BPF: Mitigating the Burstiness-Fairness Tradeoff in Multi-Resource Clusters

Paper #117, 13 Pages

Abstract

Simultaneously supporting latency- and throughout-sensitive workloads in a shared environment is a classic networking challenge that is becoming increasingly more common in big data clusters. Despite many advances, existing cluster schedulers force the same performance goal – fairness in most cases – on all jobs. Latency-sensitive jobs suffer, while throughput-sensitive ones thrive. Using prioritization does the opposite: it opens up a path for latency-sensitive jobs to dominate. In this paper, we tackle the challenges in supporting both short-term performance and long-term fairness simultaneously with high resource utilization by proposing Bounded Priority Fairness (BPF). BPF provides short-term resource guarantees to latency-sensitive jobs and maintains the long-term fairness for throughput-sensitive jobs. The key idea is "bounded" priority for latency-sensitive jobs; meaning, if bursts are not too large to hurt the long-term fairness, they are given higher priority so jobs can be completed as quickly as possible. BPF is the first scheduler that can provide long-term fairness, burst guarantee, and Pareto efficiency in a strategyproof manner for multi-resource scheduling. Additional treatment is included for uncertainties on the burst sizes. Deployments and large-scale simulations show that BPF closely approximates the performance of Strict Priority as well as the fairness characteristics of DRF. In deployments, BPF speeds up latency-sensitive jobs by $5.38\times$ compared to DRF, while still maintaining long-term fairness. In the meantime, BPF improves the average completion times of throughput-sensitive jobs by up to $3.05\times$ compared to Strict Priority.

1. Introduction

Cloud computing infrastructures are increasingly being shared between diverse workloads with heterogeneous re-

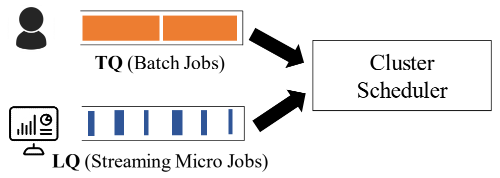
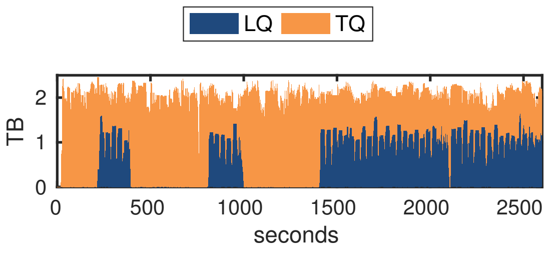


Figure 1: Users and automated processes with different throughput (TQ) and latency requirements (LQ) submit jobs to the same cluster.

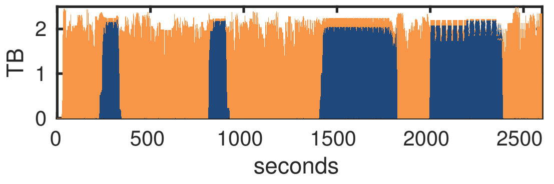
quests. In particular, throughput-sensitive batch processing systems [5, 29, 43, 53] are often complemented by latency-sensitive interactive analytics [4, 11, 64] and online stream processing systems [9, 12, 22, 56, 65]. Simultaneously supporting these workloads is a balancing act between distinct performance metrics ~~(Figure 1)~~. Figure 1 shows that cluster schedules the mix of jobs from throughput-sensitive queue (TQ) and latency-sensitive (LQ). In particular, batch processing workloads such as indexing [29] and log processing [1, 64] may submit hours-long large jobs. The *average* amount of resources received during this period is critical for its progress. On the other hand, interactive [4, 64] and online streaming [9, 65] workloads respectively submit on-demand and periodic smaller jobs; each job may take seconds or minutes to finish. Therefore, receiving enough resources immediately upon the arrival of a job is more important than the average resources received over longer time intervals.

To address the diverse goals, today's schedulers are becoming more and more complex. They are multi-resource [15, 26, 35, 37, 49], DAG-aware [25, 37, 64], and allow a variety of constraints [16, 36, 44, 63, 66]. Given all these inputs, they optimize for objectives such as fairness [19, 34, 35, 45], performance [33], efficiency [37], or different combinations of the three [38, 39]. However, all existing schedulers have one shortcoming in common: *they force the same performance goal on all jobs while jobs may have distinct goals*, and therefore fail to provide performance guarantee in the presence of multiple types of workloads with different performance metrics. In fact, the performance of existing schedulers can be *arbitrarily bad* for some workloads.

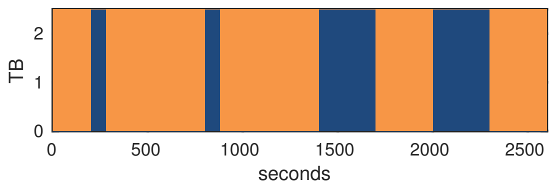
Consider the simple example in Figure 2 that illustrates the inefficiencies of existing resource allocation policies, in particular, DRF [35] and Strict Priority (SP) [51] for coex-



(a) DRF ensures instantaneous fairness, but increases the completion times of latency-sensitive jobs.

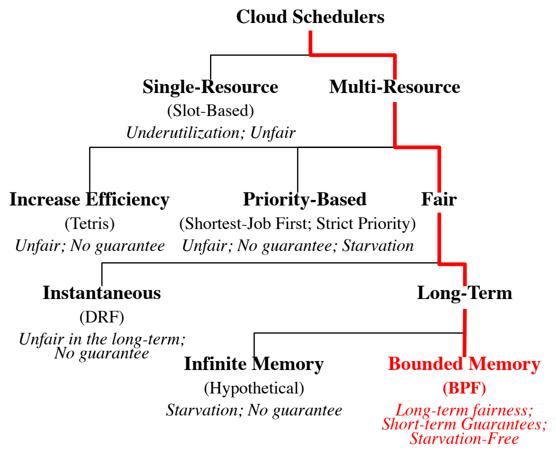


(b) SP decreases completion times of latency-sensitive jobs, but throughput-sensitive batch jobs do not receive their fair shares.



(c) The ideal solution allows first two latency-sensitive jobs to finish as quickly as possible, but protects batch jobs from latter LQ jobs by ensuring long-term fairness.

Figure 2: Need for bounded priority and long-term fairness in a shared multi-resource cluster with latency-sensitive (LQ: blue/dark) and throughput-sensitive (TQ: orange/light) jobs. The blank part on the top is due to resource fragmentation and overheads in Apache YARN. Although we focus only on memory allocations here, similar observations hold in multi-resource scenarios. **(TODO: put the memory (TB) instead of (TB) on the figures.)**


Cloud Schedulers
Single-Resource
(Slot-Based)
Underutilization; Unfair
Multi-Resource
Increase Efficiency
(Tetris)
Unfair; No guarantee
Priority-Based
(Shortest-Job First; Strict Priority)
Unfair; No guarantee; Starvation
Fair
Instantaneous
(DRF)
Unfair in the long-term; No guarantee
Long-Term
Infinite Memory
(Hypothetical)
Starvation; No guarantee
Bounded Memory
(BPF)
Long-term fairness; Short-term Guarantees; Starvation-Free
Figure 3: BPF in the cluster scheduling design space.

isting workloads with different requirements. In this example, we run memory-bound jobs in ~~Tez atop Apache YARN~~ within a cluster of 40 nodes, where each node has 32 CPU cores and 64 GB RAM. There are two queues in the system, where each queue contains a number of jobs with the same performance goals. Apache Hadoop YARN [2] is set up to manage the resource allocation among the queues. The first queue is for Spark streaming, which submits ~~a number of MapReduce jobs~~ a MapReduce job every 10 minutes. We call it latency-sensitive queue (LQ) because it aims to finish the jobs as quickly as possible. The second queue is a throughout-sensitive batch-job queue (TQ) formed by jobs generated from the BigBench workload [3] and queued up at the beginning. TQ cares more about its long-term averaged resources received, e.g., every 10 minutes. **(TODO: remove the TQ and LQ definition as they are defined in the Figure 1.)** For simplicity of exposition, all jobs are memory-bound. We consider two extreme classes of policies – priority-based and fairness-based allocation – in this example, where the former is optimized for latency and the latter for fairness. Specifically, we focus on the inefficiency of DRF and SP. The memory resource consumption under these two policies is depicted in Figures 2a and 2b, respectively. We defer the discussion of other policies to Section 2.3.

