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

Paper #: 212, 12 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 BPF (Bounded Priority Fairness). 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. Deployments and large-scale simulations show that BPF closely approximates the performance of Strict Priority, and the fairness 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

As distributed data-parallel computing matures, throughput-sensitive batch processing systems [1, 38, 52, 62] are increasingly being complemented by latency-sensitive interactive analytics [7, 16, 72] and online stream processing systems [11, 17, 29, 64, 73]. To enable data sharing and seamless transition between these models, all these applications coexist on shared clusters [4–6, 8, 25, 50, 69]. Naturally, both systems and networking communities have focused on resource sharing in these clusters [12, 24, 28, 34, 50, 69].

Today's schedulers are multi-resource [20, 33, 43, 45, 58], DAG-aware [32, 45, 72], and allow a variety of constraints [22, 44, 53, 71, 74]. Given all these inputs, they optimize for objectives such as fairness [26, 42, 43, 54], performance [41], efficiency [45], or different combinations of the three [47, 48]. However, all existing schedulers have one thing in common: *they force the same performance goal on all jobs*.

We observe that even though batch, interactive, and streaming computation models all care about performance, their notions of performance are different. For instance, the average completion time can sufficiently capture the performance of a throughout-sensitive batch-job queue (TQ) [22, 45, 47, 74]. However, interactive sessions and streaming applications form latency-sensitive queues (LQ): each LQ is a sequence of small jobs following an ON-OFF pattern. For these jobs [7, 18, 72, 73], individual completion times or latencies are far more important than the average completion time or the throughput of the LQ.

Indeed, existing "fair" schedulers are inherently unfair to LQ jobs: when LQ jobs are present (ON state), they must share the resources equally with TQ jobs, but when they are absent (OFF state), batch jobs get all the resources. The fact that LQ jobs are not using any resources during their OFF periods is completely ignored. In the long run, TQs receive more resources than their fair shares because today's schedulers make *instantaneous* decisions (§2).

One may assume that assigning higher priorities to LQ jobs (Strict Priority, or SP [60]) would resolve this. Although simple and attractive, prioritization cannot address this challenge because it incentivizes LQs to submit arbitrarily large jobs to starve TQs. Consequently, the performance-isolation tradeoff requires a careful consideration.

Additionally, how one reasons about system utilization becomes more complicated in the presence of both LQs and TQs. While it is relatively easy to fill the cluster with many batch jobs to achieve high utilization, LQs often have higher value and are harder to be accommodated with resource guarantees. Compared to a simple solution that rejects most of LQs and/or serves them with best efforts, system operators are likely to prefer to maximize the LQs served with resource guarantees, which leads to more predictable performance.

Therefore, we ponder a fundamental question: *can we enable latency-sensitive LQs and throughput-sensitive TQs to coexist, where LQs are permitted short-term resource bursts while ensuring long-term fairness between the two, as well as maximizing LQs served with resource guarantees?*

The need to separate throughput- and latency-sensitive job

queues in big data clusters is akin to the classic networking problem of supporting throughput-sensitive flows and latency-sensitive realtime communication on network access links [37, 40, 67, 68]. The key difference between the two is the multi-resource nature of cluster scheduling and challenges arising from it. We target at a solution that can be applied to multi-resourced clusters.

In this paper, we first explore opportunities and challenges in achieving three desired properties in this context (§2): (i) allowing LQs to enjoy short-term high resource usage during their ON periods to minimize *individual* job response times; (ii) maintaining the same *long-term* fairness as existing "fair" allocation policies by preventing arbitrarily large bursts; and (iii) maximizing system utilization and LQs served with resource guarantees.

The key opportunity lies in the observation that TQs do not care about allocated resources in the short term, as long as the long-term averaged resource share is the same. This temporal flexibility allows us to accommodate bursts of LQs without hurting TQs. However, there are several challenges to achieve these desired properties (§2.2). There is a fundamental tradeoff between "hard" resource guarantee to LQs and isolation protection for TQs. In addition, naive admission control may result in very few LQs admitted with resource guarantees, leading to suboptimal performance for LQs even with a high system utilization.

We present BPF in Section 3 to find a balance on the potentially conflicting properties. Given the performance-isolation tradeoff, BPF takes long-term fairness as the hard requirement as otherwise TQs may starve, which is not acceptable. Under the isolation requirement, BPF tries to optimize the other two metrics: individual completion time of LQs and number of LQs admitted with resource guarantees.

In BPF, an LQ reports on its arrival the length of its ON and OFF cycles, as well as the amount and duration of (multiple typed) resources requested. An upper limit on the cycle length is set to respect the fair share of TQs in a timely manner. BPF then decides whether to admit the LQ by checking the safety condition, i.e., admitting this LQ does not invalidate any prior resource guarantees committed. Then, if its total requested resources exceed its long-term fair share, the LQ is admitted and provided only with its long-term fair share. Otherwise, the LQ is admitted with resource guarantees. BPF provides both hard and soft guarantees. Intuitively, if there are enough available resources for every burst of the LQ, it is admitted with hard guarantee, i.e., the system guarantees to provide the requested amount of resources during each requested interval accordingly. Otherwise, the LQ is admitted with soft guarantee, i.e., the requested amount of resources are provided as long as there is no conflict with LQs with hard guarantee.

Having the queue with soft guarantee is crucial to minimize the individual completion time of LQs. While LQs with hard guarantee are provided with performance comparable to that under SP, LQs with soft guarantee can still have better performance compared to that under existing fair allocation policies.

For long-term fairness and isolation, BPF guarantees a fixed amount of resources to each admitted LQ instead of extending unlimited resources by prioritization. If some LQ increases their demand beyond her resource guarantee, that demand is only served when the system has unused resources without hurting the isolation for TQs and other LQs.

We have implemented BPF on Apache YARN [69] (§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.

