Kairos: Preemptive Data Center Scheduling Without Runtime Estimates

Pamela Delgado[1], Diego Didona[1], Florin Dinu[1,2] and Willy Zwaenepoel[1,2]

[1]*EPFL, Switzerland* [2]*University of Sydney, Australia*

ABSTRACT

The vast majority of data center schedulers use task runtime estimates to improve the quality of their scheduling decisions. Knowledge about runtimes allows the schedulers, among other things, to achieve better load balance and to avoid head-of-line blocking. Obtaining accurate runtime estimates is, however, far from trivial, and erroneous estimates lead to sub-optimal scheduling decisions. Techniques to mitigate the effect of inaccurate estimates have shown some success, but the fundamental problem remains.

This paper presents Kairos, a novel data center scheduler that assumes no prior information on task runtimes. Kairos introduces a distributed approximation of the Least Attained Service (LAS) scheduling policy. Kairos consists of a centralized scheduler and per-node schedulers. The per-node schedulers implement LAS for tasks on their node, using preemption as necessary to avoid head-of-line blocking. The centralized scheduler distributes tasks among nodes in a manner that balances the load and imposes on each node a workload in which LAS provides favorable performance.

We have implemented Kairos in YARN. We compare its performance against the YARN FIFO scheduler and Big-C, an open-source state-of-the-art YARN-based scheduler that also uses preemption. Compared to YARN FIFO, Kairos reduces the median job completion time by 73% and the 99th percentile by 30%. Compared to Big-C, the improvements are 37% for the median and 57% for the 99th percentile. We evaluate Kairos at scale by implementing it in the Eagle simulator and comparing its performance against Eagle. Kairos improves the 99th percentile of short job completion times by up to 55% for the Google trace and 85% for the Yahoo trace.

SoCC'18, Oct 11-13, Carlsbad, CA, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

CCS CONCEPTS

• **Software and its engineering** → *Scheduling*;

KEYWORDS

Cloud computing, Data center, Scheduling

ACM Reference Format:
Pamela Delgado[1], Diego Didona[1], Florin Dinu[1,2] and Willy Zwaenepoel[1,2] [1]*EPFL, Switzerland* [2]*University of Sydney, Australia* . 2018. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of ACM Symposium of Cloud Computing conference, Carlsbad, CA, USA, Oct 11-13 (SoCC'18),* 14 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Modern data centers face increasingly heterogeneous workloads composed of long batch jobs, e.g., data analytics, and latency-sensitive short jobs, e.g., operations of user-facing services. Scheduling such jobs while achieving low scheduling times, good job placement and high resource utilization is a challenging task. The complexity is exacerbated by the data-parallel nature of these jobs: a job is composed of multiple tasks, and the job completes only when all of its tasks complete.

Many state-of-the-art systems rely on estimates of the runtimes of tasks within a job to improve the quality of their scheduling decisions in the face of job heterogeneity and data-parallelism [4, 18, 20, 21, 29, 30, 35]. Execution times from prior runs [4] or a preliminary profiling phase [12] are often used for this purpose. The accuracy of such estimates has a significant impact on the performance of these schedulers. For instance, queueing a 1-second task behind a task that is estimated to take 1 second but in reality takes 3 seconds doubles the completion time of the queued task. Similarly, scheduling at the same time two tasks estimated to be of equal length may seem to provide excellent load balance, but in fact significant load imbalance occurs if one task turns out to be shorter and the other longer.

Limitations of estimates-based approaches. Unfortunately, obtaining accurate task runtime estimates is not trivial. We show in §2 that a widely employed estimation technique – using the mean task execution time as a predictor of the execution of all tasks in a job [10, 30] – can lead to large errors

(> 100%). Our findings are confirmed by recent work that shows that more sophisticated approaches based on machine learning [29] still exhibit significant estimation errors. Many factors contribute to the difficulty of obtaining reliable runtime estimates. For example, small changes in the input data of a recurring job may substantially change the execution time [1] of its tasks, thus compromising the accuracy of estimates based on previous executions. Data skew may lead a few tasks in a job to take considerably more time to complete than other tasks in the same job [8, 26]. Techniques to tackle these issues, such as queue re-balancing [30] or uncertainty-aware scheduling policies [29], have shown some success in mitigating the impact of misestimations, but they do not fundamentally address the problem.