SP gives LQ the whole cluster's resources (high priority) whenever it has jobs to run; hence, it provides the lowest possible response time. For the first two arrivals, the average response time is 130 seconds. A detrimental side effect of SP, however, is that there is no resource isolation – TQ jobs may not receive any resources at all! In particular, LQ has incentives to increase its arrivals – e.g., for more accurate sampling and more iterations in training neural networks – without any punishment. As it does so from the third job arrival, TQ no longer receives its fair share. In the worst case, LQ can take all the system resources and *starve* TQ. In summary, SP provides the best response time for LQ, but no isolation protection for TQ at all. In addition, SP is incapable of handling multiple LQs.

In contrast, DRF enforces *instantaneous* fair allocation of resources at all times. During the burst of LQ, LQ and TQ share the bottleneck resource (memory) evenly until the jobs from LQ complete; then TQ gets all resources before the next burst of LQ. Clearly, TQ is happy at the cost of longer completion times of LQ's jobs, whose response time increases by 1.6 times. In short, DRF provides the best isolation protection for TQ, but no performance consideration for LQ. When there are many TQs, the response time of LQ can be very large.

Clearly, it is impossible to achieve the best response time under *instantaneous* fairness, which fully decides the allocation. In other words, there is a hard tradeoff between providing instantaneous fairness for TQs and minimizing the response time of LQs. Consequently, we aim to answer the following fundamental question in this paper: *how well can we simultaneously accommodate multiple classes of workloads with performance guarantees, in particular, isolation protection for TQs and low response times for LQs*?

We answer this question by designing BPF: the first multi-resource scheduler that achieves both isolation protection for TQs in terms of *long-term* fairness and response time guarantees for LQs, and is strategyproof. It is simple to implement and provides significant performance improvements even in the presence of uncertainties. The key idea is "bounded" priority for LQs: as long as the burst is not too large to hurt the long-term fair share of TQs, they are given higher priority so jobs can be completed as quickly as possible. Figure 3 shows BPF in the context of cluster scheduling landscape. We make the following contributions.

Algorithm design. We develop BPF with the rigorously proven properties of strategyproofness, short-term bursts, long-term fairness, and high system utilization (§3). When LQs have different demands for each arrival, we further design mechanism to handle the uncertainties.

Design and implementation. We have implemented BPF on Apache YARN [61] (§4). Any framework that runs on YARN can take advantage of BPF. The BPF scheduler is implemented as a new scheduler in Resource Manager that runs on the master node. The scheduling overheads for admitting queues or allocating resources are negligibly less than 1 ms for 20,000 queues.

Evaluation based on both testbed experiments and large-scale simulations. In deployments, BPF significantly provides up to $5.38\times$ lower completion times for LQ jobs than DRF, while maintaining the same long-term fairness (§5.2). At the same time, BPF provides up to $3.05\times$ more fair allocation to TQ jobs compared to SP.

2. Motivation

2.1 Benefits of Temporal Co-scheduling

Consider the example in Figure 2 again. Recall that SP and DRF are two extreme cases in trading off performance and fairness: SP provides the best performance (for LQs) with no fairness consideration (for TQs); DRF ensures the best isolation (for TQs) with poor performance (of LQs). However, it is still possible for LQs and TQs to share the cluster by thoughtful co-scheduling over time.

The ideal allocation is depicted in Figure 2c. The key idea is "bounded" priority for LQs as we discuss in the previous section. In particular, before 1,400 seconds, LQ's bursts are small, so it gets higher priority, which is similar to SP. After LQ increases its demand, only a fraction of its demand can be satisfied with the entire system's resources. Then it has to give resources back to TQ to ensure the long-term fairness.

2.2 Desired Properties

We restrict our attention in this paper to the following, important properties: burst guarantee for LQs, long-term fairness for TQs, strategyproofness, and Pareto efficiency to improve cluster utilization.

Property	SP	DRF	M-BVT	BPF
Burst Guarantee (BG)	✓*	×	✓*	✓
Long-Term Fairness (LF)	×	✓	✓	✓
Strategyproofness (SPF)	×	✓	×	✓
Pareto Efficiency (PE)	✓	✓	✓	✓
Single Resource Fairness	×	✓	✓	✓
Bottleneck Fairness	×	✓	✓	✓
Population Monotonicity	✓	✓	✓	✓

Table 1: Properties of existing policies and BPF. ✓* means that the property holds when there is only one LQ.

Burst guarantee (BG) provides performance guarantee for LQs by allocating guaranteed amount of resources during their bursts. In particular, an LQ requests its minimum required resources for its bursts to satisfy its service level agreements, e.g., percentiles of response time.

Long-term fairness (LF) provides every queue in the system the same amount of resources over a (long) period, e.g., 10 minutes. Overall, it ensures that TQs progress no slower than any LQ in the long run. LF implies sharing incentive, which requires that each queue should be better off sharing the cluster, than exclusively using its own static share of the cluster. If there are n queues, each queue cannot exceed $\frac{1}{n}$ of all resources under a static sharing.[1]

Strategyproofness (SPF) ensures that queues cannot benefit by lying about their resource demands. This provides incentive compatibility, as a queue cannot improve its allocation by lying.

Pareto efficiency (PE) is about the optimal utilization of the system. A resource allocation is Pareto efficient if it is impossible to increase the allocation/utility of a queue without hurting at least another queue.

2.3 Analysis of Existing Policies

Strict Priority (SP): SP is employed to provide performance guarantee for LQs. As the name suggests, an SP scheduler always prioritize LQs. Therefore, when there is only one LQ, SP provides the best possible performance guarantee. However, when there are more than one LQs, it is impossible to give all of them the highest priority. Meanwhile, TQs may not receive enough resources, which violates long-term fairness. As the LQs may request more resources than what they actually need, strategyproofness is not enforced, and therefore the system may waste some resources – i.e., it is not Pareto efficient.

DRF: DRF is an extension of max-min fairness to the multi-resource environment, where the dominant share is used to map the resource allocation (as a vector) to a scalar value. It provides instantaneous fairness, strategyproofness, and Pareto efficiency. However, because DRF is an instantaneous allocation policy without any memory, it cannot prioritize jobs with more urgent deadlines. In particular, no

[1] For simplicity of presentation, we consider queues with the same weights, which can be easily extended to queues with different weights.

burst guarantee is provided. Even assigning queues different weights in DRF is homogeneous over time and cannot provide the burst guarantee needed. In addition, there is no admission control. Therefore, as the number of queues increases, no queue's performance can be guaranteed.

M-BVT: BVT [30] was designed as a scheduler for a mix of real-time and best-effort tasks. The idea is that for real-time tasks, BVT allows them to borrow some virtual time (and therefore resources) from the future and be prioritized for a period without increasing their long-term shares.

To make it comparable, we extend it to M-BVT for a multi-resource environment. Under the M-BVT policy, LQ-i is assigned a virtual time warp parameter W_i, which represents the urgency of the queue. Upon an arrival of its burst at A_i, an effective virtual time $E_i = A_i - W_i$ is calculated. This is used as the priority (smaller E_i means higher priority) for scheduling. When LQ-i has the only smallest E_i, it may use the whole system's resources and its E_i increases at the rate of its progress calculated by DRF. Eventually, its E_i is no longer the only smallest. Then resources are shared in a DRF-fashion among queues with the smallest virtual times.

The M-BVT policy has some good properties. For instance, the DRF component ensures long-term fairness, and the BVT component strives for performance. Pareto efficiency follows from the work conservation of the policy.

However, it does not provide general burst guarantees as any new arriving queue (with larger virtual time warp parameter) may occupy the resources of existing LQs or share resources with them, thus hurting their completion time. In addition, it is not strategyproof because queues can lie about their needs in order to get a larger virtual time warp.

Other policies such as the CEEI [55] provide fewer desired properties.

2.4 Summary of the Tradeoffs

As listed in Table 1, no prior policy can simultaneously provide all the desired properties of fairness/isolation for TQs while providing burst guarantees for all the LQs with strategyproofness. In particular, if strict priority is provided to an LQ without any restriction for its best performance (e.g., SP), there is no isolation protection for TQs' performance. On the other hand, if the strictly instantaneous fairness is enforced (e.g., DRF), there is no room to prioritize short-term bursts. While the idea in M-BVT is reasonable, it is not strategyproof and cannot provide burst guarantee.

The key question of the paper is, therefore, how to allocate system resources in a near-optimal way; meaning, satisfying all the critical properties in Table 1.

