This is the accepted version of the paper published in CCGrid 2020.
The published version is available at http://dx.doi.org/10.1109/CCGrid49817.2020.00-62.

Multi-resource Low-latency Cluster Scheduling without Execution Time Estimation

Hidehito Yabuuchi
The University of Tokyo
yabuuchi@os.ecc.u-tokyo.ac.jp

Takahiro Shinagawa
The University of Tokyo
shina@ecc.u-tokyo.ac.jp

***Abstract*—Cluster scheduling based on the prior estimation of job execution time is vulnerable to inaccurate estimates. To avoid performance degradation due to this misestimation, recent studies have proposed cluster schedulers that do not rely on prior estimation. However, they do not assume tasks with multi-type heterogeneous computing resource demands, resulting in high job latency in real environments. Unfortunately, the optimal scheduling of such tasks is inherently difficult. In this paper, we present a cluster scheduler that heuristically handles multi-type heterogeneous resource demands without prior estimation. To reduce job latency, especially that of short jobs, our scheduler employs two techniques: (1) distributing tasks to nodes based on the similarity between resource demands and availability to simultaneously run as many tasks as possible, and (2) finding a suitable set of tasks for preemption in a node to minimize the number of task preemptions. Experimental evaluations using a real cluster and practical workloads confirm that our scheduler reduced the 90th percentile of slowdown rates by 6.4% and the 99th percentile by 29% compared to a naive extension of Kairos, an existing non-estimation-based scheduler. The experimental results also demonstrate that our scheduler is more effective when workloads have higher heterogeneity in resource demands.**

***Index Terms*—Cluster scheduling, resource management**

I. Introduction

Using computer clusters consisting of many nodes and large storage is now common for processing large amounts of data. A cluster is usually shared by many users in a company or a research institute, and therefore it must handle jobs with various characteristics. Each job consists of several tasks, and each task demands various types and amounts of computing resources (e.g., CPU, memory, and I/O bandwidth) specified by the users. Cluster schedulers are supposed to allocate the demanded resources to the tasks and manage job execution

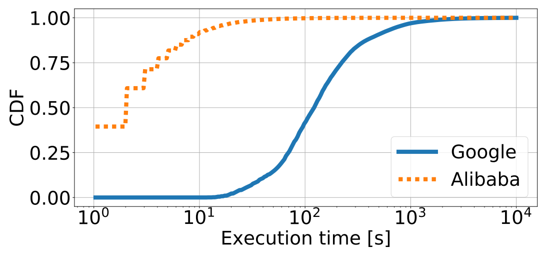

Fig. 1. CDF of task execution times extracted from cluster traces at Google and Alibaba (2018 version). The workloads consist of many short jobs and a few long jobs.

to minimize the latency of each job (i.e., the time from job submission to completion) and achieve high job throughput.

One of the characteristics of recent cluster workloads is heterogeneity in execution time [1], [2]. Fig. 1 plots the cumulative distribution function (CDF) of task execution times extracted from cluster traces at Google [3] and Alibaba (2018 version) [4]. While most tasks have short execution times of a few seconds to hundreds of seconds, a few tasks take a couple of hours. The coefficients of variation (CoV) reach up to 2.97 and 6.60 for Google and Alibaba traces, respectively. This is because typical modern cluster workloads are composed of short jobs, such as database queries and software development, and long jobs, such as large data conversion and analysis. Short jobs generally have stringent latency requirements, while long jobs can tolerate relatively high latency. Thus, reducing the latency of short jobs has become increasingly important for modern cluster schedulers.

To reduce the latency of short jobs in the face of heterogeneous execution time, most existing cluster schedulers rely on the prior estimation of job execution time [5]–[16]. For example, prioritizing a job that is expected to be short reduces the time until the job completes. The estimates are usually generated by a system or provided by users before scheduling. The accuracy of this estimation greatly affects the performance of estimation-based schedulers. Unfortunately, estimation systems often fail to achieve sufficient accuracy [17], and user predictions are generally also inaccurate [18], leading to sub-

© 2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Hidehito Yabuuchi, Takahiro Shinagawa. Multi-resource Low-latency Cluster Scheduling without Execution Time Estimation. In Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid 2020), pp. 310-319, May 2020. http://dx.doi.org/10.1109/CCGrid49817.2020.00-62

TABLE I
STATISTICS OF RESOURCE DEMANDS IN CLUSTER TRACES AT GOOGLE AND ALIBABA (2018 VERSION).

	CPU CoV	Memory CoV	Corr. Coef.
Google	0.81	0.91	0.33
Alibaba	0.38	0.34	0.28

optimal scheduling performance [19], [20].

To avoid performance degradation due to misestimation, recent studies have proposed cluster schedulers that do not rely on prior estimation [17], [21]–[23]. They reduce the latency of short jobs by, for example, approximating a job's execution time by its cumulative executed time (CET) and prioritizing jobs with short CETs. However, these schedulers either do not explicitly handle resource allocation, or they assume that each task demands a single-type homogeneous amount of resources. This limits their applicability to real-world clusters.

In fact, tasks in real environments demand multi-type resources, and their amounts are highly heterogeneous [1], [7]. Table I shows the statistics of resource demands in the Google and Alibaba traces. In both traces, tasks demand two types of resources: CPU and memory. In the Google trace, the CoVs of the CPU and memory demands reach 0.81 and 0.91, respectively. Moreover, the correlation coefficients between the CPU and memory demands are small in both traces: 0.33 and 0.28 in the Google and Alibaba traces, respectively. Grandl et al. reported that other production clusters also have little correlation between the demands of various resource types [7].

Cluster schedulers must consider these characteristics; otherwise, they will likely produce sub-optimal performances. If a scheduler considers only a particular type of resource, other resources might be overallocated. For example, the default schedulers of YARN [24] consider only memory demands, which can overallocate CPU and slowdown CPU-intensive jobs. Also, if a scheduler allocates resources in proportion to a single type of resource, as in slot-based schedulers, internal resource fragmentation can occur, leading to low utilization. Unfortunately, scheduling jobs with multi-type heterogeneous resource demands is far from trivial; it is analogous to the multi-dimensional online bin packing problem. The bin packing problem is NP-hard even in the one-dimensional offline case, and more difficult in the multi-dimensional online case.

In this paper, we present a cluster scheduler that heuristically handles multi-type heterogeneous resource demands without relying on the prior estimation of job execution time. Our scheduler is based on Kairos [17], which uses preemption to reduce the chance of head-of-line (HoL) blocking while not assuming prior estimation. Our goal in this paper is to extend Kairos to reduce job latency, especially that of short jobs, in the face of multi-type heterogeneous resource demands.

