Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization

Wei Chen[1], Jia Rao[2], and Xiaobo Zhou[1]

[1]University of Colorado, Colorado Springs, {cwei, xzhou}@uccs.edu
[2]University of Texas at Arlington, jia.rao@uta.edu

Abstract

Data centers are evolving to host heterogeneous workloads on shared clusters to reduce the operational cost and achieve higher resource utilization. However, it is challenging to schedule heterogeneous workloads with diverse resource requirements and QoS constraints. On the one hand, latency-critical jobs need to be scheduled as soon as they are submitted to avoid any queuing delays. On the other hand, best-effort long jobs should be allowed to occupy the cluster when there are idle resources to improve cluster utilization. The challenge lies in how to minimize the queuing delays of short jobs while maximizing cluster utilization. Existing solutions either forcibly kill long jobs to guarantee low latency for short jobs or disable preemption to optimize utilization. Hybrid approaches with resource reservations have been proposed but need to be tuned for specific workloads.

In this paper, we propose and develop BIG-C, a container-based resource management framework for Big Data cluster computing. The key design is to leverage lightweight virtualization, a.k.a, containers to make tasks preemptable in cluster scheduling. We devise two types of preemption strategies: *immediate* and *graceful* preemptions and show their effectiveness and tradeoffs with loosely-coupled MapReduce workloads as well as iterative, in-memory Spark workloads. Based on the mechanisms for task preemption, we further develop a preemptive fair share cluster scheduler. We have implemented BIG-C in YARN. Our evaluation with synthetic and production workloads shows that low-latency and high utilization can be both attained when scheduling heterogeneous workloads on a contended cluster.

1 Introduction

Recently, the proliferation of data-intensive cluster applications, such as data mining, data analytics, scientific computation, and web search has led to the development of datacenter-scale computing. Resource efficiency is a critical issue when operating such datacenters at scale. Studies [5, 20, 21, 35] have shown that increasing utilization by sharing the hardware infrastructure among multiple users leads to superior resource and energy efficiencies. Therefore, cluster management frameworks, such as [16, 29, 31, 33] face the challenges of efficiently hosting a variety of heterogeneous workloads with diverse QoS requirements and resource demands.

Short jobs have stringent latency requirements and are sensitive to scheduling delays while long jobs can tolerate long latency but have higher requirements for the quality of scheduling, e.g., preserving data locality. To reconcile the conflicting objectives, recent proposed schedulers [4, 10] reserve a portion of the cluster to run exclusively short jobs using distributed scheduling while long jobs are scheduled onto the unreserved portion using centralized scheduling. The challenge is to determine the optimal partition of the cluster to guarantee low latency to short jobs while maintaining high cluster utilization, under highly dynamic workloads.

We look at the cluster scheduling problem from a different angle – if tasks from short jobs can preempt any long tasks, their scheduling can be made simple and fast while long jobs can run on any server in the cluster to maintain high utilization. Unfortunately, existing cluster schedulers do not support efficient task preemption. For example, YARN [31] and Mesos [16] only support kill-based preemption and killed tasks need to be restarted. This could lead to substantial slowdown to long-running jobs due to the loss of execution progress.

In this paper, we leverage application containers, a form of lightweight virtualization, to enable preemptive and low latency scheduling in clusters with heterogeneous workloads. Although containers, such as Docker, are being increasingly adopted in distributed systems [6, 33, 34], their usage is primarily for agile application deployment, leaving much of containers' potential in resource management unexploited. We explore the flexible resource management provided by container virtualization to enable low-cost task preemp-

tion. Specifically, tasks encapsulated in containers can be suspended by depriving the resources allocated to their containers and resumed later by replenishing the resources. Based on this preemption mechanism, we propose and develop BIG-C, a Container-based resource management framework for BIG data analytics that provides low-latency scheduling to preempting jobs while minimizing performance penalty to preempted jobs.

BIG-C uses Docker [22] to containerize tasks and relies on Linux `cgroups` to precisely control the CPU and memory allocations to such containers. In BIG-C, long jobs are immediately preempted upon the arrival of short jobs to guarantee low latency. We devise two types of preemptions: *immediate* and *graceful* preemptions. Immediate preemption instantaneously reduces the resources of the preempted task to a minimum footprint while still keeping the task alive to the task management. Graceful preemption gradually takes resources away from long tasks, minimizing long job slowdown. The two container-based preemption schemes replace the kill-based task preemption and can be seamlessly integrated into any fair sharing cluster schedulers and are transparent to applications. In BIG-C, we integrate the two preemption schemes into a preemptive fair share scheduler based on YARN's capacity scheduler.

We have implemented BIG-C on Apache YARN and evaluated it on a 26-node cluster using heterogeneous workloads composed of TPC-H queries and batch jobs from HiBench [17]. Experimental results show that BIG-C strikes a balance between short job latency and cluster utilization compared with state-of-the-art schedulers. The source code of BIG-C is publicly available [1].

Our results also provide insights on serving heterogeneous workloads in MapReduce and Spark. Immediate preemption works generally well for MapReduce jobs: (1) tasks are loosely coupled and the preemption of some tasks does not impede the progress of other tasks; (2) tasks have dedicated containers and their intermediate results are periodically persisted to disk, making it faster to reclaim their memory when preempted. In contrast, immediate preemption incurs significant performance loss to jobs in Spark: (1) Spark executors employ a multi-threaded model and the preemption of one container affects multiple tasks; (2) for Spark jobs with iterative computation, tasks involve in frequent synchronization between each other within and across executors/-containers; (3) due to in-memory processing, reclaiming memory from preempted containers may incur significant memory swapping. We find that graceful preemption is more suitable for Spark workloads.

[1] `https://github.com/yncxcw/big-c`

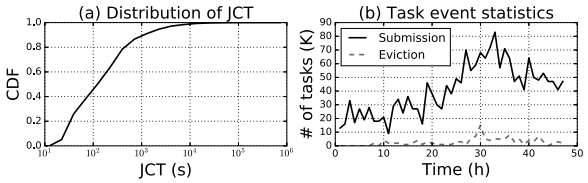

Figure 1: The analysis of Google's trace. (a) Job completion time (JCT). (b) Job submission and eviction rates.

2 Motivation

2.1 Real-world Trace Analysis

To understand job characteristics and resource usage of production workloads, we analyze the publicly available traces [28, 7] from Google data centers. Figure 1(a) shows that short jobs that complete within 1 minute dominate the traces and contribute to 80% of the total number of jobs. Although data center workloads mostly consist of short jobs, long jobs account for most resource usage. It has been reported by other researchers [10, 18, 27] that the top 10% long jobs, which are only responsible for 28% of the total number of tasks, account for more than 80% of the total task execution time. According to the analysis, there is a difficult tradeoff in data center scheduling: short jobs are sensitive to delays and critical to QoS enforcements while long jobs are important to maintaining high resource utilization. Existing research [11, 20, 21] shows that prioritizing short jobs and serving long jobs in a best-effort manner on a shared infrastructure meet both requirements.