3. BPF: A Scheduler With Memory

In this section, we first present the problem settings (§3.1) and then formally model the problem in Section 3.2. BPF achieves the desired properties by admission control, guaranteed resource provision, and spare resource allocation presented in Section 3.3. Finally, we prove that BPF satisfies all the properties in Table 1 (§3.4).

Notation	Description
$\mathbb{H}$	Admitted LQs with hard guarantee
$\mathbb{S}$	Admitted LQs with soft guarantee
$\mathbb{E}$	Admitted TQs and LQs with fair share only

Table 2: Important notations

3.1 Problem Settings

We consider a system with K types of resources. The capacity of resource k is denoted by C^k. The system resource capacity is therefore a vector $\overrightarrow{C} = \langle C^1, C^2, ..., C^K \rangle$. For ease of exposition, we assume $\overrightarrow{C}$ is a constant over time, while our methodology applies directly to the cases with time-varying $\overrightarrow{C}(t)$, e.g., with estimations of $\overrightarrow{C}(t)$ at the beginning and leveraging stochastic optimization [57] and online algorithm design [46].

We restrict our attention to LQs for interactive sessions and streaming applications, and TQs for batch jobs.

LQ-i's demand comes from a series N_i of bursts, each consisted of a number of jobs. We denote by $T_i(n)$ the arrival time of the n-th burst, which must be finished within $t_i(n)$. Therefore, its n-th burst needs to be completed by $T_i(n) + t_i(n)$ (i.e., deadline). Denote the demand of its n-th arrival by a vector $\overrightarrow{d_i}(n) = \langle d_i^1(n), d_i^2(n), ..., d_i^K(n) \rangle$, where $d_i^k(n)$ is the demand on resource-k.

In practice, inter-arrival time between consecutive bursts $T_i(n+1) - T_i(n)$ can be fixed for some applications such as Spark Streaming [65], or it may vary for interactive user sessions. In general, the duration is quite short, e.g., several minutes. Similarly, the demand vector $\overrightarrow{d_i}(n)$ may contain some uncertainties, and we assume that queues have their own estimations. Therefore, our approach has to be strategyproof so that queues report their estimated demand, as well as their true deadlines.

To enforce the long-term fairness, the total demand of LQ-i's n-th arrival $\overrightarrow{d_i}(n)$ should not exceed its fair share, which can be calculated by a simple fair scheduler – i.e., $\frac{\overrightarrow{C}(T_i(n+1)-T_i(n))}{N}$, when there are N queues admitted by BPF – or a more complicated one such as DRF. We adopt the former in analysis because it provides a more conservative evaluation of the improvements brought by BPF.

In contrast, TQs' jobs are queued at the beginning with much larger demand than each burst of LQs.

3.2 Modeling the Problem

Completion time: Let us denote by $R_i(n)$ the (last) completion time of jobs during LQ-i's n-th arrival. If LQ-i is admitted with *hard* guarantee, we ensure that a large fraction $(\alpha_i)^2$ of arrivals are completed before deadlines, i.e.,

^{2} α_i can be 95% or 99% depending on the SLAs

$\sum_{n \in N_i} \mathbf{1}_{\{R_i(n) \leq T_i(n)+t_i(n)\}} \geq \alpha_i |N_i|$, where $\mathbf{1}_{\{.\}}$ is the indicator function which equals to 1 if the condition is satisfied and 0 otherwise, $|N_i|$ is the number of arrivals of LQ-i. If LQ-i is admitted with only *soft/best-effort* guarantee, we maximize the fraction of arrivals completed on time.

Long-term fairness: Denote by $\overrightarrow{a_i}(t)$ and $\overrightarrow{e_j}(t)$ the resources allocated for LQ-i and TQ-j at time t, respectively. For a possibly long evaluation interval $[t, t+T]$ during which there is no new admission or exit, the average resource guarantees received are calculated as $\frac{1}{T}\int_t^{t+T} \overrightarrow{a_i}(\tau)\mathrm{d}\tau$ and $\frac{1}{T}\int_t^{t+T} \overrightarrow{e_j}(\tau)\mathrm{d}\tau$. We require the allocated dominant resource, i.e., the largest amount of resource allocated across all resource types, received by any TQ queue is no smaller than that received by an LQ. Formally, $\forall i \in \mathbb{A}, \forall j \in \mathbb{B}$, where $\mathbb{A}$ and $\mathbb{B}$ is the set of admitted LQs and TQs, respectively, $\max_k \left\{\frac{1}{T}\int_t^{t+T} a_i^k(\tau)\mathrm{d}\tau\right\} \leq \max_k \left\{\frac{1}{T}\int_t^{t+T} e_j^k(\tau)\mathrm{d}\tau\right\}$, where $a_i^k(\tau)$ and $e_j^k(\tau)$ are allocated type-k resources for LQ-i and TQ-j at time τ, respectively. This condition provides long-term protections for admitted TQs.

The optimization problem: We would like to maximize the arrivals completed before the deadlines for admitted LQs with soft guarantee while meeting the specified fraction of deadlines of admitted LQs with hard guarantees and keeping the long-term fairness.

The decisions to be made are (i) admission control, which decides the set of admitted LQs ($\mathbb{H}$, $\mathbb{S}$) and the set of admitted TQs ($\mathbb{E}$); and (ii) resources allocated to admitted queues LQ-i and TQ-j ($\overrightarrow{a_i}(t)$ and $\overrightarrow{e_j}(t)$, respectively) over time. If there are some unused/unallocated resources, queues with unsatisfied demand can share them.

3.3 Solution Approach

Our solution BPF consists of three major components: admission control procedure to decide $\mathbb{H}$, $\mathbb{S}$ and $\mathbb{E}$, guaranteed resource provisioning procedure for $\overrightarrow{a_i}(t)$, and spare resource allocation procedure.

Admission control procedure: BPF admits queues into the following three classes:

- $\mathbb{H}$: LQs admitted with hard resource guarantee.
- $\mathbb{S}$: LQs admitted with soft resource guarantee. Similar to hard guarantee, but need to wait when some LQs with hard guarantee are occupying system resources.
- $\mathbb{E}$: Elastic queues that can be either LQs or TQs. There is no burst guarantee, but long-term fair share is provided.

Before admitting LQ-i, BPF checks if admitting it invalidates any resource guarantees committed for LQs in $\mathbb{H} \cup \mathbb{S}$, i.e., the following *safety condition* needs to be satisfied:

$$\overrightarrow{d_j}(n) \leq \frac{\overrightarrow{C}\,(T_j(n+1) - T_j(n))}{|\mathbb{H}| + |\mathbb{S}| + |\mathbb{E}| + 1}, \forall n, \forall j \in \mathbb{H} \cup \mathbb{S}, \qquad (1)$$

where $|\mathbb{H}| + |\mathbb{S}| + |\mathbb{E}|$ is the number of already admitted queues. If (1) is not satisfied, LQ-i is rejected. Otherwise, it

Algorithm 1 BPF Scheduler

```
1: procedure PERIODICSCHEDULE()
2:     if there are new LQs $\mathbb{Q}$ then
3:         $\{\mathbb{H}, \mathbb{S}, \mathbb{E}\}$ =LQADMIT($\mathbb{Q}$)
4:     if there are new TQs $\mathbb{Q}$ then
5:         $\{\mathbb{E}\}$ =TQADMIT($\mathbb{Q}$)
6:     ALLOCATE($\mathbb{H}, \mathbb{S}, \mathbb{E}$)
7:
8: function LQADMIT(LQS $\mathbb{Q}$)
9:     for all LQ $Q \in \mathbb{Q}$ do
10:        if safety condition (1) satisfied then
11:            if fairness condition (2) satisfied then
12:                if resource condition (3) satisfied then
13:                    Admit $Q$ to hard guarantee $\mathbb{H}$
14:                else
15:                    Admit $Q$ to soft guarantee $\mathbb{S}$
16:            else
17:                Admit $Q$ to elastic $\mathbb{E}$ with long-term fair share
18:        else
19:            Reject $Q$
20:    return $\{\mathbb{H}, \mathbb{S}, \mathbb{E}\}$
21:
22: function TQADMIT(QUEUE $\mathbb{Q}$)
23:    for all TQ $Q \in \mathbb{Q}$ do
24:        if safety condition (1) satisfied then
25:            Admit $Q$ to elastic $\mathbb{E}$ with long-term fair share
26:        else
27:            Reject $Q$
28:    return $\{\mathbb{E}\}$
29:
30: function ALLOCATE($\mathbb{H}$, $\mathbb{S}$, $\mathbb{E}$)
31:    for all LQ $Q \in \mathbb{H}$ do
32:        $\overrightarrow{a_i}(t) = \frac{\overrightarrow{d_i}(n)}{t_i(n)}$ for $t \in [T_i(n), T_i(n+1)]$
33:    for all LQ $Q \in \mathbb{S}$ do
34:        allocate $\overrightarrow{C} - \sum_{j \in \mathbb{H}} \overrightarrow{a_j}(t)$ based on SRPT until each
   LQ-$i$'s allocation reaches $\overrightarrow{d_i}(n)$ or the deadline arrives.
35:    Obtain the remaining resources $\overrightarrow{L}$
36:    DRF($\mathbb{E}$, $\overrightarrow{L}$)
```

is safe to admit LQ-i and the next step is to decide which of the three classes it should be added to.