To achieve this goal, we introduce two heuristic techniques: (1) To simultaneously run as many tasks as possible on a cluster, a cluster-level *central scheduler* distributes a task to a node whose distribution of available resources is similar to that of the task's resources demand. This technique reduces the time from job submission to starting under medium and high loads. (2) To minimize the number of task preemptions, a *node scheduler* on each node finds a suitable set of running tasks for preemption that meets the multi-type heterogeneous resource demand of each waiting task. This technique minimizes the number of unnecessary preemptions, thus reducing the time from job starting to completion under high loads. Coupled together, these techniques reduce job latency, the time from job submission to completion.

We implemented a prototype of our scheduler by extending YARN and compared its performance against a naive extension of Kairos. On a 32-node real cluster and with workloads derived from the well-studied Google trace, our scheduler reduced the 90th percentile of slowdown rates by 6.4% and the 99th percentile by 29%. We also confirmed that our scheduler is more effective under higher heterogeneity in resource demands and that it incurs only negligible overhead.

The remainder of this paper is organized as follows: In Sec. II, we briefly discuss related work. We describe the design of our scheduler in Sec. III and its implementation on YARN in Sec. IV. We experimentally evaluate our scheduler in Sec. V. Finally, Sec. VI concludes the paper.

II. Related Work

A. Cluster Scheduling with Execution Time Estimation

To achieve low latency for short jobs in the face of heterogeneous execution time, most existing cluster schedulers rely on the prior estimation of execution time. EASY [5] introduced backfilling, which executes short jobs when a long job at the top of the queue cannot run due to resource shortage. Carastan-Santos et al. [6] argued that Smallest Area First policy with backfilling provides a performance improvement over EASY while maintaining simplicity. Tetris [7], Graphene [8], Yaq [9], and Peacock [10] reduce the latency of short jobs by prioritizing them through Shortest Remaining Time First policy.

Hybrid schedulers combine a centralized scheduler and a set of distributed schedulers to handle short and long jobs independently. Mercury [11] makes high-quality scheduling decisions for long jobs in a centralized component and quickly assigns short jobs to workers in distributed components. Hawk [12] and Eagle [13] reserve a portion of cluster resources to guarantee the immediate execution of short jobs. Pigeon [14], a hierarchical scheduler that divides workers into groups, also reserves some workers for short jobs.

Both Big-C [15] and Neptune [16] reduce the latency of short jobs by suspending long jobs and dynamically reassigning their resources to short jobs. Big-C leverages container-based virtualization and Neptune uses coroutines. Neptune assumes that each task demands a single CPU core.

The estimates are usually generated by a system or provided by users before scheduling. For a recurring job, a system can estimate the execution times of the job's tasks from its history [25]. Also, existing studies presented estimation techniques based on profiling [26] or execution in a simulated environment [27]. Unfortunately, they cannot efficiently estimate the execution time with sufficient accuracy [17], and user predictions are generally also inaccurate [18]. Scheduling jobs in the belief that the estimates are reliable will likely result in sub-optimal scheduling performance [19], [20].

To reduce estimation error, JamaisVu [28] tracks the estimation accuracy with different sets of job features and adaptively uses the most effective feature set. 3Sigma [19] mitigates performance degradation due to misestimation by leveraging full distributions of execution time history instead of point estimates. Although these studies had some success, estimation error is essentially inevitable, and estimation-based schedulers still cannot achieve optimal scheduling performance.

B. Cluster Scheduling without Estimation

As another approach to coping with the misestimation of execution time, recent studies have proposed cluster schedulers that do not rely on the prior estimation of execution time. A. Ilyushkin et al. presented several non-estimation-based scheduling algorithms for DAG-structured jobs and found that a backfilling-based algorithm could reduce job latency in practice [21]. Tyrex [22] avoids HoL blocking by partitioning cluster resources and migrating long-running jobs to a dedicated partition to make room for other jobs. Both LAS_MQ [23] and Kairos [17] exploit jobs' cumulative executed times by following Least Attained Service policy [29]. Kairos leverages task preemptions to reduce the chance of HoL blocking. Although these schedulers either do not explicitly handle resource allocation or assume that each task demands single-type homogeneous resources, tasks in real environments demand highly heterogeneous amounts of resources of multiple types [1], [7]. Scheduling without considering this characteristic results in sub-optimal performance.

C. Handling Multi-type Heterogeneous Resource Demands

Dominant Resource Fairness (DRF) [30] and its derivatives satisfy desirable properties for fairness among jobs in the face of multi-type heterogeneous resource demands. However, DRF does not focus on other performance metrics, such as job latency and throughput. To improve job throughput, Tetris increases resource efficiency by adopting a heuristic algorithm that packs multi-type heterogeneous resource demands into node capacities. However, Tetris relies on estimated execution time to reduce the latency of short jobs. Skewness-avoidance multi-resource allocation (SAMR) [31] efficiently allocates virtual machines that demand multi-type heterogeneous resources. SAMR works for IaaS clouds, which have virtually unlimited resources, meaning that it cannot be straightforwardly applied to most clusters, which have limited resources. RUPAM [32] is a cluster scheduler for Spark [33] and is aware of heterogeneity in both resource demands and underlying hardware. For each resource type, RUPAM greedily matches a node that has the lowest contention for the type to a task that likely has the type as its bottleneck. RUPAM does not consider multiple resource types simultaneously in its scheduling process. DeepPlace [34] learns a job scheduling strategy using deep reinforcement learning and is aware of multiple resource types. It has a scalability problem originating from the nature of deep reinforcement learning; as the number of nodes or the number of job types increases, the size of the search space increases exponentially. This makes the learning process take a long time and converge to a worse value.

III. Design

A. Assumptions

We assume that tasks are preemptible, i.e., they can be suspended when needed and resumed thereafter. Suspension preserves the execution progress of a preempted task, and the allocated resources become available to other tasks. At the resumption, the resources are restored to the preempted task, and it resumes running instead of starting again from the beginning. This preemption can be realized by using container-based virtualization as in Big-C [15].

For simplicity, we also assume that tasks in a job can be executed independently. Supporting dependencies between tasks in a job will be tackled in our future work.

