offers that slot to a task of the backlogged job that completes the earliest in the reference system. As a running example, Fig. 3 illustrates the algorithm in the previous example of Fig. 2. At time 0, there are only two jobs, where job-1 completes earlier than job-2 under the idealized fair sharing (Fig. 3a), and is assigned the highest priority in the real system. Since job-1 has only one task and takes one slot, the remaining two slots go to job-2 (Fig. 3b). Later at time 4 s, job-3 arrives, who completes earlier than job-2 under the idealized fair sharing (Fig. 3a). Job-3 is assigned a higher priority than job-2 in the real system, and the two slots that were previously used to service job-2 are offered to job-3 when they become available (Fig. 3b). The allocation remains until job-3 completes at 8 s, after which job-2 regains the two slots and resumes execution.

Maintaing the idealized fair sharing as a reference system is preferred than simulating the prevalent fair schedulers. First, with the idealized fair sharing, *perfect fairness* is achieved at all time: each job is allocated the same number of slots and is serviced at the same progress. This is not possible with the prevalent fair schedulers, where different jobs may receive different number of slots (e.g., job-1 and job-2 in Fig. 2a). Also, since tasks are indivisible in practice and cannot be preempted during the execution, a newly arrived job may not be assigned slots immediately, which results in temporary unfairness. Therefore, compared to the prevalent fair schedulers, the idealized fair sharing ensures better predictable job performance.

Moreover, unlike the prevalent fair schedulers, the idealized fair sharing can be efficiently maintained without the need for frequent prediction of job completion orders, thus avoiding the implementation issues described in Sec. III-D. Because jobs are allocated the same number of slots and are serviced at the same progress at all time under the idealized fair scheduler, the completion orders of two jobs, once determined, *do not change* over time. More precisely, for a newly arrived job, if the processing time it requires—measured by the *slot-time* production—is less (greater) than the remaining slot-time required by an existing job, the new one completes earlier (later) than the existing one, *regardless* of the future job arrivals or completions. This consistency in job completion orders makes frequent predictions and updates no longer needed. We shall show in Sec. IV-C that it is sufficient to track the job completion orders by profiling the required slot-time of each job only upon its arrival.

We refer to our algorithm as Cluster Fair Queueing (CFQ) as the idealized fair scheduler resembles Generalized Processor Sharing (GPS) in the classical fair queueing [16], [17]. GPS was initially proposed as an idealized algorithm to fairly share the link bandwidth of a router among different flows. In GPS, the equivalent of a job is a *flow*; the equivalent of a task is a *packet*. GPS can be accurately implemented as a practical packet scheduler by WFQ [16], [17], where the packet that completes the earliest in GPS is scheduled first.

CFQ vs. fair queueing. CFQ differs from the traditional fair queueing algorithms in both the design philosophy and algorithmic behaviors. First, traditional fair queueing algorithms such as WFQ [16] and PGPS [17] aim to achieve fair bandwidth allocation by closely tracking GPS in practice. To do so, these algorithms track the packet completion times in GPS so as to *interleave* the service of packets (tasks) across flows (jobs) as much as possible. In contrast, CFQ aims to speed up the completion of jobs by prioritizing their executions so as to service all the tasks of each job *in batches*.

Second, traditional fair queueing algorithms are the *single-server* schedulers, where packets (tasks) are sequentially serviced on an outgoing link of a router, one at a time.[1] In contrast, CFQ is a *multi-server* scheduler, where multiple tasks run on a cluster of machines in parallel. Compared to single-server scheduling, multi-server scheduling is more complex, and is technically more challenging to analyze [18], [19]. In

[1]Some fair queueing algorithms are multi-server schedulers, e.g., [18], [19]. These algorithms aim to achieve fair bandwidth allocations across multiple links by closely tracking multi-server GPS in the unit of packets (tasks).

TABLE I: Summary of important notations and definitions.