When resources are contended, both prioritization and the enforcement of fair sharing can lead to the preemption of already running tasks. Preempted tasks are often evicted or killed. Figure 1(b) plots task submission and eviction rates in the Google trace for a period of 48 hours. According to Google, a task can be evicted by higher priority tasks in the case of resource shortage or stopped if the task owner user/group exceeds its fair share of cluster resources. Figure 1(b) shows that task eviction rate climbs up as submission rate increases, indicating a clear relationship between task preemptions and resource contention. In the 48-hour period, there were 910,099 tasks submitted, of which 93,714 were evicted and most of them were long jobs. As discussed above, long jobs account for a disproportionate amount of resource usage. The evictions of long jobs are especially expensive as the loss of their execution progress translates to significant resource waste. While evicted jobs can be relaunched when there are sufficient resources, the eviction can also add substantial delay to the completion time of such jobs.

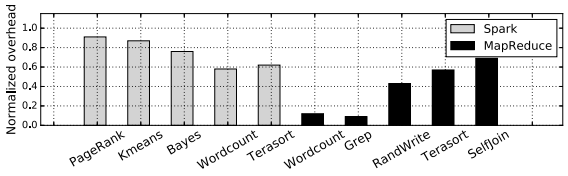

Figure 2: The overhead of kill-based task preemption in different types of workloads.

2.2 Overhead of Kill-based Preemption

Task killing is a simple means to realize preemption. However, killed tasks cannot be resumed and have to be relaunched. Most cluster schedulers use this approach due to its simplicity.

Figure 2 shows the overhead of kill-based preemption for different types of MapReduce and Spark jobs. We configured long jobs to fully utilize a 26-node YARN cluster. Detailed configuration of the cluster can be found in §5.1. During the execution of each long job, we injected a 6-minute burst of short Spark-SQL queries. The YARN default capacity scheduler was set to assign a share of 95% cluster resources to Spark-SQL queries, enforcing a strictly higher priority for the short jobs. Upon the arrival of short jobs, YARN kills tasks selected randomly from the long job to free resources needed by short jobs. Killed tasks are immediately resubmitted to YARN for rescheduling.

As shown in Figure 2, MapReduce jobs suffer less performance penalty from task killing than Spark jobs do. Task killing degraded the overall performance of MapReduce jobs by 8% - 64% while incurring as much as 92% overhead to Spark jobs. Among MapReduce jobs, those are dominated by the map phase, e.g., *wordcount*, suffered marginal degradation compared to the noticeable slowdown experienced by reduce-heavy jobs, e.g., *terasort*. Because mappers are usually small and independent from each other, the termination of a few mappers does not lead to much computation loss nor significantly delay job completion. In contrast, reducers require all-to-all communications with mappers. This data shuffling phase runs much longer than mappers. Therefore, the killing of one reducer requires the lengthy and resource intensive shuffling process to be restarted, which substantially delays job completion.

Spark jobs are more susceptible to delays due to task killings for the following reasons. First, Spark jobs, especially machine learning algorithms that iterate over a data set, require frequent synchronizations between tasks. If one task is killed, other dependent tasks are unable to make any progress. Second, Spark in-memory processing does not persist intermediate results to storage. For jobs with multiple stages, the killing of one task could lead to the re-computation of dependent stages. This recovery process is usually quite expensive. Figure 2 shows that Spark jobs suffered on average 70% slowdown when interrupted by the burst of short jobs.

3 Container-based Task Preemption

In this section, we present two simple container-based approaches for task preemption and in the next section we integrate them into cluster scheduling.

3.1 Container-based Virtualization

Container-based virtualization, such as Docker, has gained popularity due to its almost negligible overhead compared to hypervisor-based virtualization. A container provides isolated namespaces for applications running inside the container and forms a resource accounting and allocation unit. Linux uses control groups (`cgroups`) to precisely control the resource allocation to a container. Not only priorities can be set to reflect the relative importance of containers, hard resource limits guarantee that containers consume resources no more than a predefined upper bound even there are available resources in the system.

3.2 Immediate Task Preemption

We leverage the flexible resource allocation enabled by containers to temporarily suspend a task in Big Data analytics and reclaim its resources without losing the execution progress. We assume that each task is encapsulated into a container [2]. Each container forms a `cgroup` and is configured with two types of resources: CPU and memory. Parameter `cpuset.cpus` controls the number of CPU cores that a container can use and parameter `memory.limit_in_bytes` limits the maximum memory usage. `cpu.cfs_quota_us` and `cpu.cfs_period_us` together determine the maximum CPU allocation to a container. This enables fine-grained control of CPU cycles beyond allocating CPU cores.

Task suspension involves two steps: *stop task execution* and *save task context*. To stop a task, the host container is deprived of CPU to stop task execution. To save the task's context for later resumption, its dirty data in memory needs to be written back to disk. Fortunately, no additional effort is needed to support context saving. When reclaiming a container's memory, the virtual memory management in the host operating system (OS) writes back dirty data. For fault tolerance, cluster schedulers monitor the progress of individual tasks and launch speculative tasks if stragglers are detected.

The suspension of containers will falsely trigger the failover. To avoid extensive changes to cluster schedulers to support task preemption, we suspend a task but

[2] Spark runs multi-threaded task in an executor. Therefore, a container corresponds to an executor in Spark and contains multiple tasks.

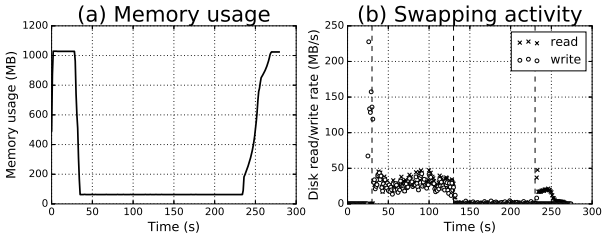

Figure 3: Immediate preemption incurs high overhead due to memory reclaiming and restoring.

maintain a minimal footprint for the task to keep it alive to the cluster scheduler. We empirically set the minimal container footprint to 1% CPU and 64 MB memory, with which the thread responsible for sending the heartbeat in the container still appears to be alive to the scheduler. We also disable speculative execution for suspended tasks.

Task resumption is simply re-activating the container by restoring its deprived resources. The resumption also follows two steps. The memory size of the container is restored from the minimal footprint back to its original size and the CPU limit is lifted. We call this type of preemption, which reclaims and restores all resources of a preempted task in one pass, *immediate preemption* (IP).

Overhead of immediate preemption Despite that kill-based preemption is crude, it guarantees timely scheduling of short jobs. The container-based immediate preemption, however, can possibly delay short job scheduling and inflict performance degradation to long jobs. First, it may take non-negligible time to reclaim the memory of preempted tasks before short tasks can be scheduled, depending on the working set size of preempted tasks. Second, task resumption requires loading saved context into memory. For certain jobs, this process is particularly long.