B. Architecture Overview

We extend Kairos [17] to efficiently handle multi-type heterogeneous resource demands. Similar to Kairos, our proposed cluster scheduler has a two-level hierarchical architecture consisting of a *central scheduler* on a master node and a *node scheduler* on each worker node (Fig. 2). Submitted jobs are first stored in a queue at the central scheduler in order of their submission. The central scheduler distributes tasks in each job one by one consecutively to appropriate nodes. The task distribution algorithm is intended to be simple so that the central scheduler will not be a bottleneck in a cluster management system. We discuss the overhead and scalability of our cluster scheduler in Sec. V-E.

Then, the node schedulers locally manage the execution of the assigned tasks on each node. Upon a task assignment, a node scheduler immediately starts the task if the node has sufficient resources available for the task's demand (i.e., resources not allocated to other running tasks). Otherwise, the node scheduler tries to suspend one or more running tasks to make room for the new task. Waiting tasks (i.e., those not yet started or suspended) are stored in a per-node queue, and the node scheduler later tries to start or resume them. The node schedulers handle the assigned tasks locally, and by design, they do not support migrating tasks across nodes. This is because task migrations incur extra costs, such as building an execution environment and transferring input data and suspended progress [16], [17].

As mentioned in Sec. I, it is becoming increasingly important for modern cluster schedulers to reduce the latency of short jobs. We divide this goal into two parts and address them in separate scheduler components: (1) the central scheduler reduces the time from job submission to starting, and (2) the node schedulers reduce the time from job starting to completion. The following two subsections describe the scheduling methods employed in these components in detail. Note that

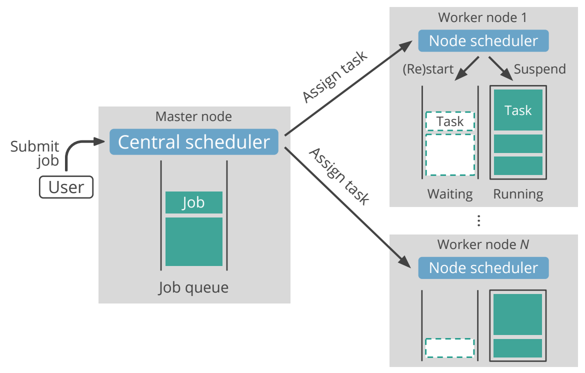

Fig. 2. Two-level hierarchical architecture of our cluster scheduler, which consists of a central scheduler and per-node schedulers.

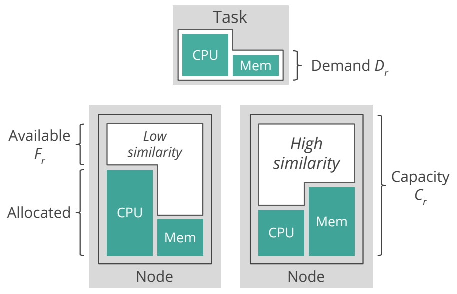

Fig. 3. Example of how nodes are compared by their similarity to a task. The right node has a higher similarity to the task.

improving job throughput is not the primary goal of our scheduler. Nevertheless, experimental evaluations (see Sec. V-B) show that our scheduler did *not* degrade job throughput.

C. Central Scheduler

1) Reducing the Time from Submission to Starting: In the central scheduler, we aim to reduce the time from job submission to starting. To achieve this goal, the central scheduler must simultaneously run as many tasks as possible on the cluster. This can be achieved by *packing* multi-type heterogeneous resource demands into the nodes' resource capacities.

When each task demands multi-type heterogeneous resources, the problem of packing tasks to nodes is analogous to the multi-dimensional online bin packing problem. Each task and node correspond to a ball and a bin, respectively. The balls and bins have the same dimensions as the number of resource types, and their sizes are proportional to the amounts of tasks' resource demands and nodes' capacities, respectively. Although the bin packing problem is NP-hard even in the one-dimensional offline case, and more difficult in the multi-dimensional online case, there exists a heuristic that efficiently packs balls into bins in the multi-dimensional cases [35]. We adapt this heuristic to the task distribution in the central scheduler to decide a node to which a task should be assigned.

In our heuristic, the central scheduler defines *similarity* between a task's resource demand and each node's availability, and it assigns the task to the node with the highest similarity. As an illustration, consider the situation depicted in Fig. 3. The central scheduler is trying to assign a task to one of two nodes. Because the task demands more CPU than memory and the right node has more available CPU than that of the left node, the central scheduler considers the right node to be more similar to the task than the left node. The nodes do not need to have lots of available memory because the task demands only a small amount of memory. By using this heuristic, the resources on each node will be efficiently used up for all types. Note that Tetris [7] adopts a similar heuristic for cluster scheduling, but it is done in a different way to ours. While Tetris uses the heuristic (coupled with estimated execution time) to decide which task to execute next, our central scheduler uses it to decide a node to which a task should be assigned.

Specifically, given a task and a list of nodes, the central scheduler calculates *similarity scores* between the task and each node. The similarity score between a task and a node is the dot product of the demanded and available resources:

$$\mathrm{Similarity}(\mathrm{task}, \mathrm{node}) = \sum_{r \in \mathcal{R}} \frac{D_r F_r}{{C_r}^2} \qquad (1)$$

where $\mathcal{R}$ is the set of resource types, and D_r and F_r are the amounts of resources demanded by the task and available on the node, respectively, for type r. To ensure that this formula is invariant to the units of resources, D_r and F_r are normalized by C_r, the node capacity. F_r is calculated as the difference between C_r and the total resource demand of the assigned and uncompleted tasks, and it can thus be negative if the total demand exceeds C_r. If a node has a higher load than a threshold (see Sec. III-C2), the node is removed from the node list. Then, the central scheduler assigns the task to the node with the highest similarity score, if it exists. In the case of Fig. 3, because the task demands more CPU than memory, the resource availability is weighted more heavily towards CPU. The right node, which has more available CPU than that of the left node, thus has a higher similarity score.

2) Load Balancing: To reduce the time from job submission to starting, the central scheduler may assign a task to a node even when all nodes are over-loaded, i.e., do not have sufficient resources available for the task. In this case, the newly assigned task preempts one or more running tasks on the node and completes quickly if it has only a short execution time. When distributing a task to over-loaded nodes, it is essential to balance the loads among nodes [9], [36], [37]. If a particular node has a much higher load than those of others, tasks on the highly-loaded node will get fewer chances to run, resulting in higher latency of the job it belongs to.