M	The number of slots available in a cluster
$\bar{f}_j$	The job completion time in GPS
f_j	The job completion time in CFQ
L_j	The slot-time of job-j
$L_{\max}$	The maximal slot-time of all jobs
$l_{\max}$	The maximal task runtime
a_j	The arrival time of job-j
e_j	The ending time of the slowdown period of job-j

addition, many nice properties for single-server scheduling are no longer held in the context of multi-server scheduling. For example, it is well known that given an input workload on a single server, no matter what scheduling algorithms are used, as long as the algorithms are *work conserving*—meaning that the server keeps busy if there is a backlogged packet (task)—the *busy periods* (i.e., the time interval during which the server is busy) of these algorithms *overlap*. This property plays the key role in the performance analysis of fair queueing algorithms [16], [17]. However, multi-server scheduling does not have this property. As shown in Fig. 3, the busy periods of CFQ and idealized fair sharing do not overlap, though both algorithms are work conserving.

To summarize, CFQ adopts a different design philosophy from the traditional fair queueing algorithms, and is technically more difficult to analyze. Nevertheless, we next show that CFQ is more than a simple heuristic.

B. Delay Analysis

CFQ achieves the predictable job performance. With CFQ, a job is guaranteed to complete within a small constant time after its completion in GPS. Our analysis critically focuses on the *slowdown period* of a job. In particular, we say a job is *slowed down* at time t if it has a *backlogged* task waiting for service. In other words, at any moment in the slowdown period, the job could have run more tasks if receiving more slots. Because slowdown delays the job completion, bounding the timespan of the slowdown periods is the key to analyzing the longest possible delay.

Suppose N jobs run on an M-slot cluster. Let jobs be indexed in ascending order of the start time at which the first task starts running in the real system, e.g., job-j is the j^{th} job that is allocated slots in CFQ. For job-j, let L_j be the required *slot-time*, which is the time to run all tasks of the job sequentially using *one* slot. Slot-time characterizes the processing work required by a job. Let $L_{\max}$ denote the *maximal* slot-time, and $l_{\max}$ the maximal *task runtime* on one slot. Finally, let f_j denote the completion time of job-j in CFQ, and $\bar{f}_j$ the GPS completion time. Table I summarizes the notations used in the analysis.

We establish the constant delay bound of CFQ through the following theorem.

Theorem 1 (Constant delay bound): With CFQ, a job is guaranteed to complete within a constant time after its completion in GPS, i.e., for each job-j, we have

$$f_j - \bar{f}_j \leq 2l_{\max} + L_{\max}/M. \tag{2}$$

Proof: Unless otherwise stated, we refer to the job execution in the real system using CFQ. For each job-j, we consider its slowdown period. Let a_j be the arrival time of job-j. Depending on the number of available slots at time a_j, job-j is either slowed down, or allocated a sufficient number of slots to run all tasks right after the arrival. In particular, let e_j be the time when the slowdown period of job-j ends. We have $a_j < e_j$ if job-j experiences slowdown, and $a_j = e_j$ otherwise. In either case, after the slowdown period, job-j runs all the backlogged tasks in parallel, and is guaranteed to complete after at most the maximal task runtime, i.e.,

$$f_j \leq e_j + l_{\max}. \tag{3}$$

Therefore, to bound the job completion time, it is critical to analyze when the slowdown period ends (i.e., e_j). We next consider the following two cases.

Case 1: Job-j experiences no slowdown, i.e., $a_j = e_j$. We consider the reference GPS system. To finish job-j as quickly as possible, GPS should service the job using all M slots. Therefore, job-j completes in GPS no earlier than

$$\bar{f}_j \geq a_j + L_j/M = e_j + L_j/M. \tag{4}$$

Subtracting (4) from (3), we have

$$f_j - \bar{f}_j \leq l_{\max} - L_j/M \leq l_{\max}.$$

Case 2: Job-j is slowed down right after the arrival, i.e., $a_j < e_j$. During the slowdown period $[a_j, e_j]$, all the slots are busy. Let job-i be the one with the *smallest index* whose arrival starts a busy period until e_j, i.e., all slots keep busy during $[a_i, e_j]$. Because job-i is such a job with the smallest index, there must be an available slot right before its arrival at a_i. We next study the job executions in $[a_i, e_j]$ to bound e_j.

