Performance-Aware Fair Scheduling: Exploiting Demand Elasticity of Data Analytics Jobs

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

***Abstract*—Efficient resource management is of paramount importance in today's production clusters. In this paper, we identify the *demand elasticity* of data-parallel jobs. Demand elasticity allows jobs to run with a significantly less amount of resources than they ideally need, at the expense of only a modest performance penalty. Our EC2 experiment using popular Spark benchmark suites confirms that running a job using 50% of demanded slots is sufficient to achieve at least 75% of the ideal performance. We show that such an elasticity is an intrinsic property of data-parallel jobs and can be exploited to speed up average job completion. In this regard, we propose Performance-Aware Fair (PAF) scheduler to identify the demand elasticity and use it to improve the average job performance, while still attaining near-optimal isolation guarantee close to fair sharing. PAF starts with a fair allocation and iteratively adjusts it by transferring resources from one job to another, improving the performance of resource-taker without penalizing resource-giver by a noticeable amount. We implemented PAF in Spark and evaluated its effectiveness through both EC2 experiments and large-scale simulations. Evaluation results show that compared with fair allocation, PAF improves the average job performance by 13%, while penalizing resource-givers by no more than 1%.**

I. Introduction

Cluster scheduler continues to be a critical component to data-parallel clouds, in which diverse coexisting jobs from many users and applications contend for resources in a shared environment. As the data volume increases and the demand for analytics jobs surges, today's production clusters are frequently resource-constrained. Therefore, *efficient resource management* comes as the top priority for cluster schedulers.

Prevalent cluster schedulers [1]–[7] let jobs estimate the amount of resources they need. For example, a Spark/Hadoop job specifies the *degree of parallelism*, which is the number of compute slots it needs to run parallel tasks in the *ideal case*. Given this ideal demand, the scheduler follows the scheduling priority and aggressively allocates each job *as many slots as possible*, e.g., FIFO and max-min fair allocations.

However, aggressively allocating a job as many resources as it ideally needs often does not translate into salient performance benefits. In fact, we show through experimental studies using recent Spark benchmark workloads (Sec. II) that data analytics jobs tend to have *elastic* resource demands. Such an elasticity allows a job to run with significantly less amount of resources than it ideally needs, at the expense of only a *marginal* performance penalty. For example, in our experiment, we observed only 5% performance degradation when running a KMeans job in Spark MLlib [8] using half of the ideal resource demands. Curious about this result, we performed deep-dive analyses and confirmed that such an elasticity is by no means an accident, but a general trend for a wide range of data-parallel jobs—mainly due to the uneven task runtime [9] and the JVM warm-up overhead recently uncovered by Lion et al. [10]. Such a demand elasticity, if well exploited, can dramatically improve the scheduling performance.

Unfortunately, production cluster schedulers [1]–[4] remain *agnostic* to the demand elasticity of data-parallel jobs. These schedulers settle for *isolation guarantee* as the primary objective and seek to maintain *fair allocations* at all time. However, blindly enforcing fair allocations is *neither necessary nor efficient*. Due to the prevalence of demand elasticity, there are ample opportunities to serve jobs using much fewer resources than fair allocations, without noticeable performance penalty. The saved resources can then be allocated to other resource-hungry jobs to achieve salient performance improvement.

Motivated by this intuition, in this paper, we propose Performance-Aware Fair scheduling (PAF) to identify and exploit the demand elasticity of data-parallel jobs. Our objective is to opportunistically improve the average job performance for fast job completion, while still achieving near-optimal isolation guarantee close to fair sharing. To identify demand elasticity, PAF builds on the recent advances in performance prediction frameworks [11]–[14] to profile job performance models against the number of allocated slots. In particular, for *recurring* jobs that run repeatedly over similar datasets, the performance models can be faithfully learned from historical traces [11], [12]; for *non-recurring* jobs, techniques such as Ernest [13] can efficiently train the performance model with small samples of input data.

Once the job performance model has been obtained, PAF computes, for each job, the fair share and uses it as the baseline allocation to provide isolation guarantee. PAF then seeks to optimize the average job performance by judiciously adjusting the baseline allocation without compromising isolation guarantee by a small degree pre-specified by the cluster operator. To do so, PAF identifies two jobs, one *indifferent* about giving up some compute slots due to demand elasticity and the other *eager* to receive more slots to gain salient performance benefits. The former then acts as a *resource-giver* and yields its slots to the latter, which we call a *resource-taker*. PAF iteratively identifies a resource-giver and a resource-taker and transfers slots from the former to the latter, until an equilibrium has been achieved. We show that this simple algorithm maximizes the

average job performance while providing isolation guarantee close to fair sharing.

We have prototyped PAF as a pluggable scheduler in Spark, and evaluated its effectiveness through both testbed experiments and large scale simulations driven by synthetic workloads. Our cluster deployment on Amazon EC2 with 64 `m4.large` instances shows that, compared with current fair scheduler, PAF can improve the average job performance by 27% while ensuring near-optimal isolation guarantee. Our large-scale simulations further confirm that, by exploiting demand elasticity, PAF improves the average job performance by up to 13%, without degrading a single job's performance over 1% in comparison with fair sharing.

II. Demand Elasticity

In this section, we demonstrate the demand elasticity of data-parallel jobs and motivate the need for performance-aware fair scheduling. Through EC2 experiments, we reveal the general performance trend for data-parallel jobs, notably machine learning and graph analytics algorithms in Spark MLlib [8]: with more allocations, the performance improvement tends to become marginal. We performed deep-dive analysis to explain the reasons behind. We show through a simple experiment that we can allocate a job much less resources than its fair share without a noticeable performance penalty. The saved resources can be used to speed up other jobs.

A. A Measurement Study of Demand Elasticity

We study how the performance of data analytics jobs varies with different amounts of resources allocated. The resource can take various forms, e.g., CPU cores, memory, disk or network I/O bandwidth. In our experiments, we measure the resource allocation by the number of compute *slots*, each containing a fixed amount of CPU cores and memory.

Metric. Given an allocation, our primary metric for job performance is the *progress rate*, defined as the shortest job completion time (JCT) normalized by the JCT with the allocated number of slots, i.e.,

$$\text{Progress Rate} = \frac{\text{Shortest JCT}}{\text{JCT with the allocated slots}}.$$

Here, the shortest JCT is achieved when the job is running alone in the cluster at full degree of parallelism. Progress rate is a normalized value between 0 and 1. Intuitively, the higher the value, the faster the job completes with the allocated number of slots. We adopt progress rate as our primary metric instead of JCT because the former can better illustrate how sensitive the performance is when the job is given more (fewer) slots.