We deployed and evaluated BPF on 40 bare-metal machines using the public benchmarks, i.e., BigBench (BB) [2], TPC-DS [9], and TPC-H [10]. In deployments, BPF significantly provides up to $5.38\times$ (totally 9 queues in a cluster) faster performance with smaller variation for LQ jobs than DRF, while maintaining the fairness among the queues (§5.2). In fact, the performance of LQ jobs is close to that under SP. On the other hand, BPF maintains the fairness among the queues and does not hurt the long-term averaged resources received by TQs. BPF protects the TQ jobs up to $3.05\times$ compared to SP. Furthermore, BPF performs well even in the presence of moderate misestimations of task demands and without preemption allowed.

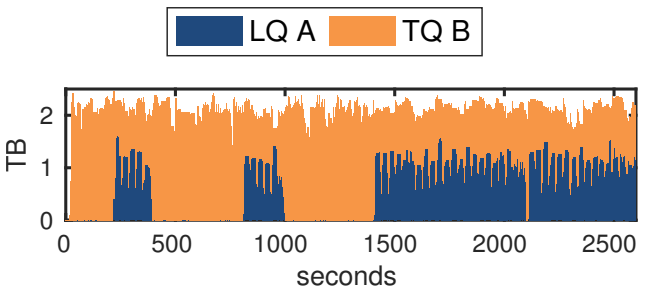
2 Motivation

2.1 Benefits of Temporal Co-scheduling

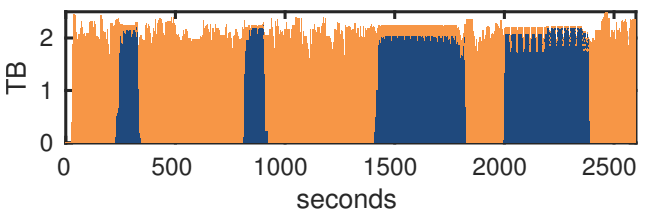
Consider Figure 1 that illustrates the inefficiencies of existing resource allocation policies, in particular, DRF [43] and Strict Priority (SP) [60] for coexisting LQs and TQs. For this experiment, we run memory-bound jobs in Tez atop YARN within a cluster of 40 nodes, where each node has 32 CPU cores and 64 GB RAM.

There are two queues: LQ A and TQ B. LQ A has MapReduce jobs submitted every 10 minutes. Her demand increases from her 3rd burst and exceeds her fair share. TQ B has jobs that are generated from the BigBench workload [2] and queued up at the beginning. For presentation simplicity, all jobs are memory-bound. While LQ A tries to complete each burst as quickly as possible, TQ B cares more about her long-term averaged resources received, e.g., every 10 minutes.

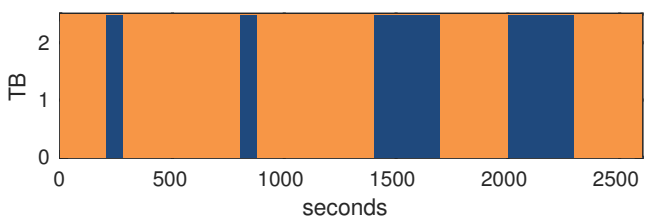
The memory resource consumption under DRF and SP is depicted in Figure 1a and 1b, respectively. DRF enforces instantaneous fair allocation of resources at any time. During the burst of LQ A, LQ A and TQ B share the bottleneck resource (memory) evenly until the jobs from LQ A complete; then TQ B gets all system resources before the next burst of LQ A. Clearly, TQ B is happy at the cost of longer completion time of LQ A's jobs. SP gives LQ A the whole system resources (high priority) whenever she has jobs to run and hence, provides lowest possible completion time. The reduction in completion time is significant (around



(a) DRF – While instantaneous fairness is enforced, the completion time of LQ jobs is much higher than that under SP.



(b) Strict Prority (SP)– While the completion time of LQ jobs are reduced, TQ B does not receive her fair share after LQ A increases her demand after 1,400 seconds.



(c) Ideal – The first two LQ jobs finish as quickly as possible. The latter jobs cannot use more resources than LQ A's fair share.

Figure 1: Motivation example

40%). However, this comes at the expense of TQ B's resource consumption smaller than her fair share after 1,400 seconds (only 1/3). In other words, LQ A can arbitrarily increase her demand without any punishment and as a result, there is no isolation protection for TQ B. The situation for SP becomes even worse when multiple LQs are present.

DRF and SP can be considered as two extremes to resolve the tradeoffs between fairness (for TQs) and performance (for LQs), and fails to accommodate the requirements from both queues simultaneously. Other existing solutions are incapable of achieving these heterogenous performance metrics either. For instance, any policy without taking the differences between LQs and TQs into consideration, such as FIFO, renders either LQs or BQs, or even both, unhappy. Any fair scheduler such as DRF and HUG [31] cannot handle the burst of jobs during the ON state of LQs. In contrast, priority-based solutions may lead to the starvation of TQs. Interestingly, DRF allows different weights to be set for each LQ and TQ. In the current DRF policy, however, weights are static over time. Therefore, lower weights for LQs cannot handle the burst, while high weights may lead to TQ starvation and difficulties with multiple LQs, similar to SP. Nonetheless, it provides a critical control knob that was

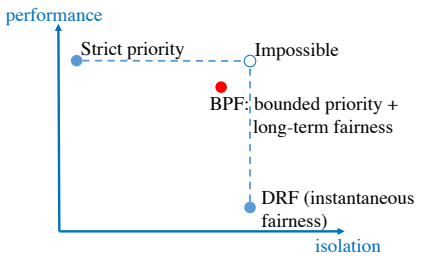


Figure 2: Design space – BPF provides LQs with bounded priority and maintains long-term fairness.

leveraged in our proposed system BPF.