To balance the loads among nodes, the above task distribution algorithm (Sec. III-C1) works automatically. By definition, a highly-loaded node has a low or negative amount of available resources (F_r), and its similarity score calculated by Eq. 1 is thus lower than those of other less-loaded nodes. Conversely, the node with the highest similarity score has the lowest load. Note that, because the similarity scores depend on the task's resource demand (D_r), the values of *loads* also vary with what task is to be distributed.

In case of a much higher cluster load, the central scheduler does not assign a task to a node whose *load factor* is higher than a configurable threshold L. The load factor is defined as the norm of the normalized total resource demand of the assigned and uncompleted tasks:

$$\mathrm{LoadFactor}(\mathrm{node}) = \sqrt{\sum_{r \in \mathcal{R}} \left(\frac{\sum D_r}{C_r} \right)^2}. \qquad (2)$$

If all nodes have higher load factors than L, the central scheduler waits until the load factor of at least one node falls below L. The remaining tasks are kept in the central scheduler and are distributed to low-loaded nodes later so that the tasks will get more chances to run.

If L is small, under high loads, fewer tasks will be distributed to nodes. This means that many tasks will have to wait in the central scheduler, but tasks running on the nodes will be preempted fewer times. On the contrary, if L is large, the wait time in the central scheduler will be reduced, but running tasks will be preempted more often. Sensitivity analysis on the setting of L (see Sec. V-D) suggests that our cluster scheduler is robust against sub-optimal settings.

Note that the scheduling algorithm of the central scheduler does *not* assume that all nodes have the same resource capacity. The similarity scores (Eq. 1) and load factors (Eq. 2) are calculated for each node independently, with normalization by its capacity. Thus, the algorithm can handle heterogeneous nodes within a cluster without modification. Also note that the algorithm does *not* limit the resource types to CPU and memory. It can, in theory, handle other types, such as I/O bandwidth and GPU.

3) Placement Constraints: Modern cluster workloads often have placement constraints [38], [39], which are divided into two categories: (1) tasks with hard constraints can run only on particular types of nodes that satisfy a specific condition, and (2) tasks with soft constraints prefer particular types of nodes over others, but can run on others at the cost of slowdown.

The central scheduler handles hard constraints by filtering out nodes that do not satisfy the condition. To deal with soft constraints, our scheduler lets users and other system components specify node preferences by numbers for each task. Then, the central scheduler weights the similarity scores by the preference values for each node and picks the node with the highest weighted score. The setting of the preference is up to users and other system components because the exact setting depends on the cluster configuration and workloads.

D. Node Scheduler

1) Reducing the Time from Starting to Completion: As mentioned in Sec. I, modern cluster workloads have heterogeneity in execution time. Short jobs generally have stringent latency requirements, while long jobs can tolerate relatively high latency. Thus, the node schedulers should reduce the latency of short jobs by avoiding HoL blocking. To achieve this, under high loads, the node schedulers must preempt running tasks of long jobs and (re)start waiting tasks of short jobs instead. The problem is that our cluster scheduler does not assume prior information about the execution time.

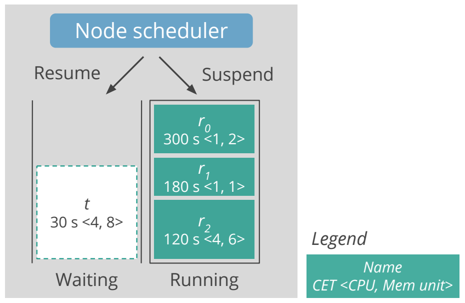

Fig. 4. Example situation where naive LAS causes an unnecessary preemption. It preempts all running tasks (r_0, r_1, and r_2) to resume t, but r_1 does not need to be preempted to suffice for t's resource demand.

To tackle this problem, our node schedulers use an algorithm based on Least Attained Service (LAS) policy [29], similar to LAS_MQ [23] and Kairos [17]. To reduce the latency of short jobs without the prior estimation of execution time, LAS approximates a task's true execution time by its *cumulative executed time* (CET; also called *attained service*) and prioritizes tasks with short CETs. LAS avoids HoL blocking by preempting long-CET running tasks and (re)starting short-CET waiting tasks instead. Because LAS is particularly effective when a workload has high variance in execution time [40], it is expected to work well for modern cluster workloads.

Unfortunately, the naive employment of LAS can cause unnecessary preemptions when tasks demand heterogeneous resources. Due to this heterogeneity, a preempted task may not offer the resources demanded by a waiting task. If a node scheduler preempts running tasks greedily in decreasing order of CET until it obtains sufficient resources, one or more running tasks may be unnecessarily preempted. As an illustration, consider the case depicted in Fig. 4. Let the waiting task to resume be t, and suppose that there are three running tasks, r_0, r_1, and r_2, in decreasing order of CET and the whole node capacity is separately allocated to them. Because t has a shorter CET than those of the running tasks, the node scheduler tries to preempt one or more running tasks and resume t instead. The naive LAS will greedily preempt all running tasks before it obtains sufficient resources for t. However, r_1 does not need to be preempted; preempting only r_0 and r_2 offers 5 CPU cores and 8 units of memory, which sufficiently meet t's demand.

Because preempted tasks must wait until they are resumed later and will suffer late completion, the number of task preemptions should be minimized. Although LAS preferentially preempts tasks with long CETs, if a node scheduler performs unnecessary preemptions, it will eventually preempt tasks with medium or short CETs, especially under high loads. Moreover, even for long jobs, cluster schedulers should not compromise their latency.

Algorithm 1 Generating preemption candidates

```
Input runningTasks                ▷ Sorted in decreasing order of CET
Output Yielding preemption candidates one by one

1: candidates = []
2: for r in runningTasks[..N] do                    ▷ First $N$ items
3:     yield r        ▷ Candidate consisting of a single running task
4:     candidates.append(r)

5:     len = candidates.length()
6:     for i from 0 to len - 2 do ▷ Combine r and other tasks
7:         c = combine(r, candidates[i])
8:         yield c
9:         candidates.append(c)
```