Methodology. We ran a number of data analytics applications provided in SparkBench [15], a recent benchmark suite for Spark [16]. These applications represent typical machine learning and graph processing algorithms in Spark MLlib [8], including KMeans, PageRank, SVDPlusPlus, SVM, DecisionTree, and PCA. We configured each application to have a degree of parallelism of 32, meaning that each analytics job consists of 32 tasks and can ideally run on 32 slots in parallel.

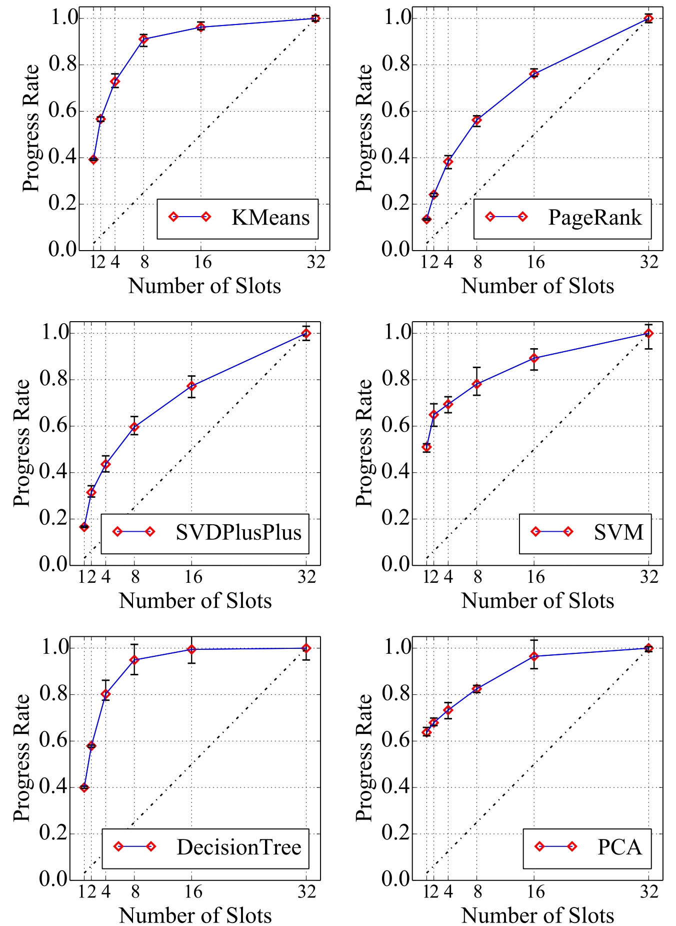
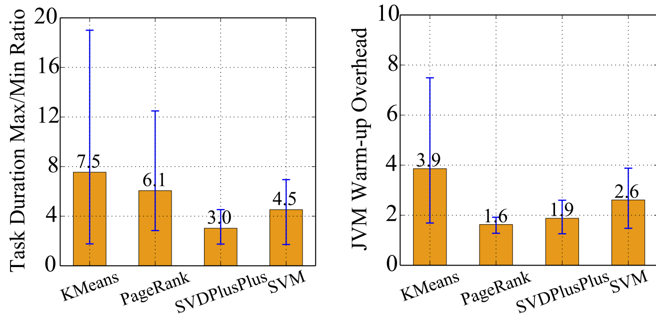


Fig. 1: [Cluster] Measured progress rate of data analytics applications in SparkBench [15] against the number of slots allocated.

We respectively ran each job in 1, 2, 4, 8, 16 and 32 slots in an Amazon EC2 [17] cluster, which contains 16 `m4.large` instances and is configured to offer totally 32 slots.

Measurement result. We depict the measured progress rate of each application against the number of allocated slots in Fig. 1, where each data point has been averaged over 5 runs, and the error bar captures the measured minimum and maximum progress. We make the following three observations.

1) *Elastic demand with marginal improvement:* Fig. 1 illustrates a clear trend that across all benchmark applications, the performance rate is by no means proportional to the allocation; instead, the performance-allocation curves show strong *concavity*. Meaning, having more resource allocations speeds up job completion, but such an improvement is marginally diminishing. For example, for KMeans job, compared with the single-thread mode, running it in two slots gains a dramatic progress improvement by $1.44\times$; in comparison, when the job is already allocated 8 slots, quadrupling its allocation to the full degree of parallelism only translates into a slight improvement less than 10%. In fact, running all workloads using half of the



(a) Ratio of the longest task duration to the shortest one in a stage

(b) Ratio of average task duration on cold JVM to that on warm JVM in a stage.

Fig. 2: [Cluster] Measurements of task duration unevenness and JVM warm-up overhead in our EC2 cluster. The results are based on five stages randomly selected from each job.

resources they ideally need is sufficient to achieve at least 75% of the maximum progress—a strong evidence that the resource demands are highly elastic.

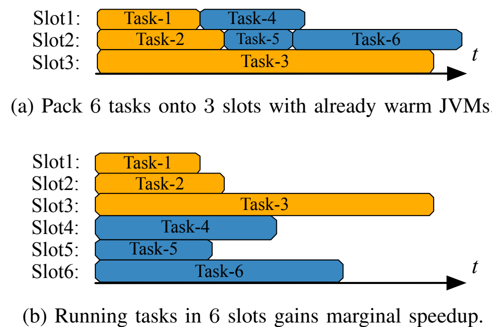
2) *Uneven demand elasticity:* While the performance-allocation curves are concave across all applications, the degree of concavity varies, so does the demand elasticity. Compared with the other jobs, the performance of SVDPlusPlus and PageRank scales more linearly with the number of slots, implying that allocating more resources to these two jobs likely gains higher benefits than others.
3) *Predictable performance model and demand elasticity:* Throughout our experiments, we observed small variance of performance across different runs (measured by the error bar in Fig. 1). This observation is in line with the previous results [11]–[14] that the performance model of data analytics jobs can be profiled and predicted accurately. In fact, we have confirmed that our measurement results in Fig. 1 can be accurately profiled using the recent prediction technique Ernest [13] (Sec. IV-A).