Figure 3 shows the memory swapping activities when a 1 GB container was suspended to the minimal footprint and later resumed. The container ran a multi-threaded synthetic Java benchmark that repeatedly and randomly touched a 1 GB array. Figure 3(a) shows that it took 3 seconds (between the 40^{th} and 45^{th} seconds) to reclaim nearly 1 GB memory. Note that the swapping activities lasted much longer until the container was deprived of CPU at the 130^{th} second. It took even longer to load saved context into memory after the container was resumed at the 230^{th} second. The reason is that the multiple threads in the container simultaneously loaded their working sets when memory was restored, resulting in a large volume of random disk access. The synthetic benchmark provides following insights on IP overhead:

- It is expensive to reclaim memory from a container that is actively dirtying its working set.
- Depriving CPU effectively throttles disk reads during memory reclaiming, shortening the suspension delay.
- Spark jobs are particularly susceptible to the resumption overhead when multiple tasks from an/a executor/container are activated to simultaneously load their working sets from disk.

To reduce the overhead, one optimization is to first reclaim CPU from a container to throttle task activity before memory is reclaimed. However, this optimization is not sufficient to guarantee short job latency or minimize long job degradation, which motivated us to develop the graceful preemption.

3.3 Graceful Task Preemption

While immediate preemption deprives a task of all resources to completely suspend the task, *graceful preemption* (GP) shrinks a preempted task and reclaims its resources in multiple rounds.

Compared to immediate preemption, graceful preemption reclaims a task's resources at a pre-defined step $\vec{r} = (c, m)$, where c and m are the unit resource reclamation for CPU cores and memory, respectively. GP is based on the following insights:

- Tasks from long jobs are usually larger than tasks from short jobs. Launching a short task often does not need to reclaim all resources of a long task.
- Resource slack is common in cluster computing. Memory slack could come from intentional overprovisioning at job launch to avoid Out-Of-Memory errors, dynamic and epochal memory demands at different job stages [24], or diminishing memory demands towards job completion. CPU slack is due to similar reasons. Even if CPU is fully utilized, gracefully reducing CPU allocation does not cause drastic performance degradation to a preempted task.
- Since tasks' requests are usually based on their peak demands, the partially reclaimed resources from preempted tasks are often sufficient for high priority tasks to make progress at an early stage of execution.

To reclaim resources from a preempted task, the cluster scheduler controls the iteration of graceful preemption. This process stops if the demands of the high priority tasks are satisfied. Similar to immediate preemption, in which we deprive the container of CPU to throttle task execution so as to accelerate memory reclamation, graceful preemption freezes a container's CPU when swapping activities are detected. Graceful preemption will continue until new tasks' demands are met.

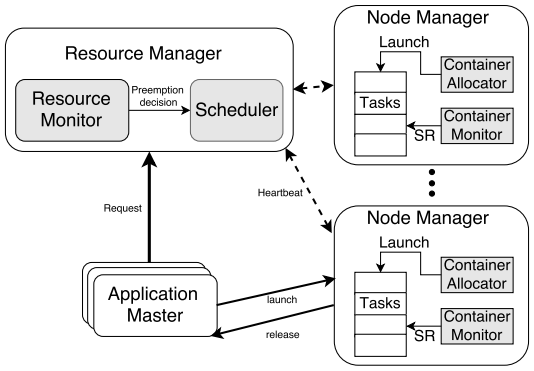

Figure 4: The architecture of BIG-C.

4 BIG-C: Preemptive Cluster Scheduling

In this section, we describe how to integrate container-based task preemption into YARN. We begin with a brief overview of YARN's resource management and task scheduling (§4.1). Next, we present the design of BIG-C and discuss the changes in YARN to support task preemption (§4.2), and present a preemptive fair share scheduler based on YARN's capacity scheduler (§4.3).

4.1 YARN Resource Management

YARN is a generic resource management framework that allows multiple applications to negotiate resources on a shared cluster. YARN uses *container*, a logical bundle of resources (e.g., ⟨1 CPU, 2GB RAM⟩) as the resource allocation unit. A container is considered as a resource lease and its resources are reclaimed as a whole when a task is completed or killed. The *resource manager* (RM), one per cluster, is responsible for allocating containers to competing applications. The *application master* (AM), one per application, submits requests for containers to RM. The *node manager* (NM), one per machine, monitors the allocation of resources on each node and updates the RM with resource availability.

4.2 BIG-C Design

Figure 4 shows the architecture of BIG-C. The key components of BIG-C include a *resource monitor* (RMon), a preemptive fair scheduler at the RM and a *container allocator* (CA), a *container monitor* (CM) at each NM.

Container allocator Although YARN also uses the notion of "container" in resource management, a YARN container is a logical representation of a task's resources but does not control the actual allocation of resources. The CA component addresses this issue. Upon receiving the request for launching a new task, CA loads the task into a Docker container. Next, CA configures the container with the resources requested by the task.

Container monitor is a per-container daemon in NM responsible for container preemption. Instructed by the NM, CM performs two actions: `container_suspend` and `container_resume` (SR operations in Figure 4). It reconfigures the preempted container to reclaim resources. If memory swapping is detected in a container, CM immediately freezes the container by setting the CPU allocation to 1%.

Resource monitor is a daemon running on the resource manager. It periodically (every 3 seconds) checks the resource distribution among queues according to the current scheduling policy, the resource availability, and resource demands of incoming tasks. Based on the resource sharing policy enforced by the scheduler, RMon together with the *scheduler* compute how much resource should be preempted from over-provisioned queues and send the preemption decision to individual NMs.

4.3 Preemptive Fair Share Scheduler

Overview The preemptive scheduler is built on YARN's capacity scheduler, which enforces fair resource allocation among users. Capacity scheduler is work-conserving and allows users, each assigned with a job queue, to use more than their fair shares if there are available resources in the cluster. When resources are contended, capacity scheduler kills tasks from over-provisioned queues to free resources for under-provisioned queues. The preemptive scheduler replaces kill-based preemption with *immediate preemption* (IP) or *graceful preemption* (GP). While IP does not require algorithmic changes to capacity scheduler, we need to augment the fair sharing algorithm to support GP.

Capacity scheduler's fair sharing algorithm enforces dominant resource fairness (DRF) [13] among job queues. Upon receiving resource requests, in the form of ⟨CPU, RAM⟩, capacity scheduler calculates the dominant resource in these requests and enforces fair allocation of the dominant resource. Non-dominant resources are allocated in proportion to the dominant resource as specified in the requests. Algorithm 1 shows how capacity scheduler calculates the amount of resources to be reclaimed from over-provisioned queues and our modification (highlighted in red) to support GP. For ease of discussion, the algorithm assumes two queues, i.e., one for long jobs and one for short jobs, and two type of resources, i.e., CPU and memory. It can be extended to support more than two queues and two resources.