Algorithm 2 Node scheduler

```
Input newTasks,                    ▷ Sorted in order of submission
      suspendedTasks,        ▷ Sorted in increasing order of CET
      runningTasks,          ▷ Sorted in decreasing order of CET
      Resource usage of the node

 1: for t in newTasks do
 2:     TrySchedule(t, runningTasks)

 3: runningTasks                              ▷ Prevent starvation
 4:   .filter(execTimeSinceStart > period)
 5: for t in suspendedTasks do
 6:     runningTasks.filter(CET > t.CET)
 7:             ▷ Filter out running tasks with shorter CET than t's
 8:     TrySchedule(t, runningTasks)

 9: function TRYSCHEDULE(waitingTask, runningTasks)
10:     if node has sufficient resources for waitingTask then
11:         (Re)start waitingTask
12:         Update resource usage of the node
13:         return
14:     for c in preemption candidates generated by Alg. 1 do
15:         if c offers sufficient resources for waitingTask then
16:             Preempt c and (re)start waitingTask
17:             Remove c from runningTasks
18:             Update resource usage of the node
19:             return
20:     return                       ▷ Failed to (re)start waitingTask
```

To avoid unnecessary preemptions, we modify LAS to find a suitable set of running tasks for preemption that offers sufficient resources for accommodating a waiting task. This is done by examining the total allocated resources for each set of running tasks. Specifically, letting running tasks be $r_0,\ r_1,\ r_2,\ \ldots$ in decreasing order of CET, a node scheduler scans sets of running tasks in the following order to generate *preemption candidates*:

$$\{r_0\},$$
$$\{r_1\},\ \{r_1,\ r_0\},$$
$$\{r_2\},\ \{r_2,\ r_0\},\ \{r_2,\ r_1\},\ \{r_2,\ r_1,\ r_0\},$$
$$\ldots$$

If a running task does not offer sufficient resources for a waiting task, the node scheduler combines it with other longer-CET running tasks and checks whether the combination offers sufficient resources. In the case of Fig. 4, because r_2 (as well as any combination of r_0 and r_1) does not offer sufficient resources for t, the node scheduler skips preempting only r_2 and proceeds to combine it with r_0 and/or r_1. Then, the node scheduler realizes that the set $\{r_2,\ r_0\}$ will suffice for t's demand and preempts them, leaving r_1 untouched.

2) Algorithm: Alg. 1 shows the algorithm to generate preemption candidates. For each running task, this algorithm first generates a preemption candidate consisting of the running task alone (Lines 3 and 4). Then, it generates other preemption candidates by combining the task with previously-generated candidates (Lines 5–9).

When a node has n running tasks, the number of possible preemption candidates reaches $2^n - 1$. Because scanning all candidates in the case of large n will take a long time, Alg. 1 considers up to the first N running tasks in decreasing order of CET (Line 2). If preempting all N running tasks will not suffice for the demand of a waiting task, the node scheduler skips (re)starting the waiting task and proceeds to process another waiting task. In our experiments (see Sec. V-B), N did not need to be larger than 3 or 4.

Alg. 2 shows the overall algorithm of a node scheduler based on the modified LAS described above. First, it tries to start new tasks (i.e., those not yet started) immediately upon assignment (Lines 1 and 2). Because most tasks require only short durations to complete (as seen in Fig. 1), immediately starting newly assigned tasks reduces the latency of short jobs with high probability. If the node's available resources are not sufficient for the task's demand, the node scheduler tries to preempt one or more running tasks using the modified LAS and start the new task instead (Lines 14–19). If it fails to find a preemption candidate that offers sufficient resources, it skips starting the new task (Line 20). The node scheduler performs this process for all new tasks in order of job submission.

Then, the node scheduler tries to resume suspended tasks in increasing order of CET (Lines 5–8). To ensure that tasks are not preempted too frequently, the node schedulers implement a starvation prevention mechanism. If a running task has not been running for a certain *no-interference period* since its most recent (re)start, the task is filtered out from the list of running tasks (Lines 3 and 4). The length of the no-interference period is calculated as $W \times (P+1)$, where W is a configurable duration and P is the number of times the task has been preempted. Tasks with shorter execution times than W will likely complete quickly because they are not preempted by other suspended tasks. Thus, the value of W should be configured so that roughly 20% to 40% of tasks in a workload will have execution times shorter than W. The period becomes longer every time the task is preempted, which prevents an unfair situation where long tasks are preempted many times.

IV. Implementation

We implemented a prototype of our cluster scheduler by extending YARN [24], which is widely used as the resource management framework for Hadoop [41]. In YARN, a ResourceManager on a master node arbitrates the cluster-wide resource allocation among jobs. It defines a scheduler interface and has some implementations, including the default `CapacityScheduler`. The scheduler distributes tasks to worker nodes by communicating with `NodeManager` daemons running on each worker node. A `NodeManager` allocates a logical bundle of resources to each assigned task, and a `ContainerManager` in the node manages task execution.

We implemented our central scheduler by extending the `CapacityScheduler`. The implementation either distributes tasks to worker nodes or waits if all nodes have higher loads than a threshold, as described in Sec. III-C.

We implemented a node scheduler as a new thread spawned from the `ContainerManager`. The thread runs a scheduling loop periodically (every $1\,\mathrm{s}$ in our experiments), where the node scheduler updates the CETs of running tasks, and (re)starts or suspends tasks by following Alg. 2. As the implementation of task preemptions, our node schedulers use a container-based lightweight mechanism [15], as in Kairos. It executes a task in a container and preempts the task by dynamically reducing the resources allocated to the container.

V. Evaluation

We evaluated our cluster scheduler using a real cluster and workloads derived from a production environment. Here, we report the following experiments:

- In Sec. V-B, we compare the performance of our scheduler against YARN [24] and Kairos [17].
- In Sec. V-C, we show the impact of the degree of heterogeneity in tasks' resource demands.
- In Sec. V-D, we analyze the sensitivity of our scheduler to different parameter settings.
- In Sec. V-E, we evaluate the overhead and scalability of the scheduling algorithm.

A. Setup

We used a cluster of CloudLab [42], [43] for evaluation. The cluster consisted of 32 bare-metal `c220g5` nodes: one master (`ResourceManager`) and 31 workers (`NodeManagers`). We configured each worker to manage 16 cores/32 threads of Intel Xeon Silver 4114 processors and $64\,\mathrm{GB}$ RAM. Nodes were connected at $10\,\mathrm{Gbps}$. We installed Ubuntu 18.04 (Linux kernel 4.15.0) and JDK 1.8.0 on all nodes.