Curious about the three findings through our experiments, we ask: why do analytics jobs exhibit elastic demand with marginal improvement when allocated more slots? Is it an edge case limited to a particular workload or a general trend? We next analyze the root causes of the demand elasticity and show that it is by no means an accident.

B. Root Cause of Demand Elasticity

Data analytics jobs typically run as DAGs (directed acyclic graph) of *stages* each containing as many parallel tasks as possible [16]. Intuitively, with more slots alloted, the job runs more tasks in parallel, and hence completes faster. To explain why such a speedup is marginally decreasing, we analyze the task execution logs in our EC2 experiments and summarize our findings as follows.

Task packing. We attribute task packing as the main cause of running jobs in a relatively small number of slots without noticeable slowdown. A job's parallel tasks, even in the same stage, have significantly *uneven* execution time. In Fig. 2a, we depict the ratio between the runtime of the slowest and the fastest tasks in one stage of a job in our EC2 experiments. Each data point has been averaged over five randomly sampled stages, and the error bar measures the max and min ratio. We see that on average, the slowest task runs up to 7.5× longer than the fastest one in the same stage. Occasionally, the runtime disparity can even jump to 20×. We attribute the significantly uneven runtime to data skew, bottlenecked network transportation, and unstable machine state [18]–[20]. Those factors are generally stable for jobs repeated running over similar datasets in the same cluster, so the extent of unevenness of a recurring job's task runtime is similar with its past runs.



(a) Pack 6 tasks onto 3 slots with already warm JVMs.

(b) Running tasks in 6 slots gains marginal speedup.

Fig. 3: An example showing that running a job in 3 slots is slightly slower than running in 6 slots, as tasks 4, 5, and 6 can be tightly packed in 3 slots with already warm JVMs.

In the presence of uneven task runtime, there are many opportunities to "pack" tasks onto a small number of slots without stretching the JCT by a noticeable amount. Consequently, having more slots only translates into a marginal benefit. We illustrate this point in Fig. 3, where a job consists of six tasks and runs in three and six slots, respectively. By packing all tasks in three slots (cf. Fig. 3a), the JCT is only slightly increased as compared with running the job at full degree of parallelism with six slots (cf. Fig. 3b).

Careful readers may have noticed that the three blue tasks (i.e., tasks 4, 5, and 6) scheduled onto slots 1 and 2 in Fig. 3a are depicted to run *faster* than they do in Fig. 3b. We purposely make this illustration to highlight the impact of *JVM warm-up overhead*, which we explain next.

Reusing warm JVM. JVM warm-up overhead plays another key role in stalling the job that scales out to more slots. Many popular data analytic frameworks, such as Hadoop [21], Spark [22], and Tez [23], are built on the Java Virtual Machine (JVM). For jobs from such frameworks, the JVM warm-up overhead, i.e., class loading and interpretation of bytecode, is usually the bottleneck [10]. To quantify how significant the overhead could be, we measured the runtime of a task when running on a JVM with cold data normalized by the runtime on an already warm JVM with compiled code and loaded classes. Fig. 2b shows the overhead of four jobs, where each data point has been averaged over five randomly sampled stages, and the error bar measures the max and min overhead. On average, a task running on a cold JVM can be slowed down by up to 3.9×. Besides, as supported in [10], the extent of JVM warm-up

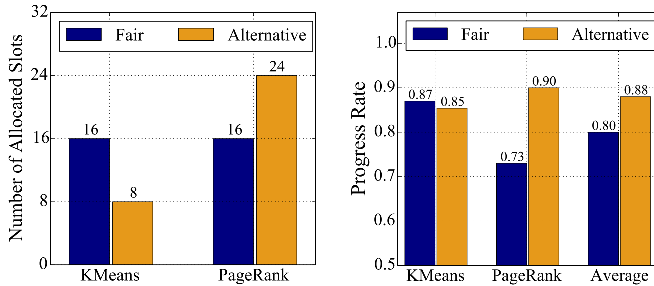


Fig. 4: [Cluster] When KMeans and PageRank are competing in a cluster with 32 slots, under Spark Fair scheduler the average progress rate is 0.80. In contrast, an alternative allocation scheme, by transferring 8 slots from KMeans to PageRank, can improve the average progress rate to 0.88.

overhead heavily depends on the job's code characteristic, thus it is stable for recurring jobs in different runs.

The significant JVM warm-up overhead can largely cancel out the benefits of job scaling out to more slots. As shown in Fig. 3b, with three additional slots, the job executes all six tasks in parallel. However, because tasks 4, 5, and 6 (dark blue) run in new slots with cold JVMs, they have to load classes and interpret bytecode from scratch, which results in longer completion time than reusing warm JVMs as in Fig. 3a.

In summary, aggressively running jobs in as many slots as possible may not justify the resource allocation. More often than not, we can expect fast job completion even with a small number of slots, as tasks can be tightly packed into a pool of well-warmed JVMs.

Other factors. Other factors can also affect the performance of jobs when executed with fewer slots than they ideally need, but their impact is limited. For example, by executing tasks in multiple batches, the TCP incast problem in the *shuffle* phase becomes less severe, allowing tasks to complete faster. Yet, it is argued in [24] that network is usually not a bottleneck, which is in line with our observation in the experiment. For the same reason, *data locality* is less of a concern, as remote data fetching does not noticeably stall the performance. In fact, the locality overhead only matters in the first *computation stage* and is amortized in later stages—by reusing the slots allocated in the first stage, downstream tasks can always have the best data locality.

C. The Need for Performance-Aware Fair Scheduling

Existing production schedulers [1]–[3], [7] seek to provide performance isolation by means of fair sharing, while keeping in the dark about the underlying job performance as well as the demand elasticity. However, blindly enforcing fair allocation is neither efficient nor necessary—due to demand elasticity, there are plenty of chances to run a job with fewer slots than its fair share, without a noticeable performance compromise.