The ideal allocation is depicted in Figure 1c. The key idea is "bounded" priority for LQs: as long as the burst is not too large to hurt the fair share of TQs, they are given higher priority so jobs can be completed as quickly as possible. In particular, before 1,400 seconds, LQ A's bursts are small, so she gets higher priority, which is similar to SP. After LQ A increases her demand, only a fraction of her demand can be satisfied with the entire system's resources. Then she has to give resources back to TQ B to ensure the long-term fairness.

2.2 Desired Properties

We restrict our attentions in this paper to the following three properties: completion time for LQs, fairness/isolation for TQs, and system utilization.

The key observation that motivates the research is that we cannot simultaneously provide the lowest completion time for all LQs and fairness/isolation for TQs. Figure 2 depicts the fundamental tradeoffs. In particular, SP provides the best performance (lowest completion time) to LQs without any isolation protection for TQs. DRF provides the best isolation (instantaneous fairness) without considering the short-term burst requirement for LQs. It is impossible to obtain the best performance and instantaneous fairness at the same time because instantaneous fairness cannot accommodate short-term burst of LQs. Therefore, long-term fairness is enforced instead to give room for the short-term burst requirement for LQs. On the other hand, the burst size of LQs cannot be too large compared to her long-term fair share. For instance, consider a link with capacity 100 MBps and shared by one LQ and one TQ. The LQ has ON status every 10 seconds. If her demand during the ON status is no larger than 500MB (0.5*100MBps*10s), we can prioritize this LQ. However, if her demand exceeds 500MB, we cannot give her high priority without hurting TQ's long-term fair share.

The key question of the paper is, therefore, how to allocate system resources in a near-optimal way; meaning, long-term fairness is always ensured, while LQs receive high priority as long as their bursts do not exceed their long-term fair share.

Challenge: Increase LQ Utilization

Another metric beyond the performance-isolation tradeoff is the system utilization. However, it is more complicated

when LQs and TQs coexist. Often, LQs have high value and are time sensitive. Therefore, they are hard to be accommodated with resource guarantees. On the other hand, it is relatively easy to fill the cluster with (possible) low value, batch jobs. In other words, system utilization is no longer the sole metric and the composition of it needs to be examined.

In this spirit, we call the utilization of LQs with resource guarantees LQ utilization, which is the third performance metric that operators are likely to be interested in.

3 BPF: A Scheduler With Memory

In this section, we model problem settings in Section 3.1 and present formal definitions of the desired properties and the optimization in Section 3.2. BPF achieves the desired properties by admission control, guaranteed resource provision, and spare resource allocation presented in Section 3.3.

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 $\overrightarrow{C}(t)$ as long as $\overrightarrow{C}(t)$ can be estimated at the beginning. Stochastic optimization [65] and online algorithm design [55] can be leveraged to cases where $\overrightarrow{C}(t)$ is hard to predict, which is beyond the scope of this paper.

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 of jobs during her ON status. We denote by $T_i(n)$ and $t_i(n)$ the starting time and duration of her n-th ON period, respectively. Therefore, her n-th OFF period is $[T_i(n) + t_i(n), T_i(n+1)]$. We also denote the demand during her n-th ON period 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, the duration of each ON/OFF period $T_i(n+1) - T_i(n)$ can be fixed for some applications such as Spark Streaming, 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. We assume users report their estimated demand for presentation simplicity. Actually, a stochastic optimization problem can be solved by each LQ to decide the vector she wants to report.

Recall that the performance metric of the LQs is the average completion time of jobs during each ON period. Therefore, each LQ requests the resource demand $\overrightarrow{d_i}(n)$ for a duration that equals to the length of her ON period $t_i(n)$.

To enforce the long-term fairness, her total demand during each period $[T_i(n), T_i(n+1)]$, i.e., $\overrightarrow{d_i}(n) * t_i(n)$, should not exceed her fair share, which can be calculated by fair scheduler, i.e., $\frac{\overrightarrow{C}}{N}$ when there are N queues admitted by BPF, or more complicated scheduler such as DRF. We adopt the easier one as it provides a more conservative evaluation of the improvements brought by BPF.

TQs' jobs are queued at the beginning with much larger demand than each burst of LQs. In contrast to LQs who wish to finish each burst as quickly as possible in each ON/OFF period lasting for several minutes, TQs only care about the resources received in the long term, e.g., hours.

Table 1: Important notations

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

3.2 Desirable Properties

Completion time

Let us denote by $R_i(n)$ the average completion time of jobs during LQ-i's n-th ON period. Then we want to minimize

$$\frac{1}{|\mathbb{H} \cup \mathbb{S}|} \sum_{i \in \mathbb{H} \cup \mathbb{S}} \left(\frac{1}{N_i} \sum_{n=1}^{N_i} R_i(n) \right),$$

where $\mathbb{H} \cup \mathbb{S}$ is the set of admitted LQs with resource guarantees, either hard or soft, and N_i is the number of ON/OFF periods for the evaluation interval.

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 $\frac{1}{T}\int_t^{t+T} \overrightarrow{a_i}(t)\mathrm{d}t$ and $\frac{1}{T}\int_t^{t+T} \overrightarrow{e_j}(t)\mathrm{d}t$. 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 a LQ. Formally, $\forall i \in \mathbb{A}, \forall j \in \mathbb{B}$, where $\mathbb{B}$ is the set of admitted TQs.

$$\max_k \left\{ \frac{1}{T} \int_t^{t+T} a_i^k(t)\mathrm{d}t \right\} \leq \max_k \left\{ \frac{1}{T} \int_t^{t+T} e_j^k(t)\mathrm{d}t \right\},$$

where $a_i^k(t)$ and $e_j^k(t)$ are allocated type-k resources for LQ-i and TQ-j at time t, respectively. This condition provides long-term isolation protections for admitted TQs.