For LQ-i to have some resource guarantee, either hard or soft, its own total demand should not exceed its long-term fair share. Formally, the *fairness condition* is

$$\overrightarrow{d_i}(n) \leq \frac{\overrightarrow{C}\,(T_i(n+1) - T_i(n))}{|\mathbb{H}| + |\mathbb{S}| + |\mathbb{E}| + 1}, \forall n, \qquad (2)$$

If only condition (1) is satisfied but (2) is not, LQ-i is added to $\mathbb{E}$. If both conditions (1) and (2) are satisfied, it is safe to admit LQ-i to $\mathbb{H}$ or $\mathbb{S}$. If there are enough uncommitted resources (*resource condition* (3)), LQ-i is admitted to

$\mathbb{H}$. Otherwise it is added to $\mathbb{S}$.

$$\frac{\overrightarrow{d_i}(n)}{t_i(n)} \leq \overrightarrow{C} - \sum_{j \in \mathbb{H}} \overrightarrow{a_j}(t), \forall n, t \in [T_i(n), T_i(n) + t_i(n)]. \tag{3}$$

For TQ-j, BPF simply checks the safety condition (1). If it is satisfied, TQ-j is added to $\mathbb{E}$. Otherwise TQ-j is rejected.

Guaranteed resource provisioning procedure

For each LQ-i in $\mathbb{H}$, during $[T_i(n), T_i(n)+t_i(n)]$, BPF allocates constant resources to fulfill its demand $\overrightarrow{a_i}(t) = \frac{\overrightarrow{d_i}(n)}{t_i(n)}$. LQs in $\mathbb{S}$ shares the uncommitted resource $\overrightarrow{C} - \sum_{j \in \mathbb{H}} \overrightarrow{a_j}(t)$ based on SRPT [18] until each LQ-i's consumption reaches $\overrightarrow{d_i}(n)$ or the deadline arrives.

After every LQ in $\mathbb{H}$ and $\mathbb{S}$ is allocated, remaining resources are allocated to queues in $\mathbb{E}$ using DRF [35].

Spare resource allocation procedure

If some allocated resources are not used, they are further shared by TQs and LQs with unsatisfied demand. This maximizes system utilization.

3.4 Properties of BPF

First, we argue that *BPF ensures long-term fairness, burst guarantee, and Pareto efficiency*.

The safety condition and fairness condition ensure the long-term fairness for all TQs.

For LQs in $\mathbb{H}$, they have hard resource guarantee and therefore can meet their SLA. For LQs in $\mathbb{S}$, they have resource guarantee whenever possible, and only need to wait after LQs in $\mathbb{H}$ when there is a conflict. Therefore, their performance is much better than if they were under fair allocation policies.

The addition of $\mathbb{S}$ allows more LQs to be admitted with resource guarantee, and therefore increases the system resources utilized by LQs. Finally, we fulfill spare resources with TQs, so system utilization is maximized, reaching Pareto efficiency.

In addition, we prove that *BPF is strategyproof.*

Let the true demand and deadline of LQ-i be $(\overrightarrow{d}, t)$ for a particular arrival. Let the request parameter be $(\overrightarrow{v}, t')$. We first argue that $\overrightarrow{v} = \beta \overrightarrow{d}$ holds with $\beta > 0$.

As $\overrightarrow{d} = \langle d^1, d^2, \cdots, d^k \rangle$, $\overrightarrow{v} = \langle v^1, v^2, \cdots, v^k \rangle$, let $p = \min_k \left\{ \frac{v^i}{d^i} \right\}$. Define a new vector $\overrightarrow{z} = \langle z^1, z^2, \cdots, z^k \rangle$ and let $z^i = pd^i$. Notice here $\overrightarrow{z}$ has the same performance as $\overrightarrow{v}$, while $z_i \leq v_i$ for any i. Therefore, $\overrightarrow{z}$ may request no more resources than $\overrightarrow{v}$, which is more likely to be admitted. Hence, it is always better to request $\overrightarrow{z}$, which is proportional to $\overrightarrow{d}$, the true demand.

Regarding the demand $\overrightarrow{d}$, reporting a larger $\overrightarrow{v}$ ($\beta > 1$) still satisfies its demand, while has a higher risk being rejected as it requests higher demand, while it does not make sense to report a smaller $\overrightarrow{v}$ ($\beta < 1$) as it may receive fewer resources than it actually needs. Therefore, there is no incentive to for LQ-i to lie about its $\overrightarrow{d}$. The argument for deadline is similar. Reporting a larger deadline does not make sense as it may receive fewer resources than it actually needs. On the other side, reporting a tighter deadline still satisfies the deadlines, while has a higher risk being rejected as it requests higher demand. Therefore, there is no incentive to for LQ-i to lie about its deadline, either.

3.5 Handling uncertainties

In practice, arrivals of LQ-i may have different sizes, i.e., $\overrightarrow{d_i}$ is not deterministic but instead has some probability distributions. Now we extend BPF to handle this case.

We assume LQ-i knows its distributions, e.g., from historical data. In particular, it knows the cumulative probability distribution on each resource k, denoted by F_{ik} if these distributions on multiple resources are independent. The requirement regarding α_i can be converted into $\prod_j F_{ik}^{-1}(d_{ik}) \geq \alpha_i$, where d_{ik} is the request demand on resource k. This gives $F_{ik}^{-1}(d_{ik}) \geq \alpha_i^{1/K}$. Finally the request on resource-k $d_{ik} = F_{ik}(\alpha_i^{1/K})$. We call this α-strategy.

When distributions on multiple resources are correlated, we only have the general form $F_i^{-1}(\overrightarrow{d_i}) \geq \alpha_i$, where F_i is the joint distribution on all resources. We have the following properties in this case.[3] When the distributions are pairwise positively correlated, α-strategy over-provisions resources. If they are pairwise negatively correlated, α-strategy under-provisions resources. Numerical approaches can be applied to adjust the α-strategy accordingly.

Taking the correlations on multiple resources into consideration is important. In particular, when these distributions are perfectly correlated, $d_{ik} = F_{ik}(\alpha_i^{1/K})$ can be reduced to $d_{ik} = F_{ik}(\alpha_i)$. When the standard deviation is large (e.g., 40% of the mean for Normal distribution, this can reduce the demand by 10% with three resources). This increases the chance of LQ-i being admitted.

4. Design Details

In this section, we describe how we have implemented BPF on Apache YARN, how we use standard techniques for demand estimation, and additional details related to our implementation.

4.1 Enabling BPF in Cluster Managers

Enabling bounded prioritization with long-term fairness requires implementing the *BPF scheduler* itself along with an additional *admission control* module in cluster managers, and it takes *additional information on demand characteristics* from the users. A key benefit of BPF is its simplicity of implementation: we have implemented it in YARN using

[3] The proof is omitted due to space limit.