To illustrate this point, we ran two SparkBench jobs, KMeans and PageRank, in our EC2 cluster with 16 `m4.large` instances (32 slots in total). The two jobs are configured to have the same weight. With Spark Fair scheduler, the allocation of each job is in proportion to its weight, and both jobs are evenly allocated 16 slots. We measured the progress rate of the two jobs and depict the results in Fig. 4. Now refer back to the performance-allocation curves of the two jobs in Fig. 1. For KMeans job, allocating it 16 slots is unnecessary, as it barely improves the job's performance over 8 slots; in comparison, PageRank job will get dramatic speedup if allocated more than 16 slots. This observation prompts us to transfer 8 slots from KMeans to PageRank. Fig. 4 shows the resultant allocation (left figure) and the corresponding progress rate measured for both jobs (right figure). Despite the loss of 8 slots, KMeans experiences only an slight progress drop by 2% (down from 0.87 to 0.85). In return, PageRank job gains significant benefits, improving its progress by 23% (up from 0.73 to 0.9).

We learn from the above experiment that, with the profiled knowledge on demand elasticity, a scheduler can achieve fast job completion without noticeably compromising performance isolation. This motivates us to propose performance-aware fair scheduling, which is the main theme of the next section.

III. Performance-Aware Fairness

In this section, we present Performance-Aware Fair scheduler (PAF) to take full use of the demand elasticity. Our objective is two-fold: optimizing the average job performance while providing near-optimal isolation guarantee. We start with a high-level overview of the algorithm, followed by a detailed description. We assume the availability of performance model and defer the discussion of how this information can be learned in Sec. IV-A.

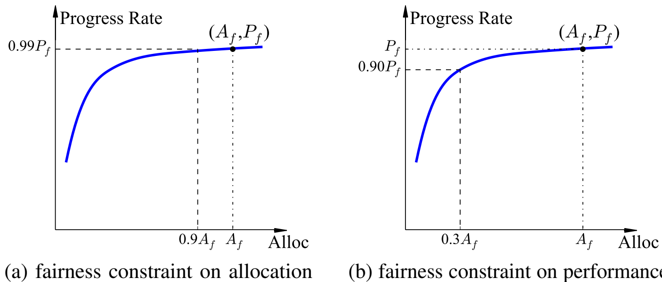
Overview. Intuitively, to pursue high performance, jobs that are *indifferent* to resource loss shall yield some of the fair share to others that can benefit more from those resources, as long as the performance of those altruistic jobs is not compromised beyond a small extent. To this end, PAF employs a two-phase procedure to work out the optimal allocation scheme.

The first phase is to impose a fairness constraint. Based on job performance models, PAF calculates the maximum resources each job can yield without sacrificing its performance over a threshold. In the second phase, to achieve the *best* overall performance, PAF adopts an heuristic algorithm that iteratively transfers resources among jobs so as to make the resources utilized most efficiently.

A. Imposing Fairness Constraint

Although performance improvement is desirable, it shall not come at the cost of severe fairness compromise. Therefore, when a job yields resources to others, we shall impose a fairness constraint to bound the maximal performance degradation that job would perceive.

Similar to [5], [9], [25], our first attempt is to impose the fairness constraints directly on *allocation* without considering the job performance models. That is, given a fairness knob α between 0 and 1, it is ensured that each job's allocation is at least an α-fraction of the fair share. However, users care more about the job performance than the actually allocated resources. In fact, due to demand elasticity, an α-fraction of fair allocation hardly translates into an α-fraction of the performance (progress



(a) fairness constraint on allocation (b) fairness constraint on performance

Fig. 5: Due to concavity of jobs' performance-allocation curves, imposing the fair constraint directly on performance instead of allocation allows more slots to be yielded. Given the fairness constraint α=0.9, the former allows the job to yield 0.7-fraction of its slots to benefit others, yet the latter only allows 0.1.

rate) under the fair allocation. For example, consider the performance-allocation curve in Fig. 5, where A_f is the fair allocation, and P_f is the corresponding progress rate. Suppose that $0.9P_f$ is also acceptable. Proportionally reducing the allocation to $0.9A_f$ only results in slightly degraded performance $0.99P_f$. In fact, it is sufficient to achieve the relaxed performance objective using $0.3A_f$.

Motivated by the example above, we propose to impose the fairness constraint on performance rather than on allocation. That is, given the job performance-allocation curves and the fairness knob α, we *implicitly* determine each job's minimum allocation by ensuring that *their progress rate is at least an α-fraction of that under fair sharing*. Thanks to demand elasticity, this approach allows a job to yield a larger number of slots to others without violating the fairness constraint.

In practice, to pursue near-optimal fairness, we require α to be close to 1. Yet, even under such a strict fairness constraint, we shall show in later sections that PAF can still achieve significant performance improvement.

B. Optimizing Overall Performance

Given the fairness constraint, the objective of PAF is to optimize the allocation to achieve the best possible performance. Formally, given a set of pending jobs J=$\{j_1, j_2, ..., j_n\}$, let f_i represent the allocation of j_i under a fair scheduler, and $p_i(x_i)$ the progress rate of j_i with allocation x_i. We formulate an optimization problem as follows:

$$\begin{aligned} \underset{\vec{x}=(x_1,x_2,...,x_n)}{\text{maximize}} \quad & \sum_i p_i(x_i), \\ \text{subject to} \quad & p_i(x_i) \geq \alpha p_i(f_i),\ i = 1, \ldots, n. \end{aligned} \tag{1}$$

By solving problem (1), PAF can find the optimal allocation scheme $(x_1^*, x_2^*, ..., x_n^*)$ that achieves the highest average progress rate. Here, we refer to x_i^* as the *PAF-allocation* of job j_i. While problem (1) can be directly solved using existing convex optimization toolboxes, doing so is too heavy-lifting. We instead resort to a simple, yet effective heuristic algorithm.

Following the basic intuition of PAF, the algorithm iteratively transfers resources from one job to another that can benefit more significantly by obtaining those resources. It starts by initializing the *PAF-allocation* of each job to its fair share, and then works by judiciously adjusting the *PAF-allocation* based on the job's performance-allocation curves. We shall show later that this algorithm converges to the optimal solution.

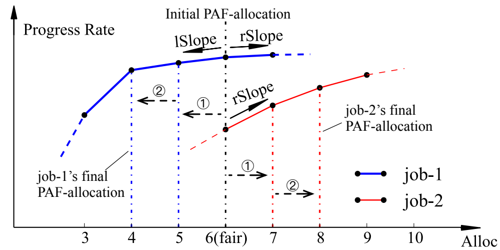


Fig. 6: PAF's working process to get the optimal allocation (i.e., *PAF-allocation*) of two jobs with the same weight. Initially each job's *PAF-allocation* is set to the fair allocation 6. Since job-1's *sacrificeRate* (i.e., the *lSlope* of the point with the *PAF-allocation*) is less than job-2's *benefitRate* (i.e., the *rSlope* of the point with the *PAF-allocation*), PAF iteratively transfers one slot from job-1's *PAF-allocation* to that of job-2. After two iterations the process terminates, and the final *PAF-allocations* of job-1 and job-2 are 4 and 8, respectively.