System utilization and LQ utilization

At any time t, denote by $\overrightarrow{f_i}(t)$ and $\overrightarrow{g_j}(t)$ the resource consumed by LQ-i and TQ-j, respectively. Over the evaluation interval $[t, t+T]$, the average system utilization for all resources can be calculated as follows:

$$\frac{1}{KT} \sum_k \int_t^{t+T} \frac{\sum_{i \in \mathbb{A}} f_i^k(t) + \sum_{i \in \mathbb{B}} g_j^k(t)}{C^k},$$

where $f_i^k(t)$ and $g_j^k(t)$ are allocated type-k resource for LQ-i and TQ-j at time t, respectively. The LQ utilization is simply the part consumed by LQs with guarantees

$$\frac{1}{KT}\sum_k \int_t^{t+T} \frac{\sum_{i\in\mathbb{H}\cup\mathbb{S}} f_i^k(t)}{C^k}.$$

Optimization problem

We would like to minimize the completion time of jobs for admitted LQs with guarantees, maximize the system utilization and LQ utilization while keeping the long-term fairness. The problem can be expressed as follows:

$$\begin{aligned} \min \quad & \frac{1}{|\mathbb{H}\cup\mathbb{S}|}\sum_{i\in\mathbb{H}\cup\mathbb{S}}\left(\frac{1}{N_i}\sum_{n=1}^{N_i} R_i(n)\right) \\ \max \quad & \frac{1}{KT}\sum_k \int_t^{t+T} \frac{\sum_{i\in\mathbb{A}} f_i^k(t) + \sum_{i\in\mathbb{B}} g_j^k(t)}{C^k} \\ \text{s.t.} \quad & \max_k\left\{\frac{1}{T}\int_t^{t+T} a_i^k(t)\mathrm{d}t\right\} \le \\ & \max_k\left\{\frac{1}{T}\int_t^{t+T} e_j^k(t)\mathrm{d}t\right\}, \forall i\in\mathbb{A}, \forall j\in\mathbb{B} \end{aligned} \tag{1}$$

The decisions to be made are the set of admitted LQs ($\mathbb{A}$, $\mathbb{H}$, $\mathbb{S}$), the set of admitted TQs ($\mathbb{B}$), and resources allocated to LQ-i and TQ-j ($\overrightarrow{a_i}(t)$ and $\overrightarrow{e_j}(t)$, respectively) over time. The resource consumptions ($\overrightarrow{f_i}(t)$ and $\overrightarrow{g_j}(t)$) are different from resource allocations ($\overrightarrow{a_i}(t)$ and $\overrightarrow{e_j}(t)$) because some queues cannot use up the allocated resource, and if there are some unused/unallocated resources, queues with unsatisfied demand can share them, in addition to their allocated resources. LQ completion time $R_i(n)$ is some non-increasing function of resource consumption $\overrightarrow{f_i}(t)$, i.e., more resources reduce completion time.

3.3 Solution Approach

Our solution BPF consists of three major components for the decision variables: admission control procedure to decide $\mathbb{H}$, $\mathbb{S}$ and $\mathbb{E}$ for long-term fairness, guaranteed resource provisioning procedure for $\overrightarrow{a_i}(t)$ to minimize completion time of LQs, and spare resource allocation procedure to decide $\overrightarrow{f_i}(t)$ and $\overrightarrow{g_j}(t)$ to maximize utilization.

Admission control procedure

BPF classifies admitted LQs and TQs 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 short-term resource guarantee, but long-term fair share is provided.

Algorithm 1 BPF Scheduler

procedure PERIODICSCHEDULE()
  if there are new LQs, $\mathbb{Q}$ **then**
    $\{\mathbb{H},\mathbb{S},\mathbb{E}\}$ =LQADMIT($\mathbb{Q}$)
  end if
  if there are new TQs, $\mathbb{Q}$ **then**
    $\{\mathbb{E}\}$ =TQADMIT($\mathbb{Q}$)
  end if
  ALLOCATE($\mathbb{H},\mathbb{S},\mathbb{E}$)
end procedure

function LQADMIT(LQs $\mathbb{Q}$)
  for all LQ $Q\in\mathbb{Q}$ **do**
    if safety condition (2) satisfied **then**
      if fairness conditions (3) satisfied **then**
        if resource conditions (4) satisfied **then**
          Update $\mathbb{H}=\mathbb{H}\cup Q$ *//Q is admitted with hard guarantee*
        else
          Update $\mathbb{S}=\mathbb{S}\cup Q$ *//Q is admitted with soft guarantee*
        end if
      else
        Update $\mathbb{E}=\mathbb{E}\cup Q$ *//Q is admitted with long-term fair share only*
      end if
    end if
  end for
  return $\{\mathbb{H},\mathbb{S},\mathbb{E}\}$
end function

function TQADMIT(QUEUE $\mathbb{Q}$)
  for all TQ $Q\in\mathbb{Q}$ **do**
    if safety condition (2) satisfied **then**
      Update $\mathbb{E}=\mathbb{E}\cup Q$
    end if
  end for
  return $\{\mathbb{E}\}$
end function

function ALLOCATE($\mathbb{H}$, $\mathbb{S}$, $\mathbb{E}$)
  for all LQ $Q\in\mathbb{H}$ **do**
    $\overrightarrow{a_i}(t)=\overrightarrow{d_i}(n)$ for $t\in[T_i(n),T_i(n+1)]$ until Q's consumption reaches $\overrightarrow{d_i}(n)*t_i(n)$.
  end for
  for all LQ $Q\in\mathbb{S}$ **do**
    $\overrightarrow{a_i}(t)=\min\{\overrightarrow{d_i}(n),\overrightarrow{C}-\sum_{j\in\mathbb{H}\cup\mathbb{S}'}\overrightarrow{d_j}(t)\}$ for $t\in[T_i(n),T_i(n+1)]$ until Q's consumption reaches $\overrightarrow{d_i}(n)*t_i(n)$.
  end for
  Obtain the remaining resources $\overrightarrow{L}$
  DRF($\mathbb{E}$, $\overrightarrow{L}$)
end function