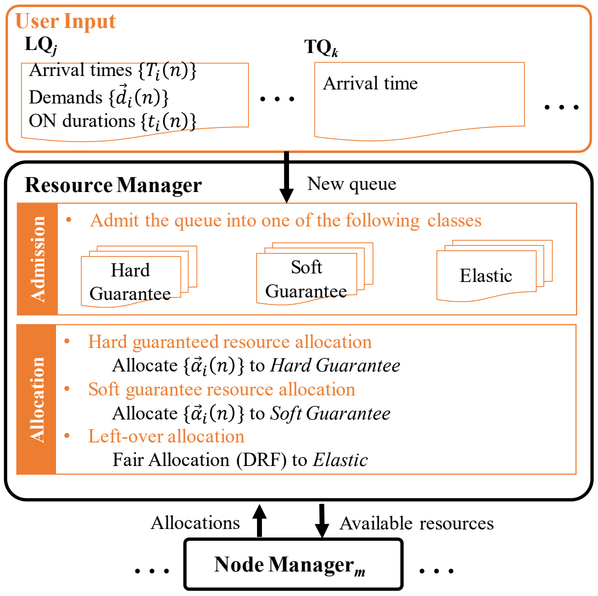

User Input
LQ$_j$
Arrival times $\{T_i(n)\}$
Demands $\{\vec{d}_i(n)\}$
ON durations $\{t_i(n)\}$
TQ$_k$
Arrival time
Resource Manager
New queue
Admission
Admit the queue into one of the following classes
Hard Guarantee
Soft Guarantee
Elastic
Allocation
Hard guaranteed resource allocation
Allocate $\{\vec{a}_i(n)\}$ to Hard Guarantee
Soft guarantee resource allocation
Allocate $\{\vec{a}_i(n)\}$ to Soft Guarantee
Left-over allocation
Fair Allocation (DRF) to Elastic
Allocations
Available resources
Node Manager$_m$

Figure 4: Enabling bounded prioritization with long-term fairness in a multi-resource cluster. BPF-related changes are shown in orange. **(TODO: change the figure ON durations to deadlines)**

only 600 lines of code. In the following, we describe how and where we have made the necessary changes.

Primer on Data-Parallel Cluster Scheduling

Modern cluster managers typically includes three components: *job manager* or application master (AM), *node manager* (NM), and *resource manager* (RM).

One NM runs on each server in the cluster, and it is responsible for managing resource containers on that server. A container is a unit of allocation and are used to run specific tasks.

For each application, a job manager or AM interacts with the RM to request job demands and receive allocation and progress updates. It can run on any server in the cluster. AM manages and monitors job demands (memory and CPU) and job status (`PENDING`, `IN_PROGRESS`, or `FINISHED`).

The RM is the most important part in terms of scheduling. It receives requests from AMs and then schedules resources using an operator-selected scheduling policy. It asks NM to prepare resource containers for the various tasks of the submitted jobs.

BPF Implementation

We made three changes for taking user input, performing admission control, and calculating resource shares – all in the RM. We do not modify NM and AM. Our implementation also requires more input parameters from the users regarding the demand characteristics of their job queues. Figure 4 depicts our design.

User Input Users submit their jobs to their queues. In our system, there are 2 queue types, i.e., LQs and TQs. We do not need additional parameters for TQs because they are the same as the conventional queues. Hence, we assume that TQs are already available in the system. However, the BPF scheduler needs additional parameters for LQs; namely, arrival times and demands.

Users submit a request job that contains their parameters of the new LQ. After receiving the parameters in the job, the RM sets up a new LQ queue for the user. Users can also ask the cluster administrator to set up the parameters.

Admission Control YARN does not support admission control. We implement an admission control module to classify LQs and TQs into Hard Guarantee, Soft Guarantee, and Elastic classes. A new queue is rejected if it cannot meet the safety condition (1), which invalids the committed performance. If it is a TQ, it is added into the Elastic class. If the new LQ does not satisfy the fairness condition (2), it is also admitted to the Elastic class. If the new LQ meets the fairness condition (2), but fails at the resource condition (3), it will be put in the Soft Guarantee class. If the new LQ meets all the three conditions, i.e., safety, fairness, and resource, it will be admitted to the Hard Guarantee class.

BPF Scheduler We implement BPF as a new scheduling policy to achieve our joint goals of bounded priority with long-term fairness. Upon registering the queues, users submit their jobs to their LQs or TQs. Thanks to admission control, LQs and TQs are classified into Hard Guarantee, Soft Guarantee, and Elastic classes. Note that resource sharing policies are implemented across queues in YARN, jobs in the same queue are scheduled in FIFO manner. Hence, BPF only sets the share at the individual queue level.

BPF Scheduler periodically set the share levels to all LQs in Hard Guarantee and Soft Guarantee classes. These share levels are actually upper-bounds on resource allocation that an LQ can receive from the cluster. Based on the real demand of each LQ, BPF allocates resources until it meets the share levels~~ (whether it is in the ON period or OFF period)~~.

BPF Scheduler allocates the resource to the three classes in the following priority order: (1) Hard Guarantee class, (2) Soft Guarantee class, and (3) Elastic class. The LQs in the Hard Guarantee class are allocated first. Then, the BPF continues allocates the resource to the LQs in Soft Guarantee class. The queues in the Elastic class are allocated with left-over resources using DRF [35].

4.2 Demand Estimation

BPF requires accurate estimates of resource demands and their durations of LQ jobs by users. These estimations can be done by using well-known techniques; e.g., users can use history of prior runs [10, 31, 37] with the assumption that resource requirements for the tasks in the same stage are similar [15, 35, 54]. We do not make any new contributions on demand estimation in this paper. When LQs have bursty arrivals of different sizes, BPF with the α-strategy ensures the performance with the average usage remains similar (§5.3). We consider a more thorough study an important future work.

4.3 Operational Issues

Container Reuse Container reuse is a well-known technique that is used in some application frameworks, such as Apache Tez. The objective of container reuse is to reduce the overheads of allocating and releasing containers. The downside is that it causes resource waste if the container to be reused is larger than the real demand of the new task. Furthermore, container reuse is not possible if the new task requires more resource than existing containers. For our implementation and deployment, we do not enable container reuse because BPF periodically prefers more free resources for LQ jobs, causing its drawbacks to outweigh its benefits in many cases.

Preemption Preemption is a recently introduced setting in the YARN Fair Scheduler [2], and it is used to kill running containers of one job to create free containers for another. By default, preemption is not enabled in YARN. For BPF, using preemption can help in providing guarantees for LQs. However, killing the tasks of running jobs often results in failures and significant delays. We do not use preemption in our system throughout this paper.

5. Evaluation

We evaluated BPF using three widely used big data benchmarks – BigBench (BB), TPC-DS, and TPC-H. We ran experiments on a 40-server CloudLab cluster [6]. We setup Tez atop YARN for the experiment.

To understand performance at a larger scale, we used a trace-driven simulator to replay jobs from the same traces. Our key findings are:

1. BPF can closely approximate the LQ performance of Strict Priority (§5.2.2) and the long-term fairness for TQs of DRF (§5.2.3).
2. BPF can provide similar benefits in the large-scale setting (§5.3).
3. BPF handles multiple LQs to accommodate bounded priority and fairness (§5.2.5).
4. When LQs have bursty arrivals of different sizes, BPF with the α-strategy ensures the performance with the average usage remains similar (§5.3).

5.1 Experimental Setup

Workloads Our workloads consist of jobs from public benchmarks – BigBench (BB) [3], TPC-DS [7], and TPC-H [8] traces. A job has multiple phases. A new phase can be executed if its prerequisite phases are finished. A phase has a number of equivalent tasks in terms of resource demand and durations. The cumulative distribution functions (CDFs) task durations across the three benchmarks are presented in Figure 5. In each experiment run, we chose the LQ jobs from one of the traces such that their shortest completion times are less than 30 seconds. We scale these jobs to make sure their instantaneous demands reach the maximum capacity of a single resource. The TQ jobs are randomly chosen from one of the traces. Each TQ job lasts from tens of seconds to tens of minutes. Each cluster experiment has 100 TQ jobs, and each simulation experiment has 500 TQ jobs. Throughout the evaluation, all the TQ jobs are submitted up at the beginning while the LQ jobs arrive sequentially. Unless otherwise specified, our default experimental setup has a single LQ and 8 TQs.

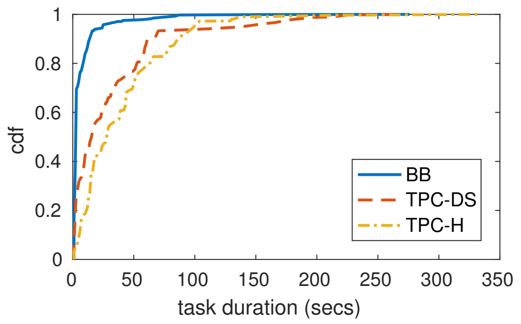


Figure 5: CDFs of task durations across workloads.