To illustrate how PAF works, we start with a simple case of two jobs, and then generalize the discussion to multiple jobs.

A simple case of two jobs. Given that job progress can only be measured with an *integral* number of slots, the profiled performance-allocation curve is *piecewise linear*. As shown in Fig. 6, we first define, for each measured point in that curve, the *lSlope* (slope of the adjacent piece to the left) and *rSlope* (slope of the adjacent piece to the right). In particular, we use *sacrificeRate* to measure *lSlope* of the point with *PAF-allocation*, as it quantifies the loss of progress rate if the job's *PAF-allocation* is reduced by one slot. Similarly, we measure *rSlope* by the *benefitRate*, which quantifies the progress gain if the job is allocated one more slot.

When there are only two jobs, as long as one job's *sacrificeRate* is less than the other's *benefitRate*, PAF iteratively transfers one slot from the former's *PAF-allocation* to the latter's. For the two jobs in Fig. 6, their initial *PAF-allocations* are both 6 slots. Noticing that job-1's *sacrificeRate* is less than job-2's *benefitRate*, PAF transfers one slot from job-1 to job-2. This process repeats in two iterations, until job-1's *sacrificeRate* exceeds job-2's *benefitRate*. We can easily verify that the resultant *PAF-allocations* of the two jobs, 4 and 8 slots, are optimal with the maximum average job performance.

Note that, the fairness constraint can be naturally integrated into the above process. Once a job's *PAF-allocation* has decreased to the lower bound required to satisfy the fairness constraint, the job would stop yielding more resources, and the slot transferring process terminates immediately.

PAF in general cases. For general cases with multiple jobs, PAF aims to converge to an equilibrium defined as follows.

Definition 1 (Global Equilibrium): Given a set of jobs J, let $Y \subseteq J$ be the set of jobs that can yield more slots without violating the fairness constraint. The *global equilibrium* is

Algorithm 1 Performance-Aware Fair Allocation

Input: Σ ▷ the set of pending jobs
Require: $\{p_j(x) \mid j \in \Sigma\}$ ▷ performance(p)-allocation(x) curves
Output: $\{j$.pafAlloc $\mid j \in \Sigma\}$ ▷ *PAF-allocations* of all jobs

```
1: for j in Σ do
2:     j.fairAlloc ← the fair share of j
3:     j.pafAlloc ← j.fairAlloc
4:     j.minAlloc ← min_x{x | p_j(x) ≥ α p_j(j.fairAlloc)}
           ▷ the minimum allocation to satisfy fairness constraint α
5: while True do
6:     resource-giver ← arg min_j{j.sacrificeRate | j ∈ Σ}
7:     resource-taker ← arg max_j{j.benefitRate | j ∈ Σ}
8:     if resource-giver.sacrificeRate ≥ resource-taker.benefitRate
9:         return
10:    else if resource-giver.pafAlloc = resource-giver.minAlloc
11:        Σ ← Σ \ {resource-giver}
12:    else
13:        resource-giver.pafAlloc ← resource-giver.pafAlloc−1
14:        resource-taker.pafAlloc ← resource-taker.pafAlloc+1
```

achieved when

$$\min_{j \in Y} j.\text{sacrificeRate} \geq \max_{j \in J} j.\text{benefitRate}.$$

Once the global equilibrium is achieved, the overall progress rate cannot be further improved without violating the fairness constraint. Therefore, the allocation scheme under the global equilibrium is the optimal solution.

We propose Algorithm 1 to find the global equilibrium. The basic idea is to iteratively reduce the gap between $\min_{j \in Y} \{j.\text{sacrificeRate}\}$ and $\max_{j \in J} \{j.\text{benefitRate}\}$. In each iteration, PAF first identifies two jobs, one with the minimum *sacrificeRate* (referred to as the *resource-giver*) and the other with the maximum *benefitRate* (referred to as the *resource-taker*). It then transfers one slot from the *resource-giver*'s *PAF-allocation* to that of the *resource-taker*. The iteration continues until there is no qualified resource givers (i.e., set Y becomes $\emptyset$) or the slot transferring does more harm than good to the overall performance (i.e., $\min_{j \in Y} \{j.\text{sacrificeRate}\} \geq \max_{j \in J} \{j.\text{benefitRate}\}$).

Algorithm 1 is more than a simple heuristic. We show that, given the fairness constraint, the algorithm can always converge to the global equilibrium with the optimal allocation.

Theorem 1 (Convergence): Algorithm 1 converges to the global equilibrium after a finite number of iterations.

Proof: We only need to prove that the algorithm always converges, because if the algorithm converges, it can only converge to the global equilibrium: otherwise, one slot can still be transferred from the *resource-giver* to the *resource-taker*.

We notice that the total slots that could be transferred among jobs is a finite number determined by the fairness constraint. Thus, to verify that the algorithm converges after a finite number of iterations, we only need to show that the transfer is *unidirectional*, i.e., each job *monotonously* yields or gains slots. Next, we will show by contradiction that a job who has yielded a slot is impossible to gain any slot later, and the other side could be proved symmetrically.