While CFQ preferentially offers slots to jobs in ascending order of their GPS completion times, jobs may start services out of order due to the dynamic arrivals. In particular, a job that completes before job-j in GPS may arrive late, after job-j starts in CFQ. Let $\mathcal{A}$ be the set of all these jobs, i.e., $\mathcal{A} = \{k \mid k > j \text{ and } \bar{f}_k \leq \bar{f}_j\}$. On the other hand, a job that completes after job-j in GPS may start earlier in CFQ, before job-j arrives. Let job-m be such a job that is serviced *the most recently*. That is, m is the *largest* integer satisfying both $i \leq m \leq j-1$ and $\bar{f}_m > \bar{f}_j$, i.e., $\bar{f}_m > \bar{f}_j \geq f_k$ for all $m < k < j$. In other words, job-m completes after jobs $m+1, \ldots, j$ in GPS, but is allocated slots before all these jobs in CFQ. We further consider the following two sub-cases.

Case 2-1: No such a job-m exists. We start to analyze $\bar{f}_j$, the GPS completion time of job-j. Let $\mathcal{B} = \{k \mid i \leq k \leq j\}$. By definition, in CFQ, jobs in $\mathcal{A}$ and $\mathcal{B}$ start services no earlier than job-i. Because CFQ is work-conserving, and there is a slot available right before job-i arrives at time a_i, jobs in $\mathcal{A}$ and $\mathcal{B}$ must arrive *no earlier* than a_i—otherwise, CFQ would have allocated slots to them before job-i. On the other hand, by definition, jobs in $\mathcal{A}$ and $\mathcal{B}$ complete *no later* than job-j in GPS. Therefore, starting from the moment job-i arrives, GPS has finished all the jobs in $\mathcal{A} \cup \mathcal{B}$ by the time job-j completes. The earliest possible GPS completion time of job-j is

$$\bar{f}_j \geq a_i + \tfrac{1}{M} \textstyle\sum_{k \in \mathcal{A} \cup \mathcal{B}} L_k. \tag{5}$$

We next bound f_j, the completion time of job-j in CFQ. By (3), this is equivalent to bounding e_j, the time when the slowdown period of job-j ends. We make the following two observations regarding the slowdown period $[a_j, e_j]$ of job-j.

1) All slots are busy in $[a_j, e_j]$;
2) Jobs that are allocated slots by CFQ during $[a_j, e_j]$ complete *no later* than job-j in GPS.

Based on these observations, we establish a case where e_j reaches the maximum. Since jobs in $\mathcal{A}$ and $\mathcal{B}$ complete no later than job-j in GPS, it is possible that these jobs are *all* serviced in $[a_i, e_j]$. In addition, by the time job-i arrives at a_i, at most $M-1$ slots are busy servicing tasks of other jobs. Therefore, during the time interval $[a_i, e_j]$, CFQ has completed, at most, all the jobs in $\mathcal{A}$ and $\mathcal{B}$ along with $M-1$ tasks requiring a maximal runtime. Since all the M slots are busy in $[a_i, e_j]$, finishing these workloads takes at most

$$e_j \leq a_i + \tfrac{1}{M} \left[\textstyle\sum_{k \in \mathcal{A} \cup \mathcal{B}} L_k + (M-1) l_{\max} \right]. \tag{6}$$

Plugging (6) into (3) and subtracting (5), we have

$$f_j - \bar{f}_j \leq l_{\max} + \tfrac{M-1}{M} l_{\max} < 2 l_{\max}.$$

Case 2-2: Such a job-m exists. Let $\mathcal{C} = \{k \mid m < k \leq j\}$. By definition, jobs in $\mathcal{A}$ and $\mathcal{C}$, though completing earlier than job-m in GPS, are serviced no earlier than job-m in CFQ. These jobs must have not yet arrived before job-m is allocated slots—otherwise CFQ would have serviced them before job-m. We then have

$$\min_{k \in \mathcal{A} \cup \mathcal{C}} \{a_k\} \geq b_m, \tag{7}$$

where b_m is the first time when job-m is allocated slots in CFQ. This suggests that since b_m, GPS has completely serviced, *at least*, all jobs in $\mathcal{A} \cup \mathcal{C}$ by the time job-m finishes, i.e.,

$$\bar{f}_j \geq b_m + \tfrac{1}{M} \textstyle\sum_{k \in \mathcal{A} \cup \mathcal{B}} L_k. \tag{8}$$