To create realistic workloads, we first approximated the empirical distributions of tasks extracted from the well-studied Google trace [3] in terms of execution time, CPU demands, and memory demands with separate log-normal distributions. We used log-normal distributions because the empirical distributions can be approximated well by power-law-like heavy-tailed distributions [1]. We truncated the distribution of execution times at $45\,\mathrm{min}$. 96% of tasks had execution times shorter than $45\,\mathrm{min}$. Then, we sampled values from the distributions to generate 500 Hadoop `WordCount` jobs. Each job had eight identical mapper tasks and one reducer task, and each task demanded the sampled numbers of cores and amount of memory. We rounded values of memory demands to multiples of $512\,\mathrm{MB}$. We modified the `WordCount` source code to conduct an extra computation while consuming the demanded resources so that we could increase a task's execution time by a controllable amount. A task would take the duration of the sampled value to complete if it had been executed alone on the cluster. The job inter-arrival time followed a Poisson distribution with $\lambda = 5$ so that we could evaluate the cluster schedulers under medium- and high-load conditions. The resulting workloads took approximately $100\,\mathrm{min}$ to complete.

We compared our scheduler against YARN Capacity Scheduler, *Random* scheduler, and Kairos. Random scheduler shares the central scheduler with ours, but its node schedulers preempt running tasks at random until they obtain sufficient resources for a waiting task. Because Kairos assumes single-type homogeneous resource demands, we modified it naively so that we could input multi-type heterogeneous resource demands. Specifically, we modified Kairos node schedulers to greedily preempt running tasks in decreasing order of CET until they obtain sufficient resources for a waiting task. For the parameters of our scheduler, unless otherwise noted, we set the threshold of load factors to $L = 2.0$, the maximum number of running tasks to be considered for preemption to $N = 4$, and the base length of no-interference periods to $W = 120\,\mathrm{s}$. Approximately 28% of tasks had execution times shorter than $120\,\mathrm{s}$. We used Hadoop 2.7.7, the latest version of the 2.7 series on which the original Kairos was implemented, as the base source code. A task ran in a container using Docker 19.03.5 with an image `sequenceiq/hadoop-docker:2.4.1`.

B. Job Performance

We evaluated the cluster schedulers based on the distributions of *slowdown rates*. The slowdown rate of a job is defined as the actual job latency divided by the hypothetical execution time if the job was executed alone on the cluster. The distribution of slowdown rates is thus an index for measuring latency with emphasis on short jobs.

Fig. 5 plots the CDF of slowdown rates for the workloads. Compared to Kairos, our scheduler reduced the 90th percentile of slowdown rates by 6.4% and the 99th percentile by 29%. Our scheduler was more effective in reducing tail latency; the maximum slowdown rate was 47% lower than that of Kairos. Compared to YARN and Random scheduler, our scheduler reduced the 90th percentile by 61% and 46%, and the 99th percentile by 75% and 63%, respectively. Average latencies were $906\,\mathrm{s}$, $1006\,\mathrm{s}$, $835\,\mathrm{s}$, and $786\,\mathrm{s}$ with YARN, Random scheduler, Kairos, and our scheduler, respectively.

One reason for this improvement is that our central scheduler distributes tasks to nodes so that as many tasks as possible will run simultaneously on a cluster. Fig. 6 shows the numbers of waiting and running tasks on the nodes with Kairos and our scheduler during the experiments. The lines show the averages of all nodes, and the filled ranges show the intervals of two standard deviations from the averages. Our scheduler executed more tasks than that of Kairos with less deviation under the medium- to high-load conditions. This result implies that our scheduler successfully reduced the time from job submission to starting by distributing tasks based on the similarity between resource demands and availability. Fig. 6 also shows that our

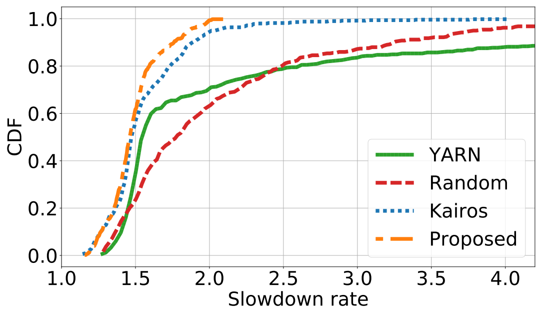

Fig. 5. CDF of slowdown rates. (The tails of YARN and Random are truncated for visibility; the maximum values were 9.9 and 6.4, respectively.)

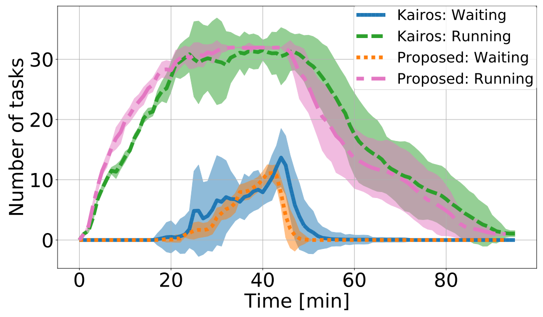

Fig. 6. Numbers of waiting and running tasks on the nodes. Lines show the averages of all nodes, and filled ranges show two standard deviations.

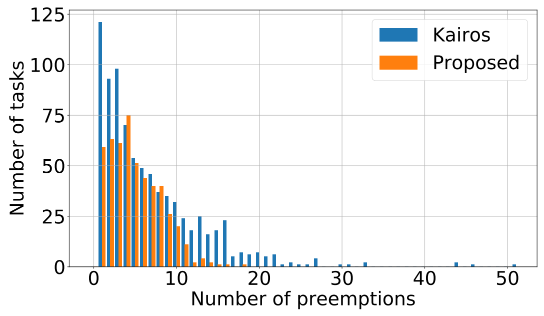

Fig. 7. Distribution of numbers of preemptions.

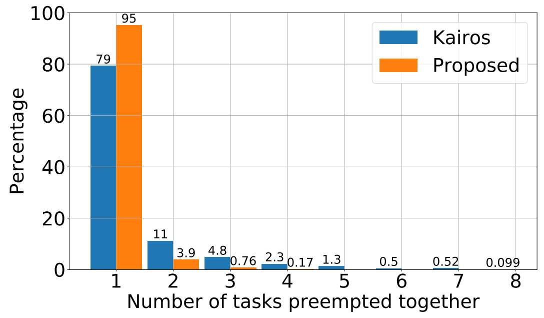

Fig. 8. Distribution of numbers of running tasks preempted together in preemption processes to make room for a waiting task.

scheduler did *not* degrade job throughput. Rather, as seen in the right of the figure, the jobs in the workload completed at a faster pace than that of Kairos.