Suppose that a job j_1 yields a slot to others in iteration p, and gains a slot later from j_2 in iteration $q(p < q)$. Let j_i^p.sacrificeRate(benefitRate) represent the sacrificeRate(benefitRate) of j_i in iteration p. Then, because j_1 instead of j_2 is chosen to yield slot in iteration p, we have:

$$j_1^p.\text{sacrificeRate} \leq j_2^p.\text{sacrificeRate}. \quad (2)$$

In iteration q, since j_2 yields one slot to j_1, we have:

$$j_1^q.\text{benefitRate} > j_2^q.\text{sacrificeRate}. \quad (3)$$

Note that j_1^q.benefitRate is identical with j_1^p.sacrificeRate, we combine (2) with (3), and get:

$$j_2^q.\text{sacrificeRate} < j_2^p.\text{sacrificeRate}. \quad (4)$$

Due to the curve concavity of j_2, (4) implies that j_2 must have gain slots between iteration p and q. Suppose j_2 gains the last slot in iteration $s(p < s < q)$, then,

$$j_2^s.\text{benefitRate}{=}j_2^q.\text{sacrificeRate} < j_1^q.\text{benefitRate}{=}j_1^s.\text{benefitRate}.$$

This implies, it's j_1 instead of j_2 that shall gain the slot in iteration s. Hence there is a contradiction. ■

We next analyze the time complexity of Algorithm 1. Assume that there are N pending jobs in total. The *for* loop (line 1 to line 4 in Algorithm 1) takes $O(N)$ time to complete. Let M be the total number of slots in the cluster. The number of iterations in the *while* loop (line 5 to line 14 in Algorithm 1) must be smaller than M, because the slot transferring is unidirectional (shown in Theorem 1). Furthermore, each iteration in that *while* loop takes $O(\log N)$ time, due to the operations to search for jobs with the minimum sacrificeRate and then the maximum benefitRate. Putting all those altogether, the total time complexity of Algorithm 1 is $O(M \log N + N)$.

IV. Prototype Implementation

A. Curve Preparation

So far, our solutions are built upon an assumption that we can obtain the *perfect* job performance models. That is, given *each* possible number of allocated slots, we know the corresponding job progress rate *accurately*. However, this is usually not the case in practice. Even for recurring jobs whose performance models can be profiled from past runs, it would be too expensive for the profiling work to cover the entire allocation space. Besides, the profiling errors are usually unavoidable, due to factors like random variations.

Therefore, to make PAF practical, we need to address the challenge of *incomplete* or *inaccurate* performance profiling. This can be solved by the recent system called Ernest [13] which proposes a general mathematical form of job performance models for a wide range of data-analytics jobs. In particular, Ernest characterizes the relationship between progress rate p and the number of allocated slots x through the following equation:

$$p(x) = 1/(a_0 + a_1 * x + a_2/x + a_3 * \log x).$$

Here, coefficients a_0, a_1, a_2 and a_3 are decided by parametric regression. Based on this form, we can obtain the desired

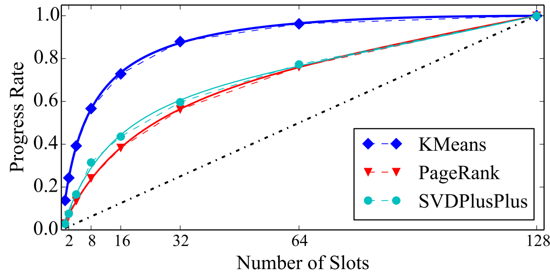


Fig. 7: [Cluster] Measured progress rate of KMeans, PageRank and SVDPlusPlus in SparkBench [15] against the number of slots allocated. The curves are fitted with Ernest model.

performance-allocation curves by parameter fitting with the available profiling data. As we shall show later in Sec. V, with only a few measured data points, the curves can be *completely* and *accurately* fit.

Our solution can also be easily made compatible with the jobs whose performance-allocation curves are not available: for those jobs, we simply allocate them the fair share and exclude them from the slot transferring process.

B. Spark Implementation

We have implemented PAF as a pluggable scheduler in Spark 2.1.0. Our implementation associates each job with its performance-allocation curve (i.e., the coefficients a_0, a_1, a_2 and a_3) as a job property, which is passed to the scheduler upon the arrival of the job. We implemented our PAF scheduler in the scheduling module of Spark called `TaskSchedulerImpl`. Once a new scheduling request is received, the scheduler computes the PAF-allocation of each pending job. It then enforces PAF-allocation by preferentially offering available slots to the jobs whose current allocation is the furthest from the target PAF-allocation.

V. Evaluation

In this section, we evaluate the effectiveness of PAF in terms of both average job performance and isolation guarantee. Through the EC2 experiments, we illustrate in a fine-grained manner how PAF behaves with different levels of fairness guarantees. Furthermore, for performance evaluation at a larger scale, we resort to simulations based on synthesized workloads.

A. Testbed Experiment

Setup. We deployed our Spark prototype in a 64-node Amazon EC2 cluster with `m4.large` instances (each with 2 cores and 8GB RAM). In total the cluster is configured to have 128 slots. As for the workloads, we chose three jobs from the SparkBench workload suite [15]: KMeans, PageRank and SVDPlusPlus. The degree of parallelism is set to 128 for each job. The performance-allocation curves of the three jobs (depicted in Fig. 7) are smoothed with the curve-fitting technique elaborated in Sec. IV. In particular, the three jobs are assigned different *weights*: 12 for KMeans, 3 for PageRank, and 1 for SVDPlusPlus. The amount of fair allocation a job receives is proportionally determined by its weight.

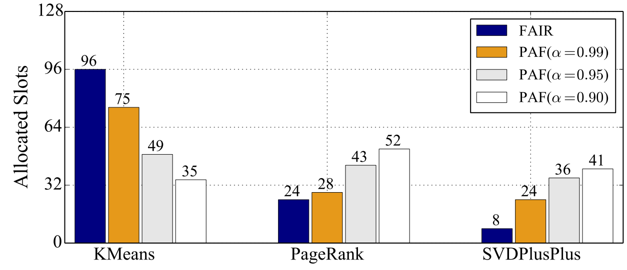


Fig. 8: [Cluster] Slot allocation results under different scheduling schemes.

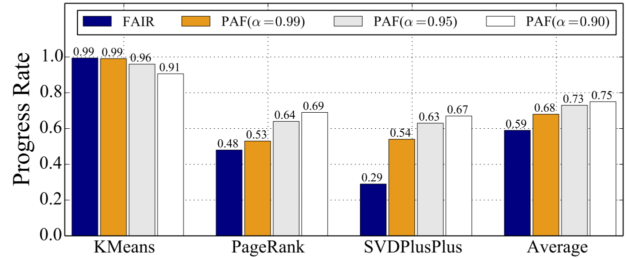


Fig. 9: [Cluster] Progress rates under different scheduling schemes. PAF can increase the average progress rate by around 27% (from 0.59 to 0.75).