Capacity scheduler first determines the desired share of resource (line 3). The over-provisioned resources for the long job queue is the difference between the queue's current resource allocation $\vec{r}_l$ and its desired share $\vec{f}_l$ (line 4). If the demand of the short jobs is less than the long job's over-occupied resources, the demand can be fully satisfied (line 5-6). Otherwise, all over-provisioned resources should be reclaimed (line 9). The amount of preempted resources $\vec{p}$ is used in Algorithm 2 to determine

Algorithm 1 Calculate resources to be preempted.

1: **Variables:** Long job's fair share ϕ_l, current resource allocation $\vec{r}_l$, fair allocation $\vec{f}_l$; total CPU C and memory M resources; short job's resource request $\vec{r}_s$; over-provisioned resources $\vec{a}$; resources of long job to be preempted $\vec{p}$.
2: /* Long job's fair and over-provisioned resources */
3: $\vec{f}_l = (C \times \phi_l, M \times \phi_l)$
4: $\vec{a} = \vec{r}_l - \vec{f}_l$
5: **if** $\vec{r}_s < \vec{a}$ **then**
6: $\quad \vec{p} = \vec{r}_s$
7: **else**
8: $\quad$ /* Use DR to calculate preempted resources*/
9: $\quad \vec{p} = \vec{a} \Longrightarrow \vec{p} =$ **ComputeDr**$(\vec{r}_s, \vec{a})$
10: **end if**
11: **procedure** ComputeDr$(\vec{r}_s, \vec{a})$
12: $\quad$ Determine dominant resource
13: $\quad$ **if** dominant resource is CPU **then**
14: $\quad\quad \vec{p} = (a_{cpu}, a_{mem} \times \frac{r_{s\_mem}}{r_{s\_cpu}})$
15: $\quad$ **else**
16: $\quad\quad \vec{p} = (a_{cpu} \times \frac{r_{s\_cpu}}{r_{s\_mem}}, a_{mem})$
17: $\quad$ **end if**
18: $\quad$ return $\vec{p}$
19: **end procedure**

Algorithm 2 Container preemption.

1: **Variables:** Set of container to be preempted $\mathbb{C}$; resources to be preempted $\vec{p}$; preempted resources at each GP step $\vec{r}_{GP}$; resources of a preempted container $\vec{r}_c$.
2: AGAIN:
3: Choose a job with the longest remaining time
4: **while** $\vec{p} > (0,0)$ **do**
5: $\quad$ Choose a container c from the job
6: $\quad \mathbb{C} \leftarrow c$
7: $\quad \vec{p} = \vec{p} - \vec{r}_c \Longrightarrow \vec{p} = \vec{p} - \vec{r}_{GP}, \vec{r}_c = \vec{r}_c - \vec{r}_{GP}$
8: $\quad$ remove $c \Longrightarrow$ **if c is empty or swapping, remove c**
9: $\quad$ **if** job has no container left and $\vec{p} > (0,0)$ **then**
10: $\quad\quad$ goto AGAIN
11: $\quad$ **end if**
12: **end while**
13: Kill$(\mathbb{C}) \Longrightarrow$ **Preempt**$(\mathbb{C})$

which containers that belong to the long job should be killed to release these resources.

As shown in Algorithm 2, capacity scheduler uses the heuristic proposed in [37] to choose a job with the longest remaining time and releases $\vec{p}$ resources from its containers. Note that $\vec{p}$ is calculated in algorithm 1. Each time such a container is found, it is added to the kill set $\mathbb{C}$ until either the job has no container left or $\vec{p}$ has been satisfied (line 3-9). If more resources need to be reclaimed, capacity scheduler moves to the next job (line 10). Note that as long as the over-provisioned dominant resource is fully reclaimed, $\vec{p}$ is considered satisfied. The to-be-killed container set $\mathbb{C}$ is then sent to NMs to perform the killings (line 13). There are two drawbacks of capacity scheduler due to kill-based preemption. First, kill-based preemption may lead to resource fragmentation. A killed long job container may be too large for one short task but not sufficient for two short tasks. Second, task killing is not a flexible way to reclaim resource. The killing of a large container only frees resources on one machine and may lead to the launch of a large number of small tasks all clustered on the machine, causing not only load balancing but also reliability problems.

We make simple changes to capacity scheduler to address the above drawbacks. To avoid resource fragmentation, the preempted resource is accurately calculated by function **ComputeDr** based on the demand of short jobs and over-provisioned resources (Algorithm 1, line 9). Specifically, the scheduler computes the dominant resource in request $\vec{r}_s$ against the over-provisioned resource $\vec{a}$. Instead of reclaiming all over-provisioned non-dominant resource as capacity scheduler does, it reclaims the non-dominant resource in proportion to the reclaimed dominant resource as indicated in $\vec{r}_s$. For instance, suppose $\vec{a} = \langle 10 \text{ CPU}, 15\text{GB RAM} \rangle$ and $\vec{r}_s = \langle 20 \text{ CPU}, 10\text{GB RAM} \rangle$. Since $\frac{20}{10} > \frac{10}{15}$, CPU is the dominant resource. Because $\vec{r}_s > \vec{a}$, capacity scheduler will compute $\vec{p} = \vec{a}$ and over-reclaim the memory resource. Instead, the preemptive scheduler computes $\vec{p} = \langle 10 \text{ CPU}, 10 \times \frac{10}{20} \text{ GB RAM} \rangle$. Note that resource preemption based on the dominant resource of short job requests is not possible in the original capacity scheduler because resource allocation of long jobs' containers is based on long jobs' dominant resource and should be reclaimed as a whole.

Further, we introduce graceful task preemption in Algorithm 2. When a job is selected, its over-provisioned resources are reclaimed from a large number of its tasks at a step of $\vec{r}_{GP}$. For each round, the remaining resources of a container c are also updated (line 7). Once a container starts swapping, showing a memory shortage, it is immediately frozen and removed from resource reclamation (line 8). As such, preempted resources are collected from many containers, avoiding drastic slowdown to individual containers as much as possible. As discussed in §2.2, evenly preempting resources from tightly-coupled tasks help minimize slowdowns of Spark jobs.

Practical considerations

Tuning $\vec{r}_{GP}$. The step at which resources are preempted in graceful preemption presents a tradeoff. The larger the $\vec{r}_{GP}$, (hopefully) the sooner the short jobs' demand can be satisfied, but at the risk of causing more pronounced slowdown to long jobs. If $\vec{r}_{GP}$ is too large and pre-

empted containers incur swapping, short jobs can even wait longer for resources to be freed from swapping containers. Small $\vec{r}_{GP}$ leads slow resource allocation to short jobs, which may suffer poor performance after launch.