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

$$\overrightarrow{d_j}(n) * t_j(n) \leq \frac{\overrightarrow{C}\,(T_j(n+1) - T_j(n))}{|\mathbb{H}| + |\mathbb{S}| + |\mathbb{E}| + 1}, \forall n, \forall j \in \mathbb{H} \cup \mathbb{S}, \tag{2}$$

where $|\mathbb{H}| + |\mathbb{S}| + |\mathbb{E}|$ is the number of already admitted queues. If (2) is not satisfied, LQ-i is rejected. Otherwise, it 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, her own total demand should not exceed her long-term fair share. Formally, the *fairness condition* is

$$\overrightarrow{d_i}(n) * t_i(n) \leq \frac{\overrightarrow{C}\,(T_i(n+1) - T_i(n))}{|\mathbb{H}| + |\mathbb{S}| + |\mathbb{E}| + 1}, \forall n, \tag{3}$$

If only condition (2) is satisfied but (3) is not, LQ-i is added to $\mathbb{E}$. If both conditions (2) and (3) are satisfied, it is safe to admit LQ-i to $\mathbb{H}$ or $\mathbb{S}$. If there are enough uncommitted resources (*resource condition* (4)), LQ-i is admitted to $\mathbb{H}$. Otherwise it is added to $\mathbb{S}$.

$$\overrightarrow{d_i}(n) \leq \overrightarrow{C} - \sum_{j \in \mathbb{H}} \overrightarrow{d_j}(t), \forall n, t \in [T_i(n), T_i(n) + t_i(n)]. \tag{4}$$

For TQ-j, BPF simply checks the safety condition (2). 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 each first stage $[T_i(n), T_i(n) + t_i(n)]$ of her ON/OFF cycles, BPF allocates resources according to her demand $\overrightarrow{a_i}(t) = \overrightarrow{d_i}(n)$. In practice, LQ-$i$ may not fully receive the guaranteed resources. For instance, non-preemption settings do not allow the cluster to kill the running jobs or tasks to give more resources to LQ-i at the beginning of her burst. Therefore, more resources are provided to LQ-i during the second stage $[T_i(n)+t_i(n), T_i(n+1)]$ until her resource consumption reaches $\overrightarrow{d_i}(n) * t_i(n)$.

For each LQ-i in $\mathbb{S}$, the procedure is similar except that the allocate resources are based on her demand and uncommitted resource, whichever is smaller, i.e.,

$$\overrightarrow{a_i}(t) = \min\{\overrightarrow{d_i}(n), \overrightarrow{C} - \sum_{j \in \mathbb{H} \cup \mathbb{S}'} \overrightarrow{d_j}(t)\},$$

where $\mathbb{S}'$ is the set of LQs in $\mathbb{S}$ that are admitted and allocated before LQ-i. Again, resources are provided to LQ-i until her consumption reaches $\overrightarrow{d_i}(n) * t_i(n)$.

After every LQ in $\mathbb{H}$ and $\mathbb{S}$ is allocated, remaining resources are allocated to queues in $\mathbb{E}$ using DRF [43].

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.

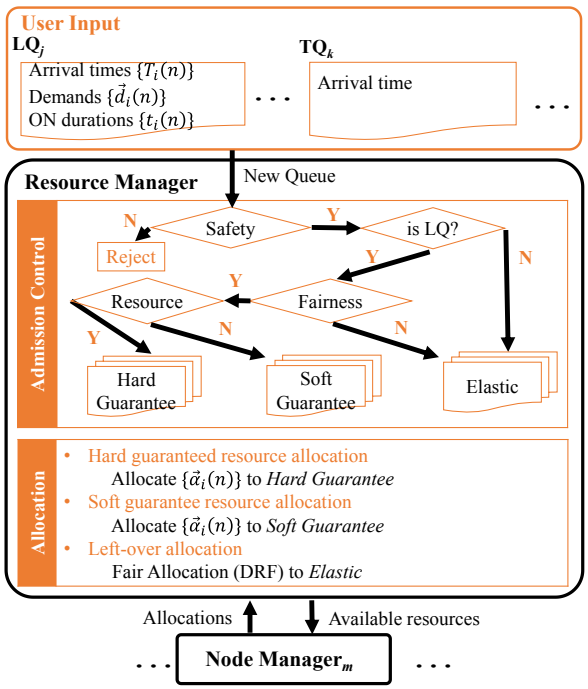


Figure 3: Enabling bounded prioritization with long-term fairness in a multi-resource cluster. BPF-related changes are shown in orange.

3.4 Properties of BPF

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 the optimal completion time. 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 near optimal and better than that under fair allocation policies.

The addition of $\mathbb{S}$ allows more LQs to be admitted with resource guarantee, and therefore increases the LQ utilization. Finally, we allocate spare resources whenever there is some, so system utilization is maximized.

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 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, to perform admission control, and to calculate resource shares jobs – 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 3 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 (2), 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 (3), it is also admitted to the Elastic class. If the new LQ meets the fairness condition (3), but fails at the resource condition (4), 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 a 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 [43].

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 [14, 39, 46] with the assumption that resource requirements for the tasks in the same stage are similar [21, 43, 63]. We do not make any new contributions on demand estimation in this paper. While BPF performs the best with accurate estimations, we have found that BPF is robust to small misestimations in practice (§5.4.1). 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 [13], 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 [3]. 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 (§5.3.1).

3. BPF handles multiple LQs to accommodate bounded priority and fairness (§5.3.2).

4. Sensitivity analysis shows that BPF is robust to estimation errors (§5.4.1), and provide benefits to LQs under the increasing impact of non-preemption (§5.4.2).

5.1 Experimental Setup

Workloads. Our workloads consist of jobs from public benchmarks – BigBench (BB) [2], TPC-DS [9], and TPC-H [10] 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 4. 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 this jobs to make sure it reaches 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.