We next analyze f_j, the completion time of job-m in CFQ. With a similar argument in Case 2-1, we see that during time interval $[b_m, e_j]$, CFQ may have completed all the jobs in $\mathcal{A}$ and $\mathcal{C}$, along with $M-1$ tasks requiring a maximal runtime. In addition, unlike Case 2-1, job-m is allocated slots at b_m, and may also complete before e_j. Finishing all these works using all M slots takes at most

$$e_j \leq b_m + \tfrac{1}{M} [\textstyle\sum_{k \in \mathcal{A} \cup \mathcal{C}} L_k + L_m + (M-1) l_{\max}]. \tag{9}$$

Plugging (9) into (3) and subtracting (8), we have

$$f_j - \bar{f}_j \leq l_{\max} + \tfrac{M-1}{M} l_{\max} + L_m / M < 2 l_{\max} + L_{\max} / M.$$

■

Remarks. We make two remarks on Theorem 1. First, the delay bound is a small constant in real systems. Recall that $L_{\max}$ is the maximal slot-time of a job, we interpret $L_{\max}/M$ as the time required to run a job *alone* in a cluster, using all M slots. In large clusters like datacenters, the number of slots is usually greater than the degree of parallelism of a job, meaning that running a job alone in a cluster takes approximately the same time as running a task on one slot. The delay bound is therefore approximately the runtime of a few tasks.

Second, Theorem 1 states that CFQ achieves the predictable performance without delaying a job long after its GPS completion, but the job may complete much faster than in GPS, because CFQ mimics the behaviors of SRPT to minimize the mean response time. We shall validate its performance optimality through experiments in Sec. V.

C. Implementation

Virtual time implementation. To implement CFQ as a cluster scheduler in real systems, we need to maintain the job completion order in the reference GPS system efficiently. This is conceptually similar to fair queueing algorithms tracking the packet completion order in GPS, where the *virtual time* implementation applies [16], [17]. Similarly, we define virtual time $V(t)$ as a function of real time t evolving as follows:

$$\begin{aligned} V(0) &= 0, \\ \tfrac{\mathrm{d}}{\mathrm{d}t} V(t) &= M / \bar{N}_t. \end{aligned} \tag{10}$$

Here, $\bar{N}_t$ is the number of active jobs in GPS at time t, and $M/\bar{N}_t$ is the service rate (slots per unit time) each job receives in GPS at time t. Thus, $V(t)$ can be interpreted as increasing at the marginal rate at which jobs receive services in GPS.

When a job-j arrives at time a_j, CFQ profiles its required slot-time L_j, using the state-of-the-art job profiling techniques (e.g., [12], [25]–[28]). CFQ then associate the job with *virtual finish time* F_j, which indicates the virtual time at which the job completes in GPS, i.e.,

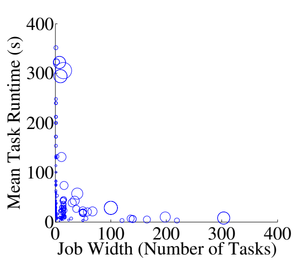
$$F_j = V(a_j) + L_j. \tag{11}$$

The virtual time of a job, once calculated, requires no update in the future. This is because the one with the smallest F_j will always complete first under GPS. Therefore, whenever a slot becomes available, CFQ offers it to the job with the smallest virtual finish time. The scheduling complexity is $O(\log N)$.

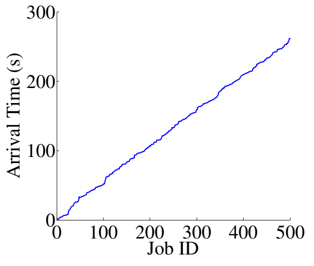
Job profiling. The virtual time implementation allows CFQ to profile each job only once, upon the arrival. Because job profiling is needed by any performance-optimal schedulers, e.g., [10], [12]–[14], CFQ incurs the least profiling overhead among these schedulers.

There are a wide spectrum of job profiling techniques that CFQ can use. For example, for recurring jobs that repeatedly run on the same or similar datasets, the performance can be accurately quantified from previous runs [12], [25], [26]. For non-recurring jobs with no historical information, accurate performance prediction models can also be quickly built by sampling the job behavior on a small subset of data [27], [28].