User Input Since the traces give us the resource demand and durations of the job tasks, we can set an ON period equal to the shortest completion time of its corresponding LQ job. The average of ON periods is 27 seconds across the traces. Without loss of generality, we assume that the LQ jobs arrive periodically [4]. Unless otherwise noted, the inter-arrival period of two LQ jobs is 300 seconds (1000 seconds) for the cluster experiment (the simulation experiment).

Experimental Cluster We setup Apache Hadoop 2.7.2 (YARN) on a cluster having 40 worker nodes on CloudLab [6] (40-node cluster). Each node has 32 CPU cores, 64 GB RAM, a 10 Gbps NIC, and runs Ubuntu 16.04. Totally, the cluster has 1280 CPU cores and 2.5 TB memory. The cluster also has a master node with the same specification running the resource manager (RM).

Trace-driven Simulator To have the experimental results on a larger scale, we build a simulator that mimics the system like Tez atop YARN. The simulator can replay the directed acyclic graph jobs (like Tez does), and simulate the fair scheduler of YARN at queue level. For the jobs in the same queue, we allocate the resource to them in a FIFO manner. Unlike YARN, the simulator supports 6 resources, i.e., CPU, memory, disk in/out throughputs, and network in/out throughputs.

[4] The case of aperiodic LQ jobs is similar to multiple LQs with different periods.

Compared baselines We compare BPF against the following approaches

1. **Dominant Resource Fairness (DRF)**: DRF algorithm is implemented in YARN Fair Scheduler [2]. DRF uses the concept of the dominant resource to compare multi-dimensional resources [35]. The idea is that resource allocation should be determined by the dominant share of a queue, which is the maximum share of any resource (memory or CPU). Essentially, DRF seeks to maximize the minimum dominant share across all queues.
2. **Strict Priority (SP)**: We use Strict Priority to provide the best performance for LQ jobs. In fact, we borrow the concept of "Strict Priority" from network traffic scheduling that enables Strict Priority queues to get bandwidth before other queues [51]. Similarly, we enable the LQs to receive all resources they need first, and then allocate the remaining resources to other queues. If there are conflicts among the LQs, we use DRF to share the resources among them.
3. **Naive-BPF (N-BPF)**: N-BPF is a simple version of BPF that can provide bounded performance guarantee and fairness. However, N-BPF does not support admission to Soft Guarantee. For the queues that satisfy the safety condition 1, N-BPF decides to admit them to Hard Guarantee if they meet the fairness condition 2 and resource condition 3. Otherwise, it put the queues into the Elastic class. We use N-BPF as a baseline when there are multiple LQs (§5.2.5).

Overall, SP is the upper bound in terms of performance guarantee, and DRF is the upper bound of fairness guarantee for our proposed approach.

Metrics Our primary metric is the *average completion times* (avg. compl.) of LQ jobs or TQ jobs. To show the performance improvement, we use the average completion times of LQ jobs across the three approaches. On the other hand, we use average completion times of TQ jobs to show that BPF also protects the TQ jobs. Additionally, we use *factor of improvement* to show how much BPF can speed up the LQ jobs compared to DRF as

$$\text{Factor of improvement} = \frac{\text{avg. compl. of DRF}}{\text{avg. compl. of BPF}}.$$

5.2 BPF in Testbed Experiments

5.2.1 BPF in practice

Before diving into the details of our evaluation, recall the motivational problem from Section 2.1. Figure 6 depicts how BPF solves it in the testbed. BPF enables the first two jobs of LQ to quickly finish in 141 and 180 seconds. For the two large jobs arriving at 1400 and 2000 seconds, the share is very large only in roughly 335 seconds but it is cut down to give back resource to TQ.

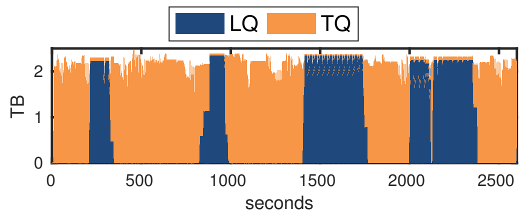


Figure 6: [Cluster] BPF's solution for the motivational problem (§2.1). The first two jobs of LQ quickly finish and the last two jobs are prevented from using too much resource. This solution is close to the optimal one.

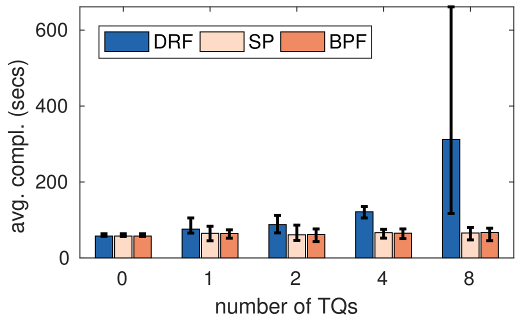


Figure 7: [Cluster] Average completion time of LQ jobs in a single LQ across the 3 schedulers when varying the number of TQs. BPF and SP guarantee the average completion time of the LQ jobs while DRF significantly suffers from the increase of number of TQs.

5.2.2 Performance guarantee

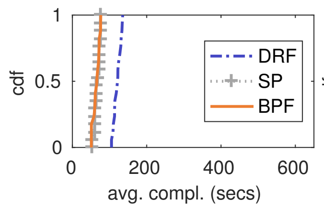
Next, we focus on what happens when there are more than one TQ. Figure 7 shows that average completion time of LQ jobs in the 40-node cluster on the BB workload. In this setting, there are a single LQ and multiple TQs. The x-axis shows the number of TQs in the cluster.

When there are no TQs, the average completion times of LQ jobs across three schedulers are the same (57 seconds). The completion times are greater than the average ON period (27 seconds) because of inefficient resource packing and allocation overheads. In practice, the resource demand of tasks cannot utilize all resources of a node that results in large unallocated resources across multiple nodes. Hence, the LQ jobs are not able to receive the whole cluster capacity as expected. More importantly, this delay is also caused by allocation overheads, such as waiting for containers to be allocated or launching containers.

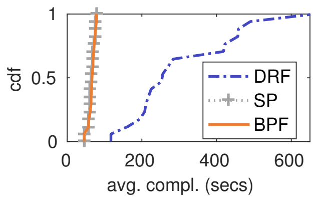
As the number of TQs increases, the performance of DRF significantly degrades because DRF tends to allocate less resource to LQ jobs. DRF is the worst among three schedulers. In contrast, BPF and SP give the highest priority to LQs that guarantees the performance of LQ jobs. The average completion times, when TQs are available (1,2,4, and 8), are almost the same (65 seconds). These average completion times are still larger than the case of no TQs

Table 3: [Cluster] Factor of improvement by BPF across various workload with respect to the number of TQs.

Workload	1 TQ	2 TQs	4 TQs	8 TQs
BB	1.18	1.42	1.86	4.66
TPC-DS	1.35	1.61	2.29	5.38
TPC-H	1.10	1.37	2.01	5.12



(a) 1 LQ & 4 TQs



(b) 1 LQ & 8 TQs

Figure 8: [Cluster] The completion time of LQ jobs is predictable using BPF.

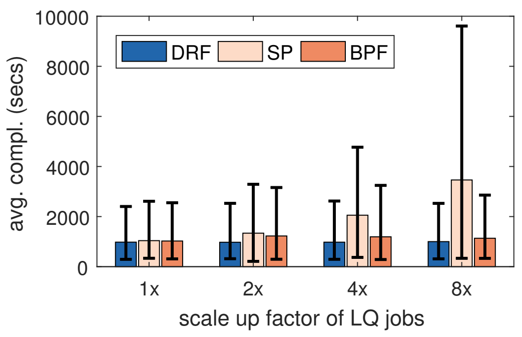


Figure 9: [Cluster] BPF protects the batch jobs up to $3.05\times$ compared to SP.

because of non-preemption. The LQ jobs are not able to receive the resources that are still used by the running tasks.

To understand how well BPF performs on various workload traces, we carried out the same experiments on TPC-DS and TPC-H. As SP and BPF achieve the similar performance, we only present the factors of improvement of BPF across the various workloads in Table 3. The numbers on the table show the consistent improvement on the average completion time of LQ jobs.

In addition to the average completion time, we evaluated the performance of individual LQ jobs. Figure 8 shows that cumulative distribution functions (cdf) of the completion times across 3 approaches. Figure 8a and 8b are the experimental results for the cases of 4 TQs and 8 TQs, respectively. We observe that the completion times of LQ jobs in DRF are not stable and vary a lot when the number of LQs becomes large as in Figure 8b. The unstable performance is caused by the instantaneous fairness and the variance of total resource demand.