Kairos. In this paper we introduce an alternative approach to data center scheduling, which does not use task runtime estimates. Our approach draws from the Least Attained Service (LAS) scheduling policy [27]. LAS is a preemptive scheduling technique that selects for execution the task that has received the smallest amount of service so far. LAS is known to achieve good task completion times when the distribution of task runtimes has high variance, as is the case in heavy-tailed data center workloads that are common in data centers.

The main challenge is to find a good approximation for LAS in a data center environment. A naive implementation would cause frequent task migrations, with their attendant performance penalties. Instead, we have developed a two-level scheduler that avoids task migrations altogether, but still offers good performance. In particular, Kairos consists of a centralized scheduler and per-node schedulers. The per-node schedulers implement LAS for tasks on their node, using preemption as necessary to avoid head-of-line blocking. The centralized scheduler distributes tasks among worker nodes in a manner that addresses the following two challenges.

A first challenge is to ensure high resource utilization in the absence of runtime estimates. To address this issue, the central scheduler aims to equalize the number of tasks per node, and reduces the amount of load imbalance possible among nodes by limiting the maximum number of tasks assigned to a worker node.

A second challenge is to ensure that the distributed approximation of LAS preserves the performance benefits of the original formulation of LAS. Kairos addresses this issue by means of a novel task-to-node dispatching approach. In this approach, the central scheduler assigns tasks to nodes in a way such that the distribution of the runtime of tasks assigned to a particular worker node has high variance.

We have implemented Kairos in YARN. We compare its performance against the YARN FIFO scheduler and Big-C, an open-source state-of-the-art YARN-based scheduler that also uses preemption [5]. Compared to YARN FIFO, Kairos reduces the median job completion time by 73% and the 99th percentile by 30%. Compared to Big-C, the improvements are 37% for the median and 57% for the 99th percentile. We evaluate Kairos at scale by implementing it in the Eagle simulator and comparing its performance against Eagle [10]. Kairos improves the 99th percentile of short job completion times by up to 55% for the Google trace and 85% for the Yahoo trace.

Contributions.

1) We demonstrate good data center scheduling performance without using task runtime estimates.

2) We present an efficient distributed version of the LAS scheduling discipline.

3) We implement this distributed approximation of LAS in YARN, and compare its performance to state-of-the-art alternatives by measurement and simulation.

Roadmap. The outline of the rest of this paper is as follows. §2 provides the necessary background. §3 describes the design of Kairos. §4 describes its implementation in YARN. §5 evaluates the performance of the Kairos YARN implementation. §6 provides simulation results. §7 discusses related work. §8 concludes the paper.

2 BACKGROUND

2.1 Estimating task runtimes

Estimates in existing systems. Most state-of-the-art data center schedulers rely on task runtime estimates to make informed scheduling decisions [4, 10–12, 18–21, 25, 36]. Estimates are used to avoid head-of-line blocking and resource contention, provide load balancing and fairness, and meet deadlines. The accuracy of task runtime estimates is therefore of paramount importance. Estimates of the runtime of a task within a job can be obtained from past executions of the same task, if any, from past executions of similar tasks [4], or by means of on-line profiling [12]. A common estimation technique for the task duration is to take the average of the task durations over previous executions of the job [11, 30]. More sophisticated techniques rely on machine learning [29].

Challenges in obtaining accurate estimates. Unfortunately, obtaining accurate estimates is not easy due to several reasons. The scheduler may have little or no information to produce estimates for tasks of jobs that have never been submitted before [30]. Even if jobs are recurring, changes in the input data set may lead to significant and hard-to-predict shifts in task runtimes [1]. Changes in data placement may also cause the task execution time to change. Skew in the input data distribution can lead to tasks in the same job having radically different runtimes [8, 26]. Finally, failures and transient resource utilization spikes may lead to stragglers [2], which not

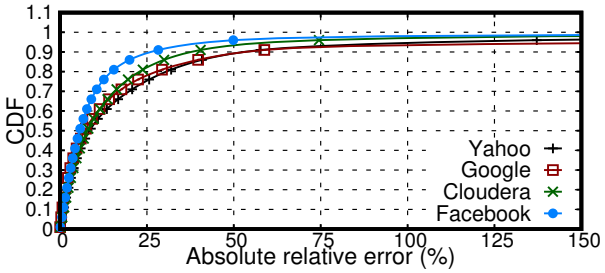


Figure 1: Prediction error for estimating the duration of each task in a job as the mean task duration in that job.

only have an unpredictable duration, but represent outliers in the data set used to predict future runtimes for tasks of the same job.