Despite the recent advances in job profiling techniques, profiling errors remain unavoidable. We next show in the experiments that CFQ is insensitive to these errors, and can achieve a salient performance improvement even with a naive, inaccurate profiling algorithm.



(a) Job size and task runtime.



(b) Job arrival times.

Fig. 4: Illustration of job profiles in the workloads sampled from the Google traces [20].

V. Evaluation

We have implemented CFQ as a pluggable Spark scheduler and evaluated its performance in a 100-node cluster. For performance study at a larger scale, we resort to trace-driven simulations over the production traces in a Google's cluster.

A. Setup

Cluster. We ran experiments in a 100-node Amazon EC2 cluster. For each node, we used an `m4.large` EC2 instance and configured it to host 2 compute slots. In total, the cluster consists of 200 cores and 800 GB memory.

Workload. We ran workloads based on traces from Google [20]. The traces consist of a mix of experimental and production jobs running in a cluster of 12K machines in one month. Given the large scale of the traces, we sampled 500 jobs in a one-hour window, and used them as the input workloads. Our sampling retains the original job profiles, including the distribution of the inter-arrival times of jobs, the number of tasks a job has, and the average task runtime of a job. Fig. 4 depicts the job profiles of the workloads. In Fig. 4a, the placement of a circle indicates the number of tasks a job has—known as the *job width*—and its average task runtime. The size of a circle is proportional to the required slot-time of a job. We see from Fig. 4a that the workloads are dominated by narrow jobs with no more than 5 tasks. Unlike wide jobs, narrow jobs may not necessarily have short-running tasks—some run hundreds of seconds. We also plot the job arrival times in Fig. 4b, from which we observe a uniform jobs arrival pattern.

Baselines. Throughout the experiments, we compare CFQ against the two built-in Spark schedulers: FIFO and fair scheduler (FAIR) [4], which represent the two most widely deployed schedulers in today's datacenters. In addition, we have implemented SRPT as an aggressive baseline that minimizes the mean response time without any fairness concern. We shall highlight the near-optimal performance of CFQ by showing that it achieves the similar job response time to SRPT.

Metrics. We consider two metrics: *slowdown* and *normalized response time*. In particular, we define *slowdown* for each job as the response time due to a scheduler normalized by the *minimum* response time if the job were running *alone* in the cluster:

$$\text{Slowdown} = \frac{\text{Compared Response Time}}{\text{Min Response Time If Running Alone}}.$$

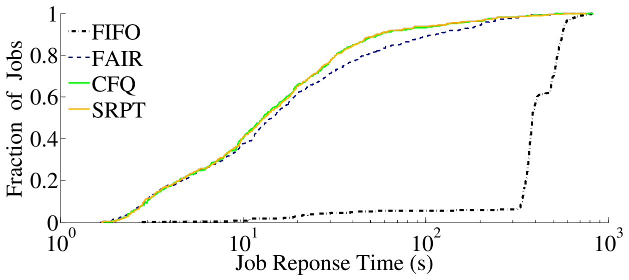


Fig. 5: CDF of job response times using different schedulers.

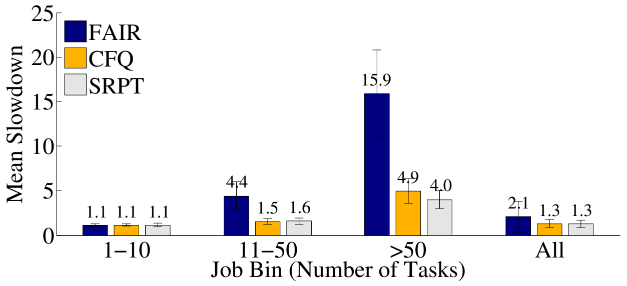


Fig. 6: Mean *slowdown* binned by job widths.

In addition, we compare a scheduler against CFQ by measuring its job response time normalized by that under CFQ, i.e.,

$$\text{Normalized Response Time} = \frac{\text{Compared Response Time}}{\text{Response Time under CFQ}}.$$

That is, compared to the given scheduler, a job runs slower (faster) using CFQ if its normalized response time is less (greater) than 1.

B. Deployment Results

Our experiments start with accurate job profiling. We shall show later that CFQ is highly insensitive to profiling errors.