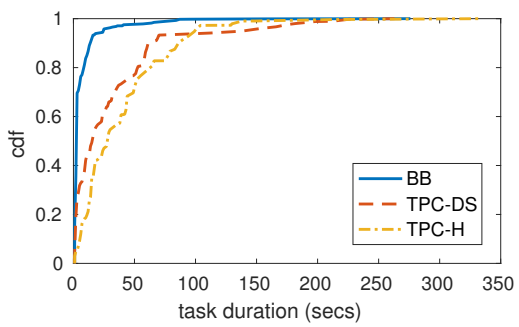


Figure 4: CDFs of task durations across workloads.

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 [1]. Unless otherwise noted, the inter-arrival period of two LQ jobs is 300 seconds (1000 seconds) for the cluster experiment (the simulation experiment).

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

Experimental Cluster. We setup Apache Hadoop 2.7.2 (YARN) on a cluster having 40 worker nodes on CloudLab [3] (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.

Compared baselines. We compare BPF against the following approaches

1. **Dominant Resource Fairness (DRF)**: DRF algorithm is implemented in YARN Fair Scheduler [13]. DRF uses the concept of the dominant resource to compare multi-dimensional resources [43]. 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 [60]. Similarly, we enable the LQ to receive all resources they need first, and then allocate the remaining resources to other queues.

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 2, N-BPF decides to admit them to Hard Guarantee if they meet the fairness condition 3 and resource condition 4. Otherwise, it put the queues into the Elastic class. We use N-BPF as a baseline when there are multiple LQs (§5.3.2).

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

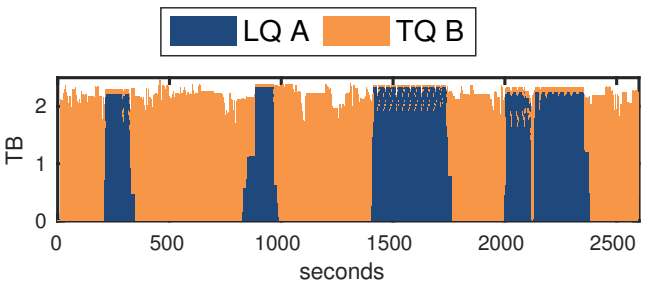


Figure 5: [Cluster] BPF's solution for the motivational problem (§2.1). The first two jobs of Bursty A quickly finish and the last two jobs are prevented from using too much resource. This solution is close to the optimal one.

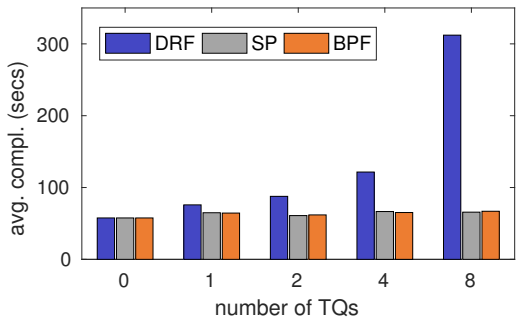


Figure 6: [Cluster] Average completion time of LQ jobs in a single LQ across the 3 schedulers when varying the number of TQs. BPF and SP guarantee the average completion time of the LQ jobs while DRF significantly suffers from the increase of number of TQs.

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 5 depicts how BPF solves it in the testbed. BPF enables the first two jobs of Bursty A 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 B.

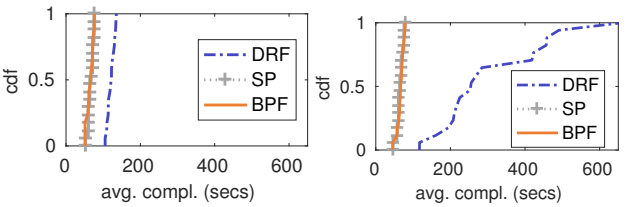
5.2.2 Performance guarantee

Next, we focus on what happens when there are more than one TQ. Figure 6 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 stage 1 duration (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.

Table 2: [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 7: [Cluster] The completion time of LQ jobs is predictable using BPF.

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 because of non-preemtion. 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 2. 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 7 shows that cumulative distribution functions (cdf) of the completion times across 3 approaches. Figure 7a and 7b 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 7b. The unstable performance is caused by the instantaneous fairness and the variance of total resource demand.

5.2.3 Fairness guarantee

Figure 8 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

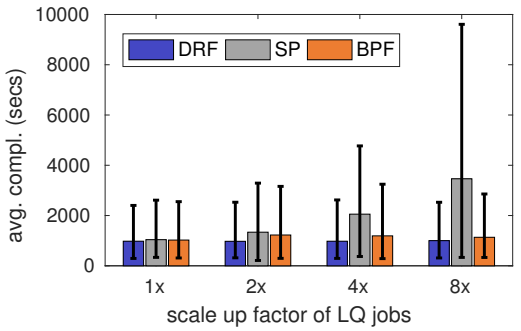


Figure 8: [Cluster] BPF protects the batch jobs up to 3.05× compared to SP.

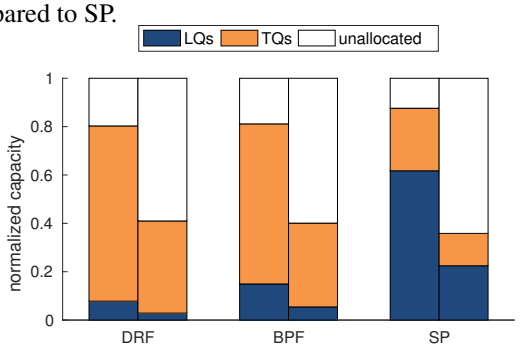


Figure 9: [Cluster] The average resource consumption of the cases of scaling up LQ jobs at 8x. The left bar is memory usage, and the right bar is CPU usage. All three policies have similar utilization.