Delayed resumption factor D. Killed and preempted tasks resubmit their resource requests to the RM and are treated like ordinary incoming tasks. However, resource request from a preempted task has a special locality requirement - it can only be satisfied on the machine where the task was preempted. Moreover, under high burst of short job arrival, a resumed container can be quickly preempted again. The repeated and wasteful preemptions hurt long job performance but also cause long queuing delay to short jobs. To address this issue, we require that a preempted container needs to try D times before it is really resumed. This also avoids possible starvation.

4.4 Implementation

We have implemented BIG-C in Hadoop YARN. The resource monitor is a new module residing in the resource manager that extends `SchedulingEditPolicy`. Our new preemptive fair share scheduler is based on YARN's capacity scheduler. The modifications includes adding a new task state `PREEMPTED`, interfaces for task suspension and resumption in the resource manager. These changes are generic and can interface with any cluster schedulers. On node manager, the container monitor extends `ContainerManagerImpl`. We build a new module called `CoresManager` to handle CPU allocation at worker nodes. A Java interface for `libcontianer` is added to each node manager to operate Docker containers. Our implementation includes 2000 lines of Java code and is based on Hadoop-2.7.1.

5 Evaluation

This section presents the performance evaluation of BIG-C on a 26-node cluster using representative MapReduce and Spark workloads. We first provide details of our testbed (§5.1). Next, we present results from synthetic workloads with MapReduce and Spark jobs (§5.2), and study the sensitivity of two tunable parameters in our design (§5.3). Last, we give results from production workloads based on the Google trace (§5.5).

5.1 Experimental Setup

Cluster Setup Each machine in the 26-node cluster has two 8-core Intel Xeon E5-2640 processors with hyper-threading enabled, 132GB of RAM, and 5x1-TB hard drivers configured as RAID-5. The machines were interconnected by 10 Gbps Ethernet. Hadoop-2.7.1. was deployed on the cluster and HDFS was configured with a replication factor of 3 and a block size of 128MB. The worker nodes sent heartbeats to the resource manager every 3 seconds. Docker-1.12.1 was used to create containers and the images were downloaded from online repository `sequenceiq/hadoop-docker`.

We configured two queues in YARN's resource manager to serve heterogeneous workloads. One queue was dedicated to short jobs and the other was for long jobs. Such a two-queue setting is commonly used in production systems and has been adopted by other works [8, 9, 10]. Additionally, BIG-C can leverage approaches presented in [10, 12] to classify short and long jobs. To enforce strictly higher priority for short jobs, we set the resource share of the short job queue to 95%. The remaining 5% was assigned to the long job queue in a best-effort manner. For comparison, we evaluated the following cluster schedulers:

- **FIFO** scheduler serves all tasks in a single first-in-first-out queue. It achieves optimal performance for long jobs, but incurs significant performance penalty for short jobs.
- **Reserve** schedulers such as Hawk [10] reserve a portion of the cluster to run short jobs exclusively without preemption. Our experiments empirically reserve 60% of cluster resources for short jobs. Long jobs can use up to 40% of cluster capacity. However, it is challenging to find the optimal reservation factor under highly dynamic workloads.
- **Kill** is the preemption mechanism in YARN. The capacity scheduler is used to enforce share between queues. It achieves optimal performance for short jobs, but causes performance degradation to long jobs.
- **IP** and **GP** are immediate preemption and graceful preemptions, respectively. Our preemptive fair share scheduler is used with these two approaches.

Workloads We used Spark-SQL [2] to generate TPC-H queries as short jobs. Hive [30] was used to populate TPC-H tables in HDFS. The total data size was 10GB. The container size for Spark-SQL tasks were set to ⟨4 CPU, 4GB⟩. We selected long jobs from HiBench benchmarks. For MapReduce jobs, we chose map-heavy *wordcount* and reduce-heavy *terasort*. The map and reduce containers were set to ⟨1 CPU, 2GB⟩ and ⟨1 CPU, 4GB⟩, respectively. The input size of the MapReduce jobs was 600GB. We selected *PageRank*, *Kmeans*, *Bayes* and *WordCount* from HiBench as the Spark jobs. The containers of Spark executors were much larger with configurations of ⟨8 CPU, 16GB⟩, ⟨8 CPU, 32GB⟩, and ⟨16 CPU, 32GB⟩, depending on the input size. [3]

Metrics We evaluated the cluster schedulers using the following metrics: *job completion time* (JCT) is the time

[3]The number of CPUs specifies the number of parallel tasks Spark will launch in each executor. Memory size should be large enough to prevent tasks from running into the *Out-Of-Memory* error.

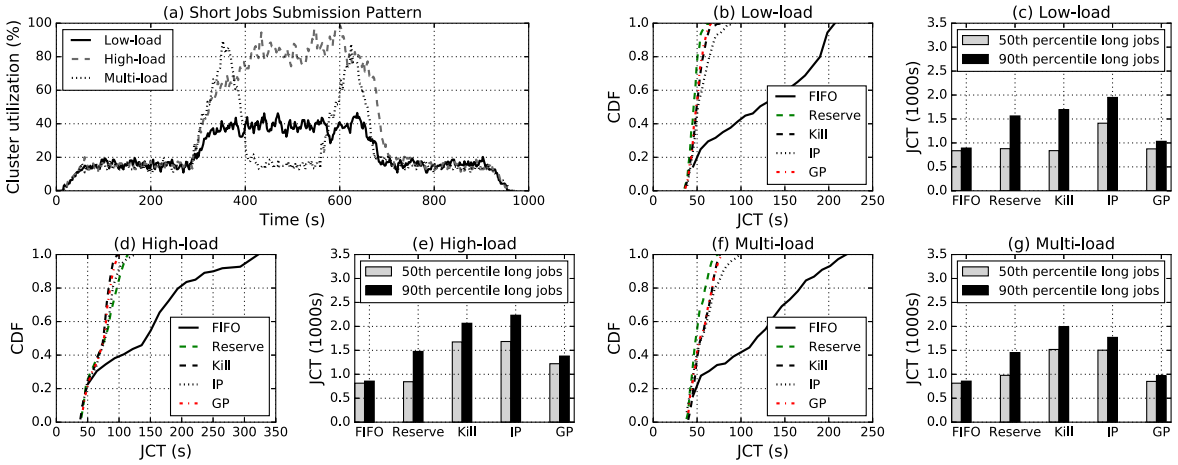

Figure 5: (a) The submission patterns of short jobs in the synthetic workloads. The performance of short Spark-SQL queries (b,d,f) and long Spark jobs (c,e,g) with different schedulers.

when job is submitted until it is completed; *job queueing delay* is the time when a job is submitted until its execution starts; *CoV* of JCT is JCT's coefficient of variation; *cluster utilization* is the CPU utilization over the total CPU capacity of the cluster.