Response time. Fig. 5 shows the distribution of job response times using the four schedulers. As expected, FIFO performs the worst. Because the newly arrived jobs have to wait for the existing ones to complete, an elephant job delays all the late comers, among which many are mice jobs. These mice will soon complete after the elephant, resulting in a big jump of the FIFO curve in Fig. 5. This problem is avoided using the other three schedulers with smooth CDF curves. Among them, CFQ and SRPT minimize the job response times.

Given the poor performance of FIFO, we exclude it from further comparisons. To better understand the performance of the other three schedulers, we divide jobs into 3 bins based on their widths: *narrow* (1-10 tasks), *medium* (11-50 tasks), and *wide* (>50 tasks). Fig. 6 compares the mean *slowdown* of the three schedulers in 3 bins, where the error bar measures one standard deviation. We see from the figure that narrow jobs experience almost no slowdown under all three schemes. These jobs consist of a small number of tasks, all of which can be accommodated in *one wave* using a fair share of the slots. Given their fast completion under fair sharing, these jobs are likely prioritized by CFQ. In addition, because narrow jobs typically have small slot-time (Fig. 4a), they are prioritized by

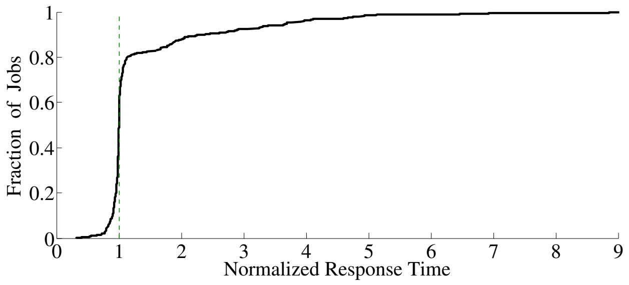


Fig. 7: Distribution of the normalized response time using FAIR. A job is sped up using CFQ if its normalized response time is greater than 1; otherwise, the job is slowed down.

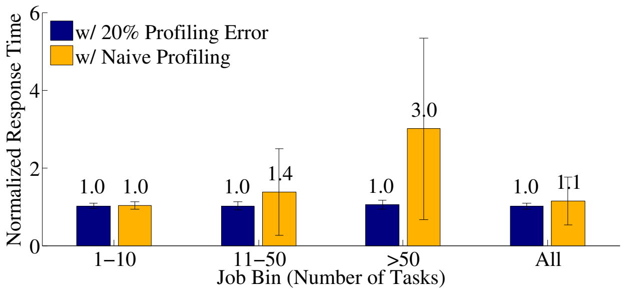


Fig. 8: CFQ is insensitive to profiling errors. For each job, we measured its response time using CFQ with inaccurate profiling, normalized by that with accurate profiling.

SRPT as well. However, the performance of wider jobs varies using different schedulers. Compared to CFQ, FAIR results in 2.9× and 3.2× longer response time on average for medium and wide jobs, respectively. This is because fair allocations cannot accommodate all these jobs' tasks at a time, forcing them to run in multiple waves. CFQ eliminates this inefficiency by preferentially offering slots to jobs that complete earliest in GPS, which reduces the mean *slowdown* by 40%. That CFQ exhibits almost the same *slowdown* performance as SRPT in all 3 bins highlights the near-optimal performance of our design.

Fairness. We have shown in Theorem 1 that with CFQ, jobs are guaranteed to complete within a small constant time after completion in GPS. Given that GPS is an idealized scheduler, in real systems FAIR is deployed to achieve the predictable performance. A natural question is: can jobs expect a similar delay guarantee with respect to FAIR using CFQ? We measured the normalized response time of each job using FAIR and depict the distribution in Fig. 7. More than 40% of jobs have normalized response time greater than 1, meaning that they are sped up using CFQ. The average speedup is over 75%. The price paid is an average of 7% *slowdown* of the other 30% of jobs, among which less than 10 jobs suffer from more than 20% longer response time. All these results indicate that CFQ does not delay jobs long after their completion using FAIR.