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.

To better see the long-term fairness in resource allocation, we present average resource usage across 3 approaches in Figure 9. There are totally 9 queues (1 LQ and 8 TQ). The x-axis is the normalized capacity. SP is dominant in both memory and CPU. BPF and DRF can achieve the fairness in resource allocation for LQs and TQs. All of them have similar resource utilization.

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). Re-

Table 3: [Simulation] Factors of improvement by BPF across various workloads w.r.t the number of TQs.

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

call 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 [13]. The additional computation in allocation is also negligibly less than 1 ms.

5.3 Performance in Trace-Driven Simulations

5.3.1 Benchmarks' performance in simulations

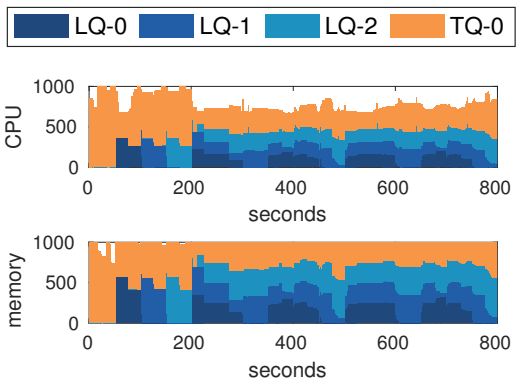
To verify the correctness of simulation, we replayed the BB trace logs from cluster experiments in the simulator. Table 3 shows the factors of improvement in completion times of LQ jobs for BB workload in simulation that are consistent with that from our cluster experiments (Table 2).

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).

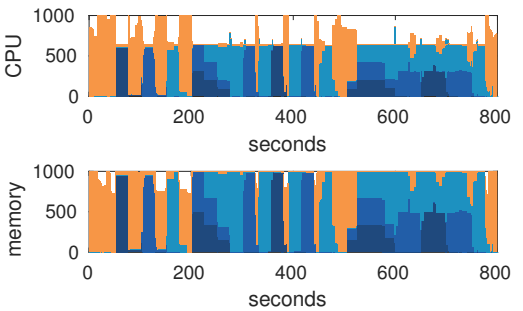
5.3.2 Admission control for Multiple LQs

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, 100, 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.

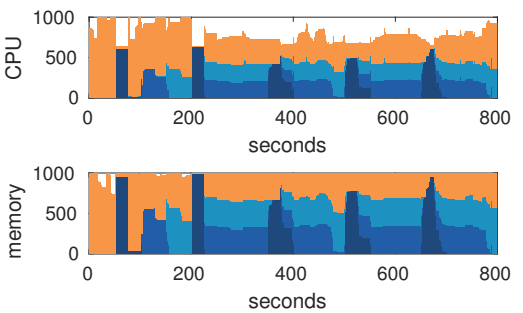
Figure 10 shows the resource usage (CPU and memory) for each queue across four schedulers, i.e., DRF, SP, N-BPF and BPF. The capacity of CPU or memory is 1000 nodes. As an instantaneously fair scheduler, DRF continuously maintains the fair share for all queues as in Figure 10a. Since LQ-2 requires a 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-BPF. Since LQ-2 demands too much resources, BPF treats



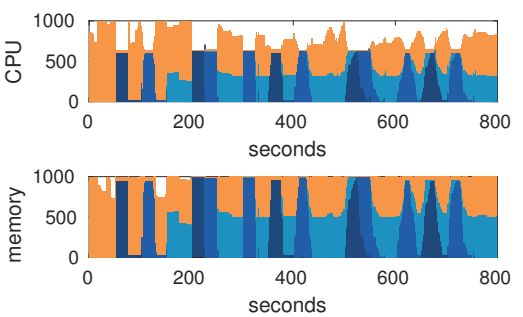
(a) DRF: LQ-0, LQ-1, LQ-2 are unhappy with high latency.



(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: [Simulation] 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 her performance cannot be guaranteed.

it like TQ-0.

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.

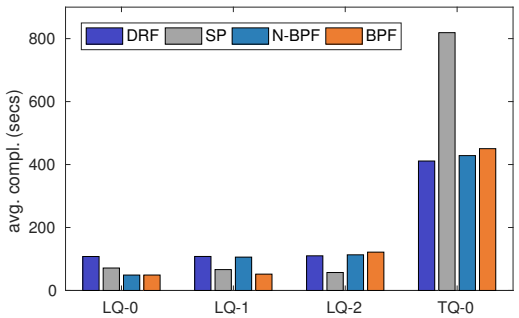


Figure 11: [Simulation] BPF provides with better performance for LQs than DRF and N-BPF. Unlike SP, BPF protects the performance of TQ jobs.

5.4 Sensitivity Analysis

5.4.1 Impact of estimation errors

BPF requires users to report their estimated demand for LQ jobs. However, it is challenging to estimate the demand accurately, which naturally results in estimation errors. To understand the impact of estimation errors on BPF, we assume that estimation errors $e(\%)$ follow the standard normal distribution with zero mean. The standard deviation (std.) of estimation errors lines in $[0, 50]$. To adopt the estimation errors, we update the task demand and durations of LQ jobs as $task_{new} = task_{orginal} * (1 + e/100)$.

Figure 12 shows the impact of estimation errors on the average completion time of LQ jobs. There are 1 single LQ and 8 TQs. LQ jobs arrive every 350 seconds. BPF is robust when the standard deviation of estimation errors vary 0 to 20. The LQ jobs in BB suffer more from the large estimation errors (std. > 30) than that of TPC-DS and TPC-H. The delays are caused by the underestimated jobs. The overestimated jobs do not suffer any delays. Although estimation errors result in performance degradation, the performance of LQ jobs is still much better than that of DRF (162 seconds).