We use Spark Fair Scheduler as the baseline. As illustrated in Fig. 8, the three jobs are respectively allocated 96, 24, and 8 slots under the baseline. In pursuit of better job performance, given the demand elasticity exposed in Fig. 7, it deserves to transfer some slots from KMeans to PageRank and SVDPlusPlus—doing so impairs KMeans only marginally, but it can largely benefit the other two jobs. However, Fair Scheduler is agnostic to this demand elasticity, and it simply settles on fair allocations that are proportional to the jobs' weights, which opportunistically hurts the overall performance.

In contrast, by being aware of the job performance, PAF is capable of achieving the *optimal* overall performance under any given fairness constraint. When that constraint α is set to 0.90 (a guarantee that any job's performance cannot be compromised by more than 10% compared with that under fair allocation), PAF transfers 61 slots from KMeans to improve PageRank and SVDPlusPlus. Consequently, as shown in Fig. 9, the average progress rate increases to 0.75—an improvement of 27% over that of fair allocation.

In general, with larger α (i.e., more restrictive fairness constraints), fewer slots are yielded by resource-givers, and the potential room for performance improvement becomes smaller. Nevertheless, even when α is set to 0.99, PAF can still improve the average progress rate by 15% (from 0.59 to 0.68). In other words, slightly relaxing the fairness requirement can result in a dramatic performance improvement.

Note that the performance improvement brought by PAF depends on the workloads. A natural question is: how would PAF behave for production applications? We next answer this question through large-scale simulations based on synthesized workloads.

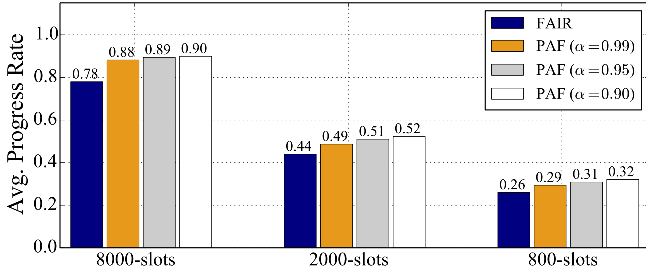
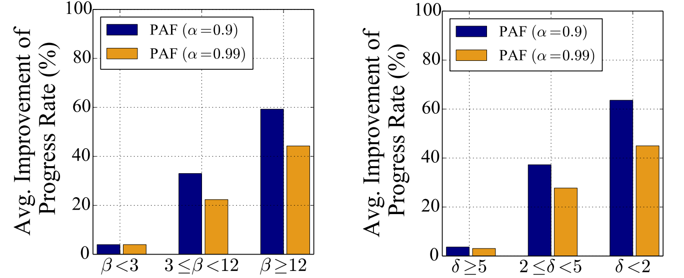


Fig. 10: [Simulation] PAF performance in different clusters.



(a) Shape parameter β in Pareto distribution of task runtime. Smaller β implies stronger concavity of job performance curve.

(b) Task acceleration factor δ due to JVM warm-up benefit. Larger δ implies stronger concavity of job performance-allocation curve.

Fig. 11: [Simulation] Average performance improvement of jobs under PAF (compared with fair scheduler), binned by factors affecting concavity of performance-allocation curves. In general, jobs whose curve exhibit stronger concavity are less benefited from PAF.

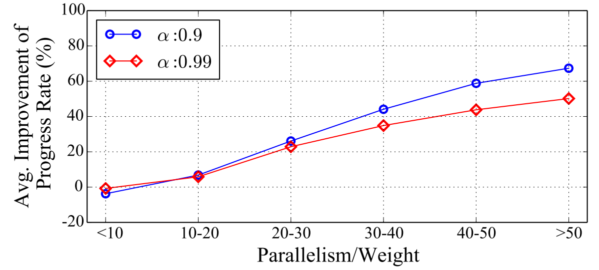


Fig. 12: [Simulation] Average performance improvement of jobs under PAF (compared with fair scheduler), binned by their values of parallelism-weight ratio. In general, jobs with larger parallelism yet smaller weight can benefit more from PAF.

B. Large-Scale Simulation

Setup. We have simulated three clusters of different sizes, consisting of 8000, 2000 and 800 slots, respectively. To emulate the real-world environment, our simulator has incorporated the impact of JVM warm-up on task performance: if a task reuses a slot that has just been released by others from the same job, that task's execution would be sped up by a JVM warm-up benefit factor δ, which is specified in each job as an internal attribute.

Workloads. We synthesize a workload suite based on the Google traces [26]. The workload suite contains over 200 jobs, and to evaluate PAF's performance in general case, the attributes of those jobs are chosen randomly from parameter pools. For each job, its degree of parallelism, weight, and JVM warm-up benefit factor δ, are randomly chosen from the range of 1-200, 1-20 and 1-8, respectively. Here, the range of δ is configured based on our observation in Fig. 2b. Besides, following prior works like [9], we assume that the task duration of a job follows a long-tail Pareto distribution, where the *shape parameter* β is randomly chosen from the reported typical range of 1.6-16. Shape parameter β measures the degree of *unevenness* of the distribution of a job's *task duration*. The smaller the shape parameter β, the more unevenly distributed the task durations of a job, and meanwhile the more concave the job performance-allocation curve. Finally, the average task duration and job submission intervals are determined based on the Google traces [26].

Fig. 10 depicts the performance of PAF in our simulation. In the large cluster with 8000 slots, a fairness constraint of $\alpha = 0.90$ allows PAF to improve the average progress rate by 15% (from 0.78 to 0.90). Even with a more restrictive fairness constraint of $\alpha = 0.99$, we can still gain an improvement of 13%. Meanwhile, we observe less improvement in clusters with fewer slots. This is because small clusters are more likely to get overloaded, and thus fewer jobs can yield slots to benefit others without violating the fairness constraints.

What workloads can benefit more from PAF? Apart from the overall improvement, we are also curious about how different workloads are affected under PAF. To figure out the answer, we performed the following deep-dive analysis based on the simulation results in the large cluster with 8000 slots.

*1) **Curve concavity**:* By randomly choosing the shape parameter β and JVM warm-up benefit factor δ, we have generated workloads with various performance-allocation curves. As described in Sec. II, a smaller β (i.e., stronger heterogeneity in task duration) and a larger δ both attribute to stronger curve concavity. To see how the degree of concavity affects the job performance under PAF, we classify all jobs into three groups based on their values of β and δ. We then measure how much the average progress rate of jobs in each group is improved under PAF compared with that under fair scheduler. Fig. 11 shows the results, from which we see that jobs whose curves exhibit stronger concavity are less benefited. This is because those jobs have more salient demand elasticity and usually serve as resource-givers.