Sensitivity to profiling errors. So far, our evaluations are based on accurate job profiling. However, profiling errors are inevitable in practice. How do the errors impact the performance of CFQ? We answer this question through two experiments. Given that the state-of-the-art workload profiling techniques can already achieve less than 20% error [27], in the first experiment we evaluated CFQ with a uniform prediction error up to 20%. In particular, for job-j with slot-time L_j, CFQ profiles its slot-time as $\tilde{L}_j$, a uniform random variable in $[0.8L_j, 1.2L_j]$. For each job, we measured the response time using CFQ with up to 20% profiling error, and normalized it by that with accurate profiling. Fig. 8 shows the normalized response time binned by job widths. We observe that, despite the profiling error, CFQ achieves almost the same job response times as that of accurate profiling in all bins. Intuitively, CFQ uses the slot-times only to determine the job completion order in GPS: a small prediction error does not result in a different order.

In the second experiment, we stress-tested CFQ's insensitivity of profiling errors. Specifically, we used a *naive profiling algorithm* in CFQ that estimates the slot-time of a newly submitted job-j as $\tilde{L}_j = n_j \times \tilde{l}_j$, where n_j is the number of tasks job-j has, and $\tilde{l}_j$ is the *cumulative average* of task runtime up to job-j's arrival. This simple profiling is highly inaccurate. Similar to the previous experiment, we measured the job response time using naive profiling by that using accurate profiling. Fig. 8 shows the normalized response time binned by job widths. To our surprise, even with such a naive profiling algorithm, the mean response time is increased by only 10% compared to the perfect prediction. In particular, inaccurate profiling hurts wide jobs the most, whereas narrow jobs are the least affected. We see from the two experiments that CFQ is highly insensitive to profiling errors, even with naive profiling.

Micro-benchmark. Our previous experiments highlight the near-optimal performance of CFQ close to SRPT. We now use micro-benchmark to contrast the different behaviors of the two schedulers on a small-sized cluster in a more controlled manner. In particular, we ran two types of jobs in a cluster of 10 EC2 `m4.large` instances: an elephant job consisting of 20 tasks each running 2 s, and a mice job consisting of 8-12 tasks running 0.8–1.2 s. Jobs arrived to the cluster uniformly over time. We ran five micro-benchmarks with one elephant and a variable number of mice. Fig. 9 compares the measured response time of the elephant in five micro-benchmarks, using CFQ and SRPT, respectively. We find that with SRPT, the response time of the elephant increases linearly with the increasing number of mice—the elephant is starved if mice keep arriving over time. In contrast, CFQ isolates the performance of the elephant from the others, with predictable completion time that is independent to the dynamic arrival of mice.

C. Simulation Results

To understand performance at a larger scale, we use trace-driven simulations. We simulated a 1000-node cluster with 8K slots and fed it the workloads sampled in the first 20 hours from the Google traces [20]. In total, we ran 543K tasks across 17K jobs using different schedulers. For each job, we measured its *slowdown*. Fig. 10 shows the distribution of the *slowdown* binned by job widths. Boxes depict 25th, 50th, and 75th percentiles; whiskers depict 5th and 95th percentiles. We observe the similar performance trend as in our Spark

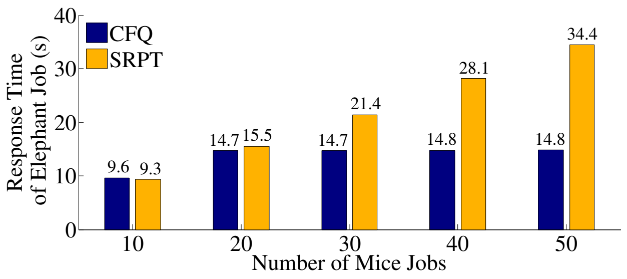


Fig. 9: The response time of the elephant job when running with a variable number of mice, using CFQ and SRPT, respectively. CFQ does not starve the elephant, but SRPT does.