Another reason for the improvement is that our node schedulers reduce the number of preemptions, allowing more tasks to continue running without interference. Our scheduler reduced the total number of preemptions by 38.3% compared to Kairos (from 812 to 501). Fig. 7 shows the distribution of the numbers of preemptions. Our scheduler preempted tasks no more than 18 times, while Kairos preempted tasks up to 51 times. This is because our node schedulers have the starvation prevention mechanism to prevent frequent preemptions (Sec. III-D2). Thus, the number of preemptions was kept small, preventing an unfair situation where a task suffered a severe slowdown. This can also be seen in the low maximum slowdown rate in Fig. 5.

Fig. 8 shows the distribution of the numbers of running tasks preempted together in the preemption processes (Lines 14–19 in Alg. 2) to make room for a waiting task. With our scheduler, 95% of preemption processes preempted only a single running task, while with Kairos, the percentage was 79%. Our scheduler minimized the number of tasks preempted in each preemption process by examining the heterogeneous resource demands of waiting and running tasks. Only 0.17% of the processes preempted four tasks, meaning that N (the maximum number of running tasks to be considered for preemption) could be set as small as 3.

C. *Impact of Heterogeneity*

We next measured the effectiveness of our scheduler when workloads have higher heterogeneity in tasks' resource demands. To do so, we modified the log-normal distributions for the CPU demands and memory demands by scaling the standard deviations of their underlying normal distributions by various numbers. In this experiment, we reduced the cluster size to 8 nodes, generated 100-job workloads, and changed the average job inter-arrival time to $\lambda = 24$, allowing us to conduct experiments with many parameter settings.

Fig. 9 shows the distribution of slowdown rates while varying the scaling factor. As the scaling factor increased, the workloads had higher heterogeneity in resource demands. In all box plots in this paper, the whiskers are at the 5th and 99th percentiles. With Kairos, the slowdown rates increased as the degree of heterogeneity increased. In contrast, our scheduler maintained low slowdown rates, almost regardless of the degree of heterogeneity. This result suggests that our scheduler minimized the impact of high heterogeneity in resource demands.

D. *Parameter Sensitivity*

We next analyzed how our scheduler performs under different settings of L (threshold of load factors) and W (base length of no-interference periods). Similar to Sec. V-C, we used an 8-node cluster and 100-job workloads.

Fig. 10 shows the distribution of slowdown rates with Kairos and our scheduler with different settings of L and W values. For L, while the settings $L = 1.5,\ 2.5,\ 3.0$ gave worse

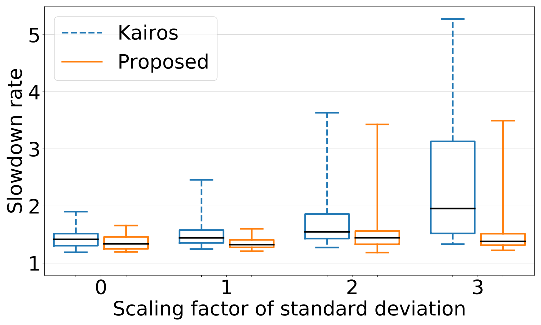

Fig. 9. Distribution of slowdown rates with variable heterogeneities in resource demands. Large scaling factors indicate that workloads had high heterogeneities.

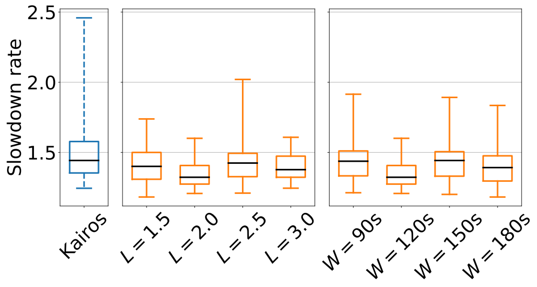

Fig. 10. Distribution of slowdown rates with variable settings of L and W.

performances than the best-performing $L = 2.0$, they still outperformed Kairos. For W, the duration $90\,\mathrm{s}$, $120\,\mathrm{s}$, $150\,\mathrm{s}$, and $180\,\mathrm{s}$ were longer than the execution times of approximately 22%, 28%, 34%, and 39% of tasks, respectively. As with L, the sub-optimal settings of W performed worse than the best setting $W = 120\,\mathrm{s}$, but our scheduler consistently performed better than Kairos. These results imply that our scheduler has sufficient robustness against sub-optimal parameter settings.

E. Overhead and Scalability

Finally, we evaluated the overhead our scheduling algorithm imposes on a cluster management system and the scalability of our central scheduler in terms of the number of nodes.

1) Central Scheduler: Because all submitted jobs are first processed in the central scheduler, it could be a bottleneck for a cluster management system if it incurs high overhead. Some modern clusters have more than 10,000 nodes [44], and thus the central scheduler should process jobs within several milliseconds for such large clusters. To quantify this overhead on a large scale, we extracted the algorithm of the central scheduler into a standalone Java program and measured its computation time with dummy nodes and dummy job input. The nodes and tasks had resource capacities and demands of randomly selected values. We first warmed up the JVM and then input 2^{16} dummy jobs for each experiment.

Fig. 11 plots the distribution of durations for the central scheduler to perform its scheduling algorithm for a task with variable numbers of nodes. Because the algorithm calculates a similarity score and a load factor for each node and identifies the node with the maximum score, its computational complexity is $O(N)$, where N is the number of nodes. The experimental results agree with this theory, and the computation time grew linearly as the number of nodes increased. We also observed that the central scheduler incurred only negligible overhead. It took less than $1\,\mathrm{ms}$ in most cases with $2^{14} (> 10,000)$ nodes. Even with 2^{16} nodes, the computation time was as small as several milliseconds in most cases.

2) Node Scheduler: The node schedulers perform more complicated computations compared to the central scheduler. Although they run in parallel on each worker node, they should process assigned tasks with low overhead to reduce job latency. Fig. 12 plots the distribution of durations for the node schedulers to run the scheduling loop (Sec. IV) once. Similar to Sec. V-C, we used an 8-node cluster and a 100-job workload. The duration data is aggregated from all worker nodes and does not include data where there were no waiting tasks. We found that the node schedulers took less than several milliseconds for the scheduling in most cases.

Combining the results from Sec. V-E1 and Sec. V-E2, we believe that our cluster scheduler imposes only negligible overhead on a cluster management system and that it is scalable to more than 10,000 nodes.