5.2 Results on Synthetic Workloads

Setting In this experiment, we created a controlled environment to study the performance of BIG-C. We generated three workload patterns, each with mixed long and short jobs and lasting for 900 seconds. Long jobs were continuously submitted throughout the experiment and persistently utilized about 80% of the cluster resources. Figure 5(a) shows the submission pattern of short jobs. While all three patterns had a base demand of around 20% cluster capacity, they differed in the submission bursts. *High-load* and *low-load* generated approximately 90% and 40% cluster utilizations, respectively, during the burst period between the 300^{th} and 700^{th} seconds. In contrast, *multi-load* had two peaks during the burst period with each peak demanding over 80% cluster resources. Clearly, the cluster was overloaded during the bursts and long jobs should be preempted by short jobs.

Spark performance Figure 5(b)-(g) shows the results. Among schedulers, *reserve* achieved the best performance for short jobs under low-load and multi-load, since the reserved 60% cluster resources were sufficient to serve the burst. Under high-load, *Reserve* had degraded performance as the resource reservation for short jobs was less than the peak demand. *Kill* was among the best performing schedulers for all three scenarios. In contrast, *FIFO* inflicted substantially delays to short jobs due to the absence of preemption. Short jobs needed to wait for the completion of long jobs before they can be scheduled. Our approaches *IP* and *GP* with the preemptive fair scheduler achieved close performance to the best performing schedulers. Overall, *GP* had superior performance than *IP* as it required less resource reclamation time before short jobs can be launched.

We draw the 50^{th} percentile and 90^{th} percentile performance for long Spark jobs to show the median and the long tail JCTs. *FIFO* achieved the best performance for long jobs because short jobs were unable to preempt and interrupt long job execution. In contrast, *kill* based preemption incurred significant delays to long jobs, especially for the 90^{th} percentile JCT, in all scenarios. It caused on average 140% degradation compared to *FIFO* for the tail JCT. Note that kill-based preemption did not affect the median performance much under low-load since only a few long jobs were killed.

In comparison, *IP* had the worst median and tail performance among all schedulers under low-load. Its aggressive resource preemption and the resulted memory swapping even affected the median JCT under low-load, in which other non-preempted jobs suffered degradation due to memory thrashing. Another reason for *IP*'s poor performance was because complete suspension of one task in Spark jobs often stops the entire job due to task synchronization. Note that even *reserve*, which only dedicated 40% cluster capacity to long jobs, achieved better performance than *kill* and *IP*, indicating that long queuing delay was less damaging on performance than aggressive killings and preemptions. Finally, *GP* was the best performing scheduler compared to baseline *FIFO*. It incurred 13%, 61%, and 17% penalty to the tail performance compared to *FIFO* under low-load, high-load, and multi-load, respectively. The slightly worse median performance of *GP* compared to *kill* was the evidence that many tasks were slowed down due to the collection of preempted resources on these tasks.

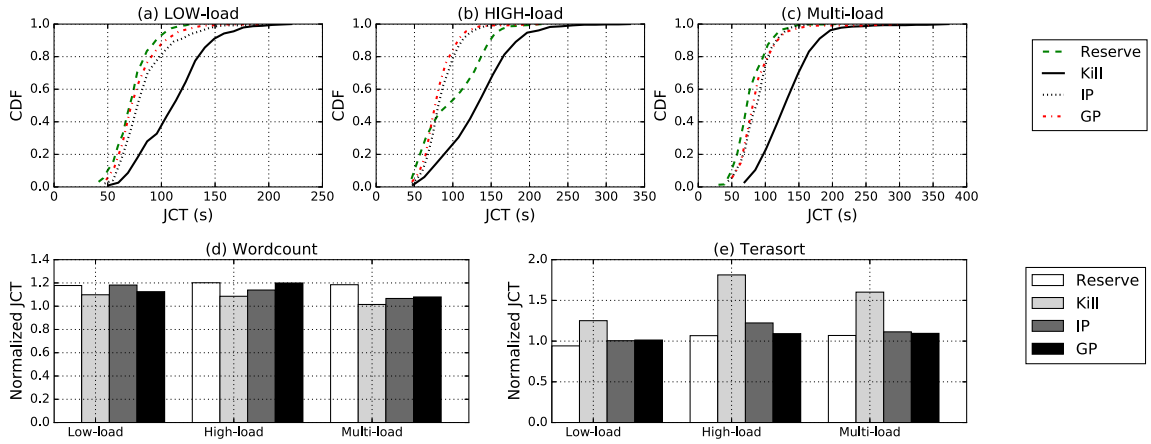

Figure 6: The performance of short Spark-SQL queries (a,b,c) and two long MapReduce jobs (d,e).

Another important finding is that kill-based preemption caused 13%, 15%, and 20% of long jobs failed to complete [4]. In contrast, although *IP* incurred significant overhead, it caused no jobs to fail.

MapReduce performance Figure 6 shows the results of MapReduce workloads. The long job performance is normalized to the scenario in which the cluster is dedicated to long jobs. The short jobs were the same Spark-SQL queries while the long jobs were map-heavy *wordcount* and reduce-heavy *terasort*. MapReduce jobs differ from Spark jobs in many ways. First, a long job usually contains a large number of small mappers, which complete quickly. Second, while Spark's in-memory computing imposes persistent resource demand throughout job execution, MapReduce jobs show clear decline in demand when entering the reduce phase. Finally, MapReduce tasks persist intermediate data onto disk whenever their memory buffers are full. These differences led to different findings in MapReduce workloads.

Figure 6 does not show the results of *FIFO* because the background long jobs had a large number of mappers backlogged and most short jobs suffered 15-20 minutes slowdown. In Figure 6(a)-(c), it is unexpected that *kill* incurred significant queuing delay to short jobs while both *IP* and *GP* performed much better. An examination of YARN's job submission log revealed that the large numbers of killed MapReduce tasks were immediately resubmitted to the scheduler and later killed again, causing wasted cluster resources and additional queuing delays to short jobs. In contrast, both *IP* and *GP* is configured with delayed resumption, which avoided repeated preemptions. *Reserve* had superior performance among the schedulers except for the scenario under high-load, in which the reservation was not sufficient. Both *IP* and *GP* performed well for short jobs.

[4]Failed jobs were not included in JCT calculation.

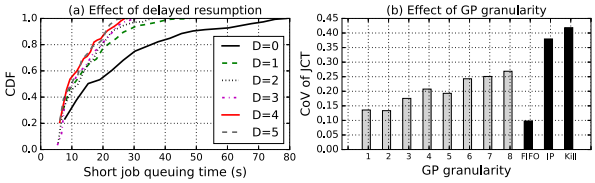

Figure 7: Effects of (a) various degrees of resumption delay and (b) the granularity of graceful preemption.

For long MapReduce jobs, the performance of schedulers depends on the type of the long job workloads. As shown in Figure 6(d), *kill*, *IP*, and *GP* had similar performance for map-heavy workload *wordcount*. It suggests that kill-based preemption is not particularly more expensive than container-based preemption as the mappers are usually small. Because there are a large number of mappers, which are independent from each other, the lost work due to the killings of small mappers can be overlapped with other mappers backlogged in the scheduler. In contrast, kill-based preemption incurred substantial overhead to reduce-heavy *terasort* workload. The cost of killing a reduce task is prohibitively high as relaunching the killed reducer requires re-shuffling all its input data over the network.