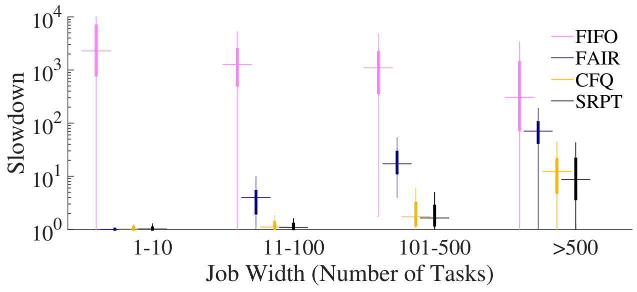


Fig. 10: Distribution of job *slowdown* using the four schedulers. Boxes depict 25th, 50th, and 75th percentiles; whiskers depict 5th and 95th percentiles.

deployment. In particular, FIFO performs the worst in all job bins, resulting in orders of magnitude longer response time. Among the other three schedulers, SRPT and CFQ simply outperform FAIR with an order of magnitude smaller *slowdown* in all bins but the narrow jobs. When it comes to SRPT and CFQ, while SRPT wins in performance, CFQ is a close follower that ties SRPT in all bins but jobs with more than 500 tasks. Our simulation results further reinforce the near optimal performance of CFQ.

VI. Conclusion

In this paper, we propose a new scheduler, Cluster Fair Queueing (CFQ), which aims to minimize the mean response time of jobs while achieving the predictable performance. CFQ preferentially offers compute slots to jobs that complete the earliest in GPS, an idealized fair scheduler that enforces perfect fairness in a cluster at all time. We demonstrate that with CFQ, jobs are guaranteed to complete within a small constant time after their completion in GPS. We have implemented CFQ in Spark and evaluated its performance on a 100-node Amazon EC2 cluster and in simulations with traces from a Google's cluster. Both implementation and simulation results show that CFQ is able to achieve the near-optimal job response time as SRPT, and is insensitive to job profiling errors.

Acknowledgement

This work was supported in part by grants from RGC under the contracts 615613, 16211715 and C7036-15G (CRF), a grant from NSF (China) under the contract U1301253, as well as a Microsoft Research Asia Collaborative Research Grant.

References

[1] 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 *USENIX NSDI*, 2012.

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *ACM SoCC*, 2013.

[3] "Hadoop Fair Scheduler," http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html, 2015.

[4] "Spark Job Scheduling," https://spark.apache.org/docs/1.6.0/job-scheduling.html, 2016.

[5] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *USENIX NSDI*, 2011.

[6] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *ACM EuroSys*, 2013.

[7] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica, "Hierarchical scheduling for diverse datacenter workloads," in *ACM SoCC*, 2013.

[8] J. Tan, X. Meng, and L. Zhang, "Delay tails in mapreduce scheduling," in *ACM SIGMETRICS*, 2012.

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

[10] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *ACM SIGCOMM*, 2014.

[11] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *ACM SPAA*, 2011.

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

[13] M. Lin, L. Zhang, A. Wierman, and J. Tan, "Joint optimization of overlapping phases in mapreduce," *Perf. Eval.*, vol. 70, no. 10, pp. 720–735, 2013.

[14] Y. Wang, J. Tan, W. Yu, L. Zhang, and X. Meng, "Preemptive reducetask scheduling for fair and fast job completion," in *USENIX ICAC*, 2013.

[15] M. Pastorelli, D. Carra, M. Dell'Amico, and P. Michiardi, "Hfsp: Bringing size-based scheduling to hadoop," 2015.

[16] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM SIGCOMM*, 1989.

[17] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, no. 3, pp. 344–357, 1993.

[18] J. M. Blanquer and B. Özden, "Fair queuing for aggregated multiple links," in *ACM SIGCOMM*, 2001.

[19] K.-K. Yap, N. McKeown, and S. Katti, "Multi-server generalized processor sharing," in *ACM Intl. Teletraffic Congrass (ITC)*, 2012.

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

[21] L. Schrage, "A proof of the optimality of the shortest remaining processing time discipline," *Oper. Res.*, vol. 16, no. 3, pp. 687–690, 1968.

[22] K. Pruhs, J. Sgall, and E. Torng, "Online scheduling," in *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 2004.

[23] E. J. Friedman and S. G. Henderson, "Fairness and efficiency in web server protocols," in *ACM SIGMETRICS*, 2003.

[24] G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011.

[25] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou, "Wave computing in the cloud," in *Proc. ACM HotOS*, 2009.

[26] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: automatic resource inference and allocation for mapreduce environments," in *ACM ICAC*, 2011.

[27] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *USENIX NSDI*, 2016.

[28] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Bridging the tenant-provider gap in cloud services," in *ACM SoCC*, 2012.