5.2.3 Fairness guarantee

Figure 9 shows the average completion time of TQ jobs when we scale up the number of tasks of LQ jobs are by 1x, 2x, 4x, and 8x. In this experiment, there are a single LQ and 8 TQs.

Since DRF is a fair scheduler, the average completion times of TQ jobs are almost not affected by the size of LQ jobs. However, SP allocates too much resource to LQ jobs that significantly hurts TQ jobs. Since SP provides the highest priority for the LQ jobs, it makes the TQ jobs to starve for resources. BPF performs closely to DRF. While DRF maintains instantaneous fairness, BPF maintains the long-term fairness among the queues.

5.2.4 Scheduling overheads

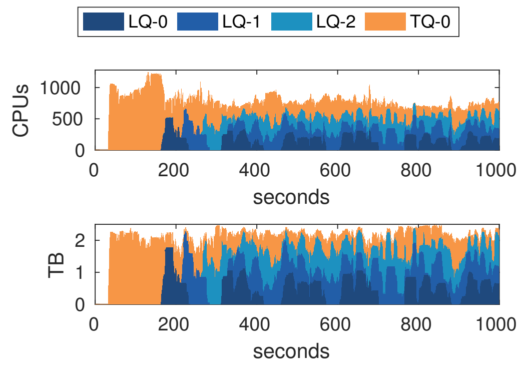
Recall from Section 4 that the BPF scheduler has three components: user input, admission control, and allocation. Compared to the default schedulers in YARN, our scheduler has additional scheduling overheads for admission control and additional computation in allocation.

Since we only implement our scheduler in the Resource Manager, the scheduling overheads occur at the master node. To measure the scheduling overheads, we run admission control for 10000 LQ queues and 10000 TQ queues on a master node – Intel Xeon E3 2.4 GHz (with 12 cores). Each LQ queue has 500 ON/OFF cycles. Recall the LQADMIT and TQADMIT functions in Algorithm 1, the admission overheads increase linearly to the number of queues. The total admission overheads are approximately 1 ms, which is much less than the default update interval in YARN Fair Scheduler, i.e., 500 ms [2]. The additional computation in allocation is also negligibly less than 1 ms.

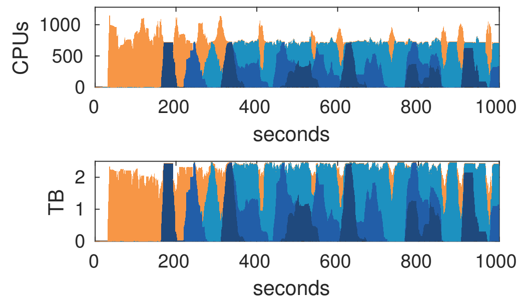
5.2.5 Admission control for multiple LQs

To demonstrate how BPF works with multiple LQs, we set up 3 LQs (LQ-0, LQ-1, and LQ-2) and a single TQ (TQ-0). The jobs TQ-0 are queued up at the beginning while LQ-0, LQ-1, and LQ-2 arrive at 50, 100, and 150 seconds, respectively. The periods of LQ-0, LQ-1, and LQ-2 are 150, 110, and 60 secs. All the LQs jobs have the identical demand and task durations. The TQ jobs are chosen from the BB benchmark. BPF admits LQ-0 to the Hard Guarantee class, LQ-1 to the Soft Guarantee class, and LQ-2 to the Elastic class.

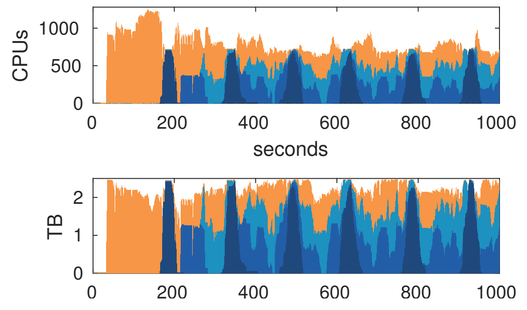
Figure 10a shows the resource usage (CPU and memory) for each queue across four schedulers, i.e., DRF, SP, N-BPF and BPF. As an instantaneously fair scheduler, DRF continuously maintains the fair share for all queues as in Figure 10a. Since LQ-2 requires a lot of resources, SP makes TQ-0 starving for resources (Figure 10b). N-BPF provides LQ-0 with resource guarantee and it fairly share the resources to LQ-1, LQ-2, and TQ-0 (Figure 10c). BPF provides hard guarantee to LQ-0 and soft guarantee to LQ-1 as in Figure 10d. The soft guarantee allows LQ-1 performs better than using N-



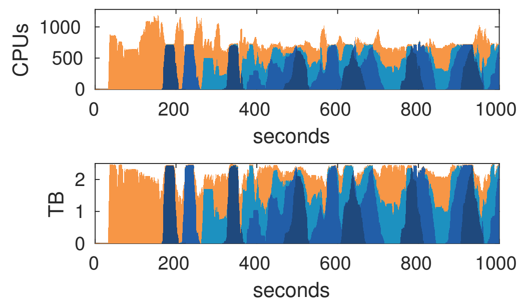
(a) DRF: LQ-0, LQ-1, LQ-2 are unhappy with high latency.



(b) SP: TQ-0 is starving of resources.



(c) N-BPF: Only LQ-0 and TQ-0 are happy.



(d) BPF: LQ-0, LQ-1 and TQ-1 are happy.

Figure 10: [Cluster]. DRF and SP fail to guarantee both performance and fairness simultaneously. BPF gives the best performance to LQ-0, near optimal performance for LQ-1, and maintains fairness among 4 queues. LQ-2 requires too much resource, so its performance cannot be guaranteed.

BPF. Since LQ-2 demands too much resources, BPF treats it like TQ-0.

Workload	Number of TQs					
	1	2	4	8	16	32
BB	1.08	1.56	2.32	4.09	7.28	16.61
TPC-DS	1.06	1.38	1.66	2.93	5.16	10.40
TPC-H	1.01	1.28	1.92	3.04	5.50	11.35

Table 4: [Simulation] Factors of improvement by BPF across various workloads w.r.t the number of TQs.

Figure 11 shows the average completion time of jobs on each queue across the four schedulers. The performance of DRF for LQ jobs is the worst among the four schedulers but it is the best for only TQ-0. The performance of SP is good for LQ jobs but it is the worst for TQ jobs. N-BPF provides the best performance for LQ-0 but not LQ-1 and LQ-2. BPF is the best among the four schedulers. The three of four queues, i.e., LQ-0, LQ-1, and TQ-0, significantly benefit from BPF. BPF even outperforms SP for LQ-0 and LQ-1 jobs and does not hurt any TQ.

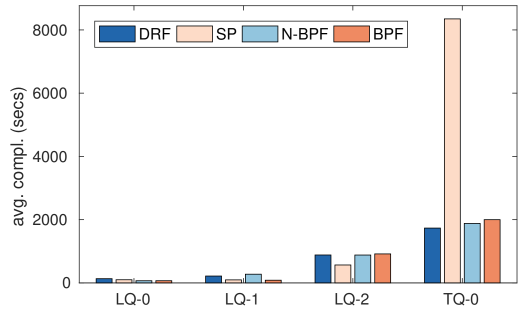


Figure 11: [Cluster] BPF provides with better performance for LQs than DRF and N-BPF. Unlike SP, BPF protects the performance of TQ jobs.

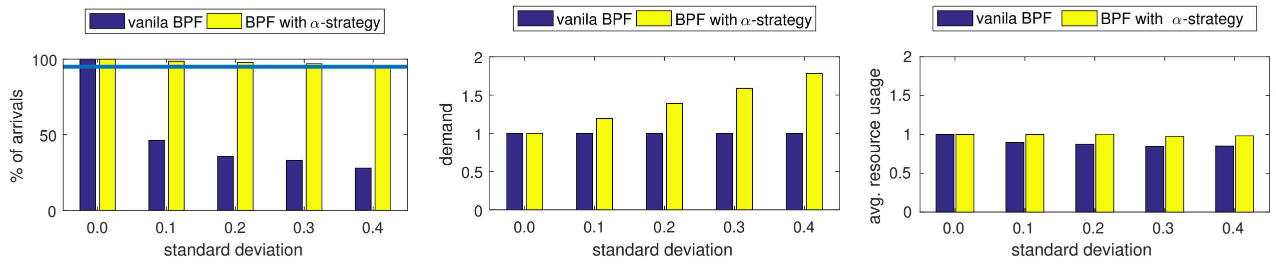
5.3 Performance in Trace-Driven Simulations

To verify the correctness of the large-scale simulator, we replayed the BB trace logs from cluster experiments in the simulator. Table 4 shows the factors of improvement in completion times of LQ jobs for BB workload in simulation that are consistent with that from our cluster experiments (Table 3).