We provide an example of the estimation errors that can affect job scheduling decisions by studying the distribution of the error incurred when using the mean execution time of tasks in a job as an indicator of the execution time of a task in that job. We analyze four public traces that are widely used to evaluate data center schedulers. In particular, we consider the Cloudera [6], Facebook [6], Google [31] and Yahoo [7] traces. Let J be a job in the trace and T the set of tasks $t_1, \cdots, t_n$, in the job, each with an associated execution time $t_i.exectime$. Let T_J be the mean execution time of tasks in J. Then, we compute the prediction error for a task as $E = |100 \times (t_i.exectime - T_J)/T_J|$. We show the CDF of E in Figure 1. While up to 50% of the predictions are accurate to within 10%, some prediction errors are higher than 100%.

Similar degrees of misestimation have also been reported in recent work that uses a machine learning approach to predict task resource demands [29].

Coping with misestimations. Previous work has shown that task runtime misestimation leads to worse job completion times [10], and failure to meet service level objectives [12, 35] or job completion deadlines [35]. Some systems deal with misestimations by runtime correction mechanisms such as task cloning [2] and queue re-balancing [30], or by using a distribution of estimates rather than single-value estimates [29]. These solutions mitigate the effects of misestimations, but they do not avoid the problem entirely, and increase the complexity of the system.

Kairos overcomes the limitations of scheduling based on runtime estimates by adapting the LAS scheduling policy [27] to a data center environment. LAS does not require *a priori* information about task runtimes and is well suited to workloads with high variance in runtimes, as is the case in the often heavy-tailed data center workloads.

2.2 Least Attained Service

Prioritizing short jobs. Data centers workloads typically consist of a mix of long and short jobs [6, 7, 31]. Giving higher priority to short jobs improves their response times by reducing head-of-line-blocking. The Shortest Remaining Processing Time (SRPT) scheduling policy [33] prioritizes short tasks by executing pending tasks in increasing order of expected runtime and by preempting a task if a shorter task arrives. SRPT is provably optimal with respect to mean response time [32].

Recent systems have successfully adopted SRPT in the context of data center scheduling [10, 23, 30]. These systems do not support preemption, so they implement a variant of SRPT, where the shortest task is chosen for execution, but once a task is started, it runs to completion.

Least Attained Service (LAS). SRPT requires task runtime estimates to determine which task should be executed. LAS is a scheduling policy akin to SRPT, but it does not rely on *a priori* estimates [27]. LAS instead uses the service time already received by the task as an indication of the remaining runtime of the task.

Given a set of tasks to run, LAS schedules the so called *youngest* task for execution. The youngest task is the one with the lowest *attained service*, or, in other words, the one that has executed for the smallest amount of time so far. In case there are n youngest tasks with the same attained service, all of them are assigned an equal $1/n$ share of processing time, i.e., they run according to the Processor Sharing (PS) scheduling policy (as in typical multiprogramming operating systems). LAS makes use of preemption to allow the youngest task to execute at any moment.

Rationale. LAS uses the attained service as an indication of the remaining service demand of a task. This prediction works well with heavy-tailed service demand distributions. If a task has executed for a long time, it is likely that it is a large task, and therefore has a long way to go towards completion. Hence, it is better to execute younger tasks, that are more likely to be short tasks and therefore complete quickly.

In addition, a new incoming task is per definition the youngest task and executes immediately. Assuming a heavy-tailed runtime distribution, this new task is likely to be a short one. If no other task arrives during its execution and it completes in a time shorter than the attained service of any other waiting task, then it executes to completion without any preemption or queueing.

3 KAIROS

3.1 Design overview

Challenges of LAS in a data center. LAS is an appealing starting point to design a data center scheduler that does not require *a priori* task runtime estimates. In a strict implementation of LAS, however, the youngest task should be running at any moment in time. Then, adapting LAS to the data center scenario with a distributed set of worker nodes requires that a preempted task must be able to resume its execution on any worker node.

Allowing task migration across worker nodes incurs costs such as transferring input data or intermediate output of the task, and setting up the environment in which the task runs (e.g., a container). Determining whether or not to migrate a task is a challenging problem, especially in the absence of an estimate of the remaining runtime of the task. Therefore, Kairos does not strictly follow LAS, but rather implements an approximation thereof.