*2) **Parallelism-weight ratio**:* In addition to the curve concavity, a job's parallelism-weight ratio may also affect its performance improvement under PAF. As shown in Fig. 12, the general trend is that jobs with larger parallelism-weight ratios usually gain more improvements. In fact, because the fair allocation of a job is proportional to its weight, jobs with a larger degree of parallelism (i.e., slot demand) but smaller weights would be starved more severely. For such slot-hungry jobs, their performance can be efficiently improved by just a few more slots. In contrast, jobs with smaller parallelism-weight ratio are more indifferent to the loss of slots. They usually act as resource-givers and are less likely to be benefited.

VI. RELATED WORK

Cluster scheduling. Most cluster schedulers [1]–[4] set *fairness* as the primary scheduling objective, where the fair allocation of each job is computed based on max-min fairness [27] or its variants, such as DRF [7] for multi-resources and Choosy [28] for cases with placement constraints. Yet, fair schedulers often cause long job response time [5], [29].

To achieve fast job response, several *performance-centric* schedulers [30], [31] resort to *Shortest Remaining Processing Time First* (SRPT)—the optimal strategy in theory [32] for minimizing the average job response time. Yet, SRPT is prone to *starvation*, meaning that large jobs are always assigned lower priority and may never get executed.

To address the deficiencies of both fairness-centric and performance-centric schedulers, there are some attempts to strike a balance between fairness and performance. For example, Grandl et al. [5] incorporates fairness into SRPT through a flexible fairness knob. Later, in light of the limitations of *instantaneous* fairness, CARBYNE [6] and CFQ [33] propose to selectively prioritize small jobs, as long as doing so does not delay the completion of other jobs, thus they achieve better efficiency without compromising performance isolation. Yet, none of the above works have noticed the job demand elasticity.

Performance modeling and prediction. Knowing in advance how job performance varies with the allocated resources can help decide the best resource configuration in terms of performance and cost. A series of performance prediction techniques have been developed recently.

An early work ARIA [12] aims to predict the completion time of recurring MapReduce jobs based on fine-grained performance information profiled in the previous runs, such as task durations. Recently, Ernest [13] proposes a general model form that captures the typical computation and communication patterns to predict the job execution time under various resource configurations. The model is trained with job behaviors on small samples of input data. More recently, to search of light-weight predictions, CherryPick [14] uses Bayesian Optimization to determine the most appropriate model with a confidence interval, which requires much fewer sample runs.

VII. CONCLUSION

In this paper, we have identified, demonstrated and exploited the demand elasticity of data-analytics jobs. With demand elasticity, jobs can run with significantly less amount of resources than they ideally need, with only a moderate performance penalty. We have verified its prevalence and predictable nature through EC2 measurements, and have shown that demand elasticity can be used to help speed up average job completion. To this end, we have proposed Performance-Aware Fair (PAF) scheduler to automatically exploit demand elasticity for optimal overall performance, while retaining near-optimal isolation guarantee. PAF starts with the fair allocation and then judiciously adjusts it by iteratively transferring resources from one job to another, so as to improve that resource-taker's performance while ensuring negligible penalty on the resource-giver. We have implemented PAF in Spark and confirmed its effectiveness using EC2 experiments and large-scale simulations.

ACKNOWLEDGEMENT

The research was supported in part by RGC GRF grants under the contracts 16211715 and 16206417, as well as an RGC CRF grant under the contract C7036-15G.

REFERENCES

[1] V. K. Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *ACM SoCC*. 2013.
[2] Hadoop Fair Scheduler. https://hadoop.apache.org/docs/r2.7.3/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.
[3] Hadoop capacity Scheduler. https://hadoop.apache.org/docs/r2.7.3/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.
[4] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI*. 2011.
[5] R. Grandl et al. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM*. 2014.
[6] R. Grandl et al. Altruistic scheduling in multi-resource clusters. In *USENIX OSDI*. 2016.
[7] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI*. 2011.
[8] Apache Spark MLlib. https://spark.apache.org/mllib/.
[9] X. Ren et al. Hopper: Decentralized speculation-aware cluster scheduling at scale. *ACM SIGCOMM CCR*, 2015.
[10] D. Lion et al. Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In *USENIX OSDI*. 2016.
[11] A. D. Ferguson et al. Jockey: guaranteed job latency in data parallel clusters. In *ACM Eurosys*. 2012.
[12] A. Verma et al. ARIA: automatic resource inference and allocation for mapreduce environments. In *ACM ICAC*. 2011.
[13] S. Venkataraman et al. Ernest: Efficient performance prediction for large-scale advanced analytics. In *USENIX NSDI*. 2016.
[14] O. Alipourfard et al. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *USENIX NSDI*. 2017.
[15] M. Li et al. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *ACM CF*. 2015.
[16] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*. 2012.
[17] Amazon EC2. https://aws.amazon.com/ec2/.
[18] J. Dean et al. The tail at scale. *Communications of the ACM*, 2013.
[19] G. Ananthanarayanan et al. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*. 2010.
[20] Y. Kwon et al. Skewtune: mitigating skew in mapreduce applications. In *ACM SIGMOD*. 2012.
[21] Hadoop. https://hadoop.apache.org.
[22] Spark. https://spark.apache.org.
[23] Tez. https://tez.apache.org.
[24] K. Ousterhout et al. Making sense of performance in data analytics frameworks. In *USENIX NSDI*. 2015.
[25] W. Wang et al. Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling. In *IEEE INFOCOM*. 2017.
[26] C. Reiss et al. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC*. 2012.
[27] D. Nace et al. A tutorial on max-min fairness and its applications to routing, load-balancing and network design. In *IEEE RIVF*. 2006.
[28] A. Ghodsi et al. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *ACM Eurosys*. 2013.
[29] M. Zaharia et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *ACM EuroSys*. 2010.
[30] Y. Wang et al. Preemptive reducetask scheduling for fair and fast job completion. In *USENIX ICAC*. 2013.
[31] B. Moseley et al. On scheduling in map-reduce and flow-shops. In *ACM SPAA*. 2011.
[32] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Oper. Res.*, 16(3):687–690, 1968.
[33] C. Chen et al. Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees. In *IEEE INFOCOM*. 2017.