BPF significantly improves over DRF when we have more TQs. We note that the factors of improvement for TPC-DS and TPC-H in the simulation are less that of the cluster experiments. It turns out that DRF in TCP-DS and TPC-H suffers from the allocation overheads that our simulation does not capture. The allocation overheads for the LQ jobs in TPC-DS and TPC-H are large because they have more phases than the LQ jobs in BB (only 2 phases).

We use the large-scale simulator to study the impact of estimation errors and non-preemption on LQ jobs. In both cases, BPF still outperforms DRF significantly. Results are omitted due to space limit.

Performance of the α-strategy. Figure 12 depicts the requested demand, performance, and resource usage under the



(a) Percentage of arrivals completed by the deadlines.

(b) Requested demand normalized by that under the vanilla BPF.

(c) Resource consumption of LQ.

Figure 12: [Simulation] The proposed α-strategy under α=95% is robust against the uncertainties.

vanilla BPF and the one with the α-strategy when arrivals have different sizes. In particular, as the variance increases, the vanilla BPF can no longer complete α arrivals before the deadline. Actually, even with 10% standard deviation, the percentage drops below 50%. On the other side, BPF with α-strategy always satisfy the α requirement. Even though the reported demand increases, the average resource usage does not change much, e.g., TQ still receives the same long-term share.

6. Related Work

Bursty Applications in Big Data Clusters Big data clusters experience burstiness from a variety of sources, including periodic jobs [10, 14, 31, 58], interactive user sessions [13], as well as streaming applications [9, 12, 65]. Some of them show predictability in terms of inter-arrival times between successive jobs (e.g., Spark Streaming [65] runs periodic mini batches in regular intervals), while some others follow different arrival processes (e.g., user interactions with large datasets [13]). Similarly, resource requirements of the successive jobs can sometimes be predictable, but often it can be difficult to predict due to external load variations (e.g., time-of-day or similar patterns); the latter, without BPF, can inadvertently hurt batch queues (§2).

Multi-Resource Job Schedulers Although early jobs schedulers dealt with a single resource [16, 44, 66], modern cluster resource managers, e.g., Mesos [41], YARN [61], and others [23, 58, 62], employ multi-resource schedulers [19, 21, 33, 35, 37, 38, 50] to handle multiple resources and optimize diverse objectives. These objectives can be fairness (e.g., DRF [35]), performance (e.g., shortest-job-first (SJF) [33]), efficiency (e.g., Tetris [37]), or different combinations of the three (e.g., Carbyne [38]). However, *all* of these focus on instantaneous objectives, with instantaneous fairness being the most common goal. To the best of our knowledge, BPF is the first multi-resource job scheduler with long-term memory.

Handling Burstiness Supporting multiple classes of traffic is a classic networking problem that, over the years, have arisen in local area networks [20, 32, 59, 60], wide area networks [42, 47, 52], and in datacenter networks [40, 48]. All of them employ some form of admission control to provide quality-of-service guarantees. They also consider only a single link (i.e., a single resource). In contrast, BPF considers multi-resource jobs and builds on top this large body of existing literature.

BVT [30] was designed to work with both real-time and best-effort tasks. Although it prioritizes the real-time tasks, it cannot guarantee performance and fairness.

Expressing Burstiness Requirements BPF is not the first system that allows users to express their time-varying resource requirements. Similar challenges have appeared in traditional networks [60], network calculus [27, 28], datacenters [17, 48], and wide-area networks [52]. Akin to them, BPF requires users to explicitly provide their burst durations and sizes; BPF tries to enforce those requirements in short and long terms. Unlike them, however, BPF explores how to allow users to express their requirements in a multi-dimensional space, where each dimension corresponds to individual resources. One possible way to collapse the multi-dimensional interface to a single dimension is using the notion of *progress* [24, 35]; however, progress only applies to scenarios when a user's utility can be captured using Leontief preferences.

7. Conclusion

To enable the coexist of latency-sensitive LQs and the TQs, we proposed BPF (Bounded Priority Fairness). BPF provides bounded performance guarantee to LQs and maintains the long-term fairness for TQs. BPF classifies the queues into three classes: Hard Guarantee, Soft Guarantee and Elastic. BPF provides the best performance to LQs in the Hard Guarantee class and the better performance for LQs in the Soft Guarantee class. The scheduling is executed in a strategyproof manner, which is critical for public clouds. The queues in the Elastic class share the left-over resources to maximize the system utilization. In the deployments, we show that BPF not only outperforms the DRF up to $5.38\times$ for LQ jobs but also protects TQ jobs up to $3.05\times$ compared to Strict Priority. When LQ's arrivals have different sizes, adding the α-strategy can satisfy the deadlines with similar resource utilization.

References

[1] Apache Hadoop. `http://hadoop.apache.org`.

[2] YARN Fair Scheduler. `http://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`, 2014.

[3] Big-Data-Benchmark-for-Big-Bench. `https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench`, 2016.

[4] Presto: Distributed SQL Query Engine for Big Data. `https://prestodb.io/`, 2016.

[5] Apache Tez. `http://tez.apache.org`, 2017.

[6] Cloudlab. `http://www.cloudlab.us/`, 2017.

[7] TPC Benchmark DS (TPC-DS). `http://www.tpc.org/tpcds`, 2017.

[8] TPC Benchmark H (TPC-H). `http://www.tpc.org/tpch`, 2017.

[9] Trident: Stateful stream processing on Storm. `http://storm.apache.org/documentation/Trident-tutorial.html`, 2017.

[10] S. Agarwal, S. Kandula, N. Burno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data parallel computing. In *NSDI*, 2012.

[11] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[12] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at Internet scale. 2013.

[13] S. Alspaugh, B. Chen, J. Lin, A. Ganapathi, M. Hearst, and R. Katz. Analyzing log analysis: An empirical study of user log mining. In *LISA*, 2014.

[14] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed popularity content in MapReduce clusters. In *EuroSys*, 2011.

[15] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[16] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in MapReduce clusters using Mantri. In *OSDI*, 2010.

[17] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *OSDI*, 2014.

[18] N. Bansal and M. Harchol-Balter. *Analysis of SRPT scheduling: Investigating unfairness*, volume 29. ACM, 2001.

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

[20] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475 (Informational), Dec. 1998. Updated by RFC 3260.

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

[22] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Data Engineering*, 2015.

[23] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.

[24] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[25] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[26] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[27] R. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[28] R. Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[29] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[30] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *ACM SIGOPS Operating Systems Review*, 33(5):261–276, 1999.

[31] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Eurosys*, 2012.

[32] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.

[33] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[34] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. *SIGCOMM*, 2012.

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

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

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

[38] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[39] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[40] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can JUMP them! In *NSDI*, 2015.

[41] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[42] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[43] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[44] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[45] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[46] P. Jaillet and M. R. Wagner. *Online Optimization*. Springer Publishing Company, Incorporated, 2012.

[47] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[48] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message completion time in the cloud. In *SIGCOMM*, 2015.

[49] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.

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

[51] L. Kleinrock and R. Gail. *Queueing systems: Problems and Solutions*. Wiley, 1996.

[52] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, et al. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM*, 2015.

[53] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[54] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A progress indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[55] H. Moulin. *Cooperative microeconomics: a game-theoretic introduction*. Princeton University Press, 2014.

[56] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[57] J. Schneider and S. Kirkpatrick. *Stochastic Optimization*. Springer Science & Business Media, 2007.

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

[59] S. Shenker, D. D. Clark, and L. Zhang. A scheduling service model and a scheduling architecture for an integrated services packet network. Technical report, Xerox PARC, 1993.

[60] I. Stoica, H. Zhang, and T. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *SIGCOMM*, 1997.

[61] 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 *SoCC*, 2013.

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

[63] 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 *EuroSys*, 2010.

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

[65] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[66] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.