Note that long-running services that should never be preempted are out of the scope of Kairos scheduling.

Kairos approach to LAS. Kairos uses a two-level scheduling hierarchy consisting of a central scheduler and per-node schedulers on each worker node. We depict the high-level architecture of Kairos in Figure 2. The node schedulers implement LAS locally on each worker node (§3.2) and periodically send statistics to the central scheduler. The central scheduler assigns tasks to worker nodes so as to achieve load balance and to maximize the effectiveness of LAS at each worker (§3.3).

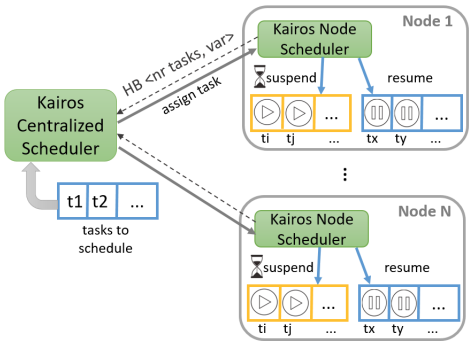


Figure 2: Kairos' two-level scheduling architecture. Node schedulers implement LAS locally. The central scheduler assigns tasks to nodes.

3.2 Node scheduler

Each worker node has N cores, which can run N concurrent tasks, and a queue, in which preempted tasks are placed. Algorithm 1 presents the data structures maintained by the node schedulers and the operations they perform. A `TaskEntry` structure maintains per task information such as its attained service time and, for running tasks, the start time of their current quantum. Each node scheduler implements LAS by taking as input the number of cores N and the quantum of time W.

When a new task arrives, it is immediately executed. If there is at least one core available, the task is assigned to that core (Line 8). Else, the task preempts the running task with the highest attained service time (Line 11). This task is moved to the node queue, and its attained service time is increased by the service time that it has received. When a task

Algorithm 1 Node scheduler

```
1:  Set<TaskEntry> IdleTasks, RunningTasks        ▷ Track suspended/running tasks
2:  upon event Task t arrives do
3:      TaskEntry t_e
4:      t_e.task ← t
5:      t_e.attained ← 0
6:      t_e.start ← now()
7:      RunningTasks.add(t_e)
8:      if (IdleCores.size() > 0) then            ▷ Free core can execute t
9:          core c = idleCores.pop()
10:     else                                      ▷ Preempt oldest running task
11:         t_p ← argmax_{t_t.attained} {t_t ∈ RunningTasks}
12:         t_p.attained+ = now() − t_p.start
13:         c ← core serving t
14:         remove t_p from c
15:         IdleTasks.add(t_p)
16:     assign t to c
17:     c.startTimer(W)
18:     start t
19: upon event Task t finishes on core c do
20:     RunningTasks.remove(t)
21:     if (!IdleTasks.isEmpty() then             ▷ Run youngest suspended task
22:         TaskEntry t_r ← argmin_{t_i.attained} {t_i ∈ IdleTasks}
23:         RunningTasks.add(t_r)
24:         assign t_r to c
25:         t_r.start ← now()
26:         c.startTimer(W)
27:         start t_r.task
28:     else
29:         IdleCores.push(c)
30: upon event Timer fires on core c running task t do
31:     TaskEntry t_s ←TaskEntry e : e.task = t
32:     t_s.attained+ = now() − t_s.start
        ▷ Find youngest suspended task
33:     TaskEntry t_m ← argmin_{t_i.attained} {t_i ∈ IdleTasks}
34:     if (t_m.attained ≤ t_s.attained) then     ▷ Preempt t
35:         IdleTasks.remove(t_m)
36:         IdleTasks.add(t_s)
37:         RunningTasks.remove(t_s)
38:         RunningTasks.add(t_m)
39:         t_m.start ← now()
40:         place t_m.task on c
41:         start t_m.task
42:     else                                      ▷ Continue running t
43:         t_s.start ← now()
44:     c.startTimer(W)
45: upon event Every Δ do
46:     Heartbeat HB
47:     HB.num ← IdleTasks.size() + RunningTasks.size()
48:     HB.var ← var_{t.attained} {t ∈ IdleTasks ∪ RunningTasks}
49:     send HB to the central scheduler
```

terminates, if the node queue is not empty, the task with the lowest attained service is scheduled for execution (Line 21).