5.4.2 Impact of non-preemption

To evaluate the impact of non-preemption, we vary the average task durations of TQ jobs. The longer task duration is, the longer the task holds its resources. In this evaluation, we set up 1 LQ and 8 TQs on the simulator. Each LQ job arrives every 350 secs. The evaluation is run on three workloads BB, TPC-DS, and TPC-H.

Figure 13 shows the impact of average task durations of TQ jobs on the average completion time of LQ jobs. When we increase the average task durations, the performance of

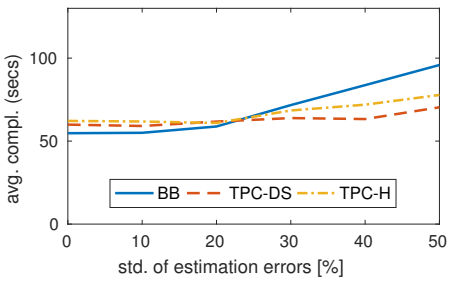


Figure 12: [Simulation] BPF's performance degrades with larger estimation errors, yet is still significantly better than DRF (162 secs).

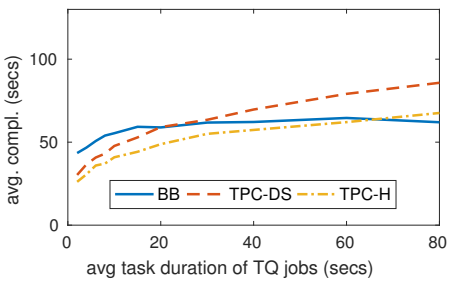


Figure 13: [Simulation] With longer average task durations, the impact of non-preemption on the performance of LQ jobs becomes larger. However, the average completion time is still significantly better than that of DRF (162 secs).

LQ jobs is degraded. Due to the variations of task durations, the BB curve stops increasing from 20, while the TPC-DS and TPC-H curves keep going up. Recall the distributions of the task durations in Figure 4: 70 percent of tasks in BB are very short. When the average task durations are more than 20 seconds, there are still a large number of short tasks in BB that allows BPF to allocate more resources to LQ jobs. TPC-DS and TPC-H have more variations of task durations that result in increasing delay for LQ jobs.

6 Related Work

Bursty Applications in Big Data Clusters Big data clusters experience burstiness from a variety of sources, including periodic jobs [15, 19, 39, 66], interactive user sessions [18], as well as streaming applications [11, 17, 73]. Some of them show predictability in terms of inter-arrival times between successive jobs (e.g., Spark Streaming [73] runs periodic mini batches in regular intervals), while some others follow different arrival processes (e.g., user interactions with large datasets [18]). 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 [22, 53, 74], modern cluster resource managers, e.g., Mesos [50], YARN [69], and others [30, 66, 70], employ multi-resource schedulers [26, 28, 41, 43, 45, 47, 59] to handle multiple resources and optimize diverse objectives. These objectives can be fairness (e.g., DRF [43]), performance (e.g., shortest-job-first (SJF) [41]), efficiency (e.g., Tetris [45]), or different combinations of the three (e.g., Carbyne [47]). 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 in Network Links Supporting multiple classes of traffic is a classic networking problem that, over the years, have arisen in local area networks [27, 40, 67, 68], wide area networks [51, 56, 61], and in datacenter networks [49, 57]. 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.

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 [68], network calculus [35, 36], datacenters [23, 57], and wide-area networks [61]. 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* [31, 43]; however, progress only applies to scenarios when a user's utility can be captured using Leontief preferences.

7 Conclusion

We observe that all existing schedulers have the same performance goal for all jobs that fails to serve both latency-sensitive LQs and throughput-sensitive TQs. In fact, the existing "fair" schedulers [26, 42, 43, 54] only make instantaneous decisions causing high latency for LQs. On the other hand, just giving high priority to LQs does not solve the problem because it may starve the TQ jobs. To enable the coexist of latency-sensitive LQs and the TQs, we proposed BPF (Bounded Priority Fairness). BPF provides bounded guarantee performance 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 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. The sensitivity analysis shows that our scheduler is robust to estimation errors and non-preemption.

8 References

[1] Apache Tez. http://tez.apache.org.

[2] Big-Data-Benchmark-for-Big-Bench. https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench.

[3] Cloudlab. http://www.cloudlab.us/.

[4] Databricks Cloud. http://databricks.com/cloud.

[5] Google Container Engine. http://kubernetes.io.

[6] Google Dataflow Cloud. https://cloud.google.com/dataflow.

[7] Presto: Distributed SQL Query Engine for Big Data. https://prestodb.io/.

[8] The Berkeley Data Analytics Stack (BDAS). https://amplab.cs.berkeley.edu/software/.

[9] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[10] TPC Benchmark H (TPC-H). http://www.tpc.org/tpch.

[11] Trident: Stateful stream processing on Storm. http://storm.apache.org/documentation/Trident-tutorial.html.

[12] YARN Capacity Scheduler. http://goo.gl/cqwcp5.

[13] YARN Fair Scheduler. http://goo.gl/w5edEQ.

[14] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 21–21. USENIX Association, 2012.

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

[16] 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.

[17] 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. *VLDB*, 2013.

[18] 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.

[19] 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.

[20] 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.

[21] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association, 2012.

[22] 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.

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

[24] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[25] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

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

[34] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

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

[36] R. Cruz. A calculus for network delay, Part II:

Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[37] R. L. Cruz. Service burstiness and dynamic burstiness measures: A framework. *Journal of High Speed Networks*, 1(2):105–127, 1992.

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

[39] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Eurosys*, pages 99–112, 2012.

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

[41] 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.

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

[43] 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.

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

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

[46] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2014.

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

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

[49] 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.

[50] 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.

[51] 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.

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

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

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

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

[56] 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.

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

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

[59] 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.

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

[61] 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.

[62] 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.

[63] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 507–518. ACM, 2010.

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

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

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

[67] 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.

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

[69] 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.

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

[71] 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.

[72] 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.

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

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