Both *IP* and *GP* achieved near-optimal performance with *IP* incurring slight degradation under high-load and multi-load. The write-back of dirty data effectively reduces the in-memory footprint of preempted tasks, making it easier for *IP* and *GP* to reclaim memory compared to that in Spark jobs.

5.3 Parameter Sensitivity

As discussed above, the delayed resumption in our approaches effectively avoided repeated preemptions. In this section, we evaluate the effects of two configurable

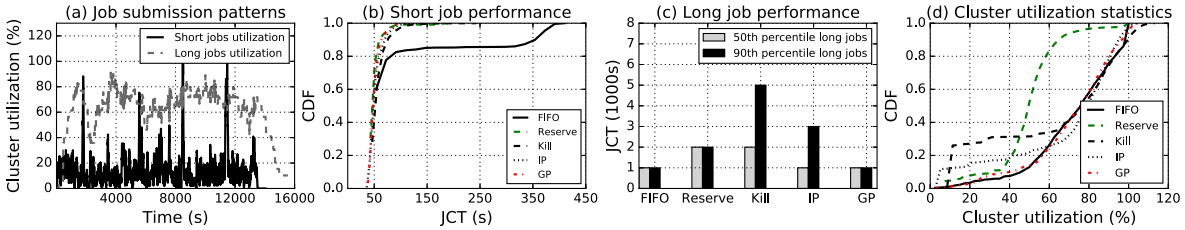

Figure 8: Results on the Google trace. (a) Jobs submission patterns. (b) The CDF of short job's JCT. (c) Long job's 50^{th} and 90^{th} percentile completion time. (d) Statistics on cluster utilization, sampled every 2 seconds.

parameters in our approach. The first parameter is the number of tries a preempted container needs to perform before it is actually resumed. Figure 7(a) shows the effect of varying numbers of the delayed try D on short job performance. The figure suggests that delayed resumption is critical to guaranteeing low latency for short jobs. Disabling delayed resumption ($D = 0$) led to queuing delay as high as 80 seconds for short jobs. Enabling delayed preemption had salient impact on performance but with diminishing gain when further increasing D. We empirically set resumption factor D to 3 to strike a balance between short job latency and long job starvation. This setting was used in all other experiments.

We have shown that there exist tradeoffs between aggressive and graceful resource preemption. Next, we quantitatively study how the granularity (aggressiveness) of GP affects long job performance. We use the coefficient of variance (CoV) of JCT to measure the distribution of preempted resources. The basic preemption unit was set to ⟨1 CPU, 2GB RAM⟩. The GP granularity is determined by how many basic resource units should be reclaimed in one round. Figure 7(b) shows that the CoV of job completion time increased as we increased the granularity of GP. Compared to kill-based and immediate preemptions, graceful preemption, even with aggressive resource reclamation, still incurred less variation across jobs. We set the preemption granularity to ⟨2 CPU, 4GB RAM⟩, i.e., two basic units.

5.4 Overhead

The overhead of BIG-C mainly comes from reclaiming the memory of preempted tasks and the delay caused by memory restoration. Our experiments show that it takes approximately 3 seconds to reclaim 1GB dirty memory, which adds considerable scheduling delay to short jobs. Although BIG-C avoids such overhead for most of time due to graceful preemption, performance degradation is inevitable if GP fails to satisfy short job demands.

5.5 Results on the Google Trace

We also evaluated BIG-C by replaying the production Google trace on our testbed. This trace contains 2202 jobs, of which 2020 are classified as short jobs and 182 as long jobs based on job completion time and resource usage. The setting conforms with that used in [10]. We scaled down the task numbers in each job to match our cluster capacity so that each job takes a reasonable amount of time to complete. The total trace ran for 3.8 hours. We first dedicated the entire cluster to short jobs and long jobs to respectively quantify their resource usage. The results are shown in Figure 8(a). The average cluster utilization was about 17% and 75% for short and long jobs, respectively. The short jobs only consumed a small portion of the total resource, but with highly variable and unpredictable submission rates. We can clearly see a few short job usage spikes throughout the trace. For example, the spike at the 8000^{th} second used up to 95% of the cluster capacity. Similarly, we configured the short job share to be 95% of the cluster capacity.

Figure 8(b) plots the latency distribution of short job in the Google trace. Most schedulers except *FIFO* achieved good performance. *FIFO* had a 90^{th} percentile latency of 335s, which was 6 times larger than its median latency. We also examined the tail latency under other schedulers. The 95^{th} percentile latency for *reserve*, *kill*, *IP*, and *GP* were 183s, 176s, 118s, and 96s, respectively. Our two approaches outperformed other schedulers with *GP* being the best.

Figure 8(c) draws the 50^{th} percentile and 90^{th} percentile performance for long jobs. With the default kill-based preemption, 105 out of the 182 long jobs were killed, among which 41 failed and 105 suffered significant slowdown due to job re-launch. Note that failed jobs were excluded from JCT calculation. As shown in the figure, *GP* improved the 90^{th} percentile job runtime by 67%, 37% and 32% over *kill*, *IP*, and *reserve*, respectively. Compared to the optimal *FIFO* scheduler for long jobs, *GP* only added 4% delay to JCT. Similarly, about 23% long jobs failed with kill-based preemption while our approaches did not cause any job failures.

Figure 8(d) plots the cluster utilization under different schedulers. Work-conserving schedulers achieved much higher resource utilizations than *reserve* did. For more than 60% of time in the experiment, the overall cluster

utilization was above 80% for *FIFO*, *kill*, *IP*, and *GP*. In contrast, *reserve* rarely used more than 60% of cluster capacity due to the reservation for short jobs. Note that both *kill* and *IP* had periods when the cluster utilization was lower than 40%. This was due to the killing and aggressive preemption of tasks that impeded the overall progress of the tightly-coupled long jobs, e.g., Spark jobs. When waiting for the killed or preempted task to relaunch or resume, other sibling tasks were idling.

6 Related Work

The last few years have witnessed the growth of workloads provisioned on top of data processing frameworks like Hadoop [1], Naiad [23] or Spark [36]. The characteristics of such workloads have been well-studied in previous work [32].