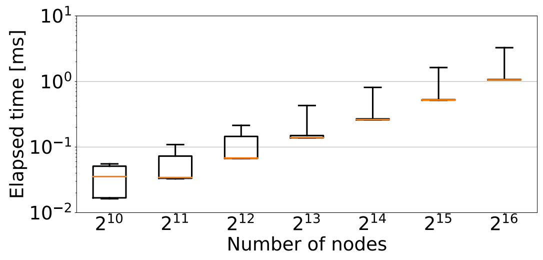

Fig. 11. Distribution of durations for the central scheduler to schedule a task with variable numbers of nodes (log-log scale).

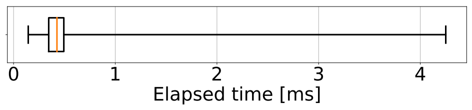

Fig. 12. Distribution of durations for the node schedulers to run the scheduling loop once.

VI. Conclusion

In this paper, we present a cluster scheduler that heuristically handles multi-type heterogeneous resource demands without relying on the prior estimation of execution time. To reduce job latency, especially that of short jobs, our scheduler employs two techniques: (1) distributing tasks to nodes based on the similarity between resource demands and availability, and (2) finding the minimal set of tasks for preemption by examining the heterogeneous resource demands. Experimental evaluations on a real cluster and with practical workloads demonstrate that our scheduler reduces job latency without imposing high overhead and that it is more effective under higher heterogeneity in resource demands.

Future directions include incorporating job-aware strategies into our scheduler to support complicated job structures and inter-task dependencies. We also plan to explore the integration with estimation-based scheduling methods.

Acknowledgment

We are grateful to CloudLab for hosting the computing infrastructure on which we conducted our large-scale experiments. We also would like to thank Editage (www.editage.com) for English language editing.

References

[1] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[2] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the Cloud: an Analysis on Alibaba Cluster Trace," in *Proceedings of the 2017 IEEE International Conference on Big Data*, 2017, pp. 2884–2892.

[3] Google. Borg cluster traces from Google. https://github.com/google/cluster-data.

[4] Alibaba. cluster data collected from production clusters in Alibaba for cluster management research. https://github.com/alibaba/clusterdata.

[5] D. A. Lifka, "The ANL/IBM SP Scheduling System," in *Proceddins of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1995, pp. 295–303.

[6] D. Carastan-Santos, R. Y. de Camargo, D. Trystram, and S. Zrigui, "One can only gain by replacing EASY backfilling: A simple scheduling policies case study," in *Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2019, pp. 1–10.

[7] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-Resource Packing for Cluster Schedulers," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014, pp. 455–466.

[8] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 81–97.

[9] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient Queue Management for Cluster Scheduling," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[10] M. Khelghatdoust and V. Gramoli, "Peacock: Probe-Based Scheduling of Jobs by Rotating Between Elastic Queues," in *Proceedings of the 24th International European Conference on Parallel and Distributed Computing*, 2018, pp. 178–191.

[11] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters," in *Proceedings of the 2015 USENIX Annual Technical Conference*, 2015, pp. 485–497.

[12] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid Datacenter Scheduling," in *Proceedings of the 2015 USENIX Annual Technical Conference*, 2015, pp. 499–510.

[13] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware Scheduling in Eagle: Divide and Stick to Your Probes," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 497–509.

[14] Z. Wang, H. Li, Z. Li, X. Sun, J. Rao, H. Che, and H. Jiang, "Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler," in *Proceedings of the ACM Symposium on Cloud Computing 2019*, 2019, pp. 246–258.

[15] W. Chen, J. Rao, and X. Zhou, "Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization," in *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017, pp. 251–263.

[16] P. Garefalakis, K. Karanasos, and P. Pietzuch, "Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications," in *Proceedings of the ACM Symposium on Cloud Computing 2019*, 2019.

[17] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Kairos: Preemptive Data Center Scheduling Without Runtime Estimates," in *Proceedings of the ACM Symposium on Cloud Computing 2018*, 2018, pp. 135–148.

[18] A. W. Mu'alem and D. G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[19] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger, "3Sigma: Distribution-based cluster scheduling for runtime uncertainty," in *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[20] A. Ilyushkin and D. Epema, "The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflow," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2018, pp. 331–341.

[21] A. Ilyushkin, B. Ghit, and D. Epema, "Scheduling Workloads of Workflows with Unknown Task Runtimes," in *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2015, pp. 606–616.

[22] B. Ghit and D. Epema, "Tyrex: Size-Based Resource Allocation in MapReduce Frameworks," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2016, pp. 11–20.

[23] Z. Hu, B. Li, Z. Qin, and R. S. M. Goh, "Job Scheduling without Prior Information in Big Data Processing Systems," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*, 2017, pp. 572–582.

[24] 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 *Proceedings of the 2013 ACM Symposium on Cloud Computing*, 2013.

[25] 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 Symposium on Networked Systems Design and Implementation*, 2012.

[26] C. Delimitrou and C. Kozyrakis, "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, pp. 127–144.

[27] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed Job Latency in Data Parallel Clusters," in *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012, pp. 99–112.

[28] A. Tumanov, A. Jiang, J. W. Park, M. A. Kozuch, and G. R. Ganger, "JamaisVu: Robust Scheduling with Auto-Estimated Job Runtimes," Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep., 2016.

[29] M. Nuyens and A. Wierman, "The Foreground-Background queue: A survey," *Performance Evaluation*, vol. 65, pp. 286–307, 2008.

[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[31] L. Wei, C. H. Foh, B. He, and J. Cai, "Towards Efficient Resource Allocation for Heterogeneous Workloads in IaaS Clouds," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 264–275, 2018.

[32] L. Xu, A. R. Butt, S.-H. Lim, and R. Kannan, "A Heterogeneity-Aware Task Scheduler for Spark," in *Proceedings of the 2018 IEEE International Conference on Cluster Computing*, 2018, pp. 245–256.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[34] S. Mitra, S. S. Mondal, N. Sheoran, N. Dhake, R. Nehra, and R. Simha, "DeepPlace: Learning to Place Applications in Multi-Tenant Clusters," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019, pp. 61–68.

[35] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for Vector Bin Packing," Microsoft Research, Tech. Rep., 2011.

[36] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 69–84.

[37] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 285–300.

[38] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, “Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[39] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, “Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[40] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack, “Analysis of LAS Scheduling for Job Size Distributions with High Variance,” in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003, pp. 218–228.

[41] Apache Hadoop. https://hadoop.apache.org.

[42] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, “The Design and Operation of CloudLab,” in *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019.

[43] CloudLab. https://www.cloudlab.us.

[44] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand, “Firmament: Fast, Centralized Cluster Scheduling at Scale,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 99–115.