Cluster Scheduling is a core component in data-intensive cluster computing. YARN [31] and Mesos [16] are two widely used open-source cluster managers. Both YARN and Mesos use a two-level architecture, decoupling allocation from application-specific logic such as task scheduling, speculative execution or failure handling. Omega [29] is a parallel scheduler architecture based on lock-free optimistic concurrency control to achieve implementation extensibility, globally optimized scheduling, and scalability. Another thread of work focuses on distributed scheduler to overcome the scalability problem in large-scale clusters. Sparrow [25] is a fully distributed scheduler that performs scheduling by performing randomized sampling. Hawk [10] and Mercury [18] both implement a hybrid scheduler to avoid inferior scheduling decisions for a subset of jobs as a trade-off of scheduling quality and scalability. yaq-d and yaq-c [27] provides queue management at worker nodes to improve cluster utilization and to avoid head-of-line blocking. Our proposed container-based preemption is orthogonal to these approaches and helps simplify the design of cluster schedulers by providing an alternative means of enforcing task priority. Note that our work does not intend to improve task classification but focuses on a lightweight mechanism for task preemption.

Preemption Amoeba [3] and Natjam [8] implement preemption using checkpointing to achieve elastic resource allocation. Natjam targets at Hadoop applications and Amoeba built a prototype based on Sailfish [26]. Li et. al., propose a new checkpoint mechanism by leveraging CRIU [19]. Their approaches interact with the Application Master and dump the checkpoints to user space. There are two drawbacks in checkpoint-based preemption. First, it is challenging to determine the frequency of checkpointing. On-demand checkpointing, such as the preemption approaches based on CRIU [19], requires the entire preempted task to be dumped onto HDFS and is equivalent to our proposed immediate preemption. Periodic checkpointing at each iteration reduces preemption delay but incurs considerable runtime overhead. Second, existing checkpointing approaches require changes to user applications. Our proposed container-based preemption is application agnostic and the tuning of the GP granularity is relatively straightforward.

Utilization To improve cluster utilization, authors in [21, 35, 20, 38, 14] propose to consolidate applications on a shared infrastructure and separately manage their interference so that applications' QoS could be guaranteed. These techniques employ online profiling to identify the best combinations of workloads that do not interfere with each other. However, in data center scheduling, in which job submissions are unpredictable and the composition of jobs is heterogeneous, offline training or online profiling may not be accurate. Our approach does not require the cluster to be under-provisioned nor assumes scheduling-friendly job submissions.

Lightweight virtualization Container-based virtualization have been widely used both in industry and in research. Xavier et. al., [34] evaluated the HPC performance in container based environments. Burns et al., [6] propose a new design pattern for container based distributed systems. Google Borg [33] has used OS container to aid cluster management. However, its container usage is limited to task isolation and preemption is still based on task killing. Harter et al., [15] propose a Docker storage driver to enable fast container startup. The YARN community has also provided support to run Docker containers to replace the logical YARN container. However, there still lacks support to fully control the resource allocation to containers in YARN.

7 Conclusion

In this paper, we tackle the problem of scheduling heterogeneous workloads on a shared cluster. Inspired by task scheduling in operating systems, in which fast and low-cost preemption is key to achieving both responsiveness and high utilization, we leverage lightweight virtualization to enable task preemption in cluster computing, such as YARN. Experimental results show that our proposed mechanism for preemption is effective for different types of Big Data workloads, e.g., MapReduce and Spark. Note that container-based preemption is not yet suitable for workloads with sub-second latency, like those studied in [25]. Suspending and saving the context of a data-intensive task still takes a few seconds. Providing extremely low-latency task preemption for sub-second workloads is an interesting future direction.

Acknowledgement We are grateful to our reviewers for their comments on this paper and our shepherd Mona Attariyan for her suggestions. This research was supported in part by U.S. NSF grants CNS-1422119, CNS-1649502 and IIS-1633753.

References

[1] Apache hadoop project. `https://hadoop.apache.org/`.

[2] Spark-sql. `http://spark.apache.org/sql/`.

[3] ANANTHANARAYANAN, G., DOUGLAS, C., RAMAKRISHNAN, R., RAO, S., AND STOICA, I. True elasticity in multi-tenant data-intensive compute clusters. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012).

[4] ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of ACM SIGMETRICS Performance Evaluation Review* (2000).

[5] BARROSO, L. A., AND HOELZLE, U. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[6] BURNS, B., AND OPPENHEIMER, D. Design patterns for container-based distributed systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[7] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *In Proceedings of the VLDB Endowment* (2012).

[8] CHO, B., RAHMAN, M., CHAJED, T., GUPTA, I., ABAD, C., ROBERTS, N., AND LIN, P. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013).

[9] CURINO, C., DIFALLAH, D. E., DOUGLAS, C., KRISHNAN, S., RAMAKRISHNAN, R., AND RAO, S. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing* (2014).

[10] DELGADO, P., DINU, F., KERMARREC, A.-M., AND ZWAENEPOEL, W. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015).

[11] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[12] FERGUSON, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012).

[13] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2011).

[14] GRANDL, R., CHOWDHURY, M., AKELLA, A., AND ANANTHANARAYANAN, G. Altruistic scheduling in multi-resource clusters. In *Proceedings of OSDI16: 12th USENIX Symposium on Operating Systems Design and Implementation* (2016).

[15] HARTER, T., SALMON, B., LIU, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Slacker: fast distribution with lazy docker containers. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016).

[16] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R. H., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2011).

[17] HUANG, S., HUANG, J., DAI, J., XIE, T., AND HUANG, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proceedings of the Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on* (2010).

[18] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015).

[19] LI, J., PU, C., CHEN, Y., TALWAR, V., AND MILOJICIC, D. Improving preemptive scheduling with application-transparent checkpointing in shared clusters. In *Proceedings of the 16th Annual Middleware Conference* (2015).

[20] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: improving resource efficiency at scale. In *Processings of the ACM SIGARCH Computer Architecture News* (2015).

[21] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture* (2011).

[22] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Proceedings of the Linux Journal* (2014).

[23] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013).

[24] NGUYEN, K., FANG, L., XU, G., DEMSKY, B., LU, S., ALAMIAN, S., AND MUTLU, O. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[25] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013).

[26] RAO, S., RAMAKRISHNAN, R., SILBERSTEIN, A., OVSIANNIKOV, M., AND REEVES, D. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012).

[27] RASLEY, J., KARANASOS, K., KANDULA, S., FONSECA, R., VOJNOVIC, M., AND RAO, S. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016).

[28] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Proceedings of the Intel Science and Technology Center for Cloud Computing, Tech. Rep* (2012).

[29] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013).

[30] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* (2009).

[31] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache hadoop yarn: Yet another

resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013).

[32] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016).

[33] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015).

[34] XAVIER, M. G., NEVES, M. V., ROSSI, F. D., FERRETO, T. C., LANGE, T., AND DE ROSE, C. A. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2013).

[35] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the ACM SIGARCH Computer Architecture News* (2013).

[36] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. *Proceedings of HOTCLOUD'16 USENIX Workshop on Hot Topics in Cloud Computing* (2010).

[37] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R. H., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *OSDI* (2008).

[38] ZHANG, Y., PREKAS, G., FUMAROLA, G. M., FONTOURA, M., GOIRI, Í., AND BIANCHINI, R. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).