When a task t is assigned to a core, a timer is set to expire after W seconds (Line 17). If t has not completed by the time the timer fires (Line 27), the scheduler increases the attained service time of the task by W. Let T be the updated value of the attained service time for task t. If there is a task t' in the node queue with attained service time lower than T, t' is scheduled for execution by preempting t (Line 31). Otherwise, t continues its execution, and the timer is reset (Line 39).

Periodically, the node scheduler sends to the central scheduler the number of tasks currently assigned to it, and the variance in the service times already attained by the tasks (Line 42). The latter information is used by the central scheduler in deciding where to send a task, as explained in §3.3.3.

The node scheduler implements a starvation prevention mechanism (not shown in the pseudocode) to guarantee that all tasks get scheduled eventually. If a task is not able to run for a given number of consecutive quanta, then Kairos guarantees that the task gets to run for at least a given amount of time (a multiple of W), during which it cannot be preempted. This mechanism ensures the progress of every task.

Impact and setting of W. The value of W determines the trade-off between task waiting times and completion times. A high value for W allows the shortest tasks to complete within a single quantum. However, it may also lead a preempted task in the node queue to wait for a long time before it can run again, an undesirable situation for a short task that has been preempted to make room for a new incoming task. A low value for W, instead, gives a task frequent opportunities to execute and hence potentially complete. However, it may also lead to longer completion times because of frequent task interleaving. We study the sensitivity of Kairos to the setting of W in §5.3.3, where we show that Kairos is relatively robust to sub-optimal settings of W.

3.3 Central scheduler

Algorithm 2 presents the data structures maintained by the central scheduler and its operations.

3.3.1 Challenges in the absence of estimates. The lack of *a priori* task runtime estimates makes it cumbersome to achieve load balancing.

Existing approaches use task runtime estimates to place a task on the worker node that is expected to minimize the waiting time of the task [4, 30]. This strategy improves task completion times and achieves high resource utilization by equalizing the load on the worker nodes. Kairos cannot re-use such existing techniques in a straightforward fashion, because it cannot accurately estimate the backlog on a worker node and the additional load posed by a task being scheduled.

Algorithm 2 Central scheduler

```
1:  Queue CentralQueue                    ▷ Queue where incoming tasks are placed
2:  Node[numNodes] Nodes                  ▷ Entries track # tasks and attained service times

3:  upon event New job J arrives do
4:      for task t ∈ J do
5:          Queue.push(t)

6:  upon event Heartbeat HB from Node i arrives do
7:      Nodes[i].var ← HB.var
8:      Nodes[i].numTasks ← HB.numTasks

9:  procedure MAINLOOP
10:     while (true) do
11:         for i = 0, ..., N + Q do
12:             S_i ← {Node m ∈ Nodes : m.numTasks = i}
13:             while (!S_i.isEmpty() ∧!CentralQueue.isEmpty()) do
14:                 Node m ← argmin_{n.var}{n ∈ S_i}
15:                 Task t ← CentralQueue.pop()
16:                 Assign t to m
17:                 S_i ← S_i \ {m}
18:             Sleep(Δ)
```

To circumvent this problem, Kairos decouples the problems of achieving load balance and high resource utilization from the problem of achieving low completion times. Kairos leverages the insight that short completion times are already achieved by implementing LAS in the individual node schedulers. In fact, LAS gives shorter tasks the possibility to completely or partially bypass the queues on the worker nodes. As a result, the central scheduler can to some extent be agnostic of the actual backlog on worker nodes, because the backlog is not an indicator of the waiting time for a task.

Hence, in Kairos, the central scheduler has two goals:

1) Achieve high resource utilization and load balance, by reducing the probability that cores are idle while tasks are waiting in some queue, either the central queue or any of the worker queues (§3.3.2).

2) Maximize LAS effectiveness, e.g., by improving the proability that short tasks can bypass long tasks and by reducing the probability that tasks hurt each other's response times by an excessive number of task switches (§3.3.3).

3.3.2 Load balancing. The central scheduler aims to balance the load across worker nodes by assigning to each of them an equal *number* of tasks. Hence, the first outstanding task in the central queue is placed on the worker node with the smallest number of assigned tasks.

This policy alone, however, is not sufficient with heavy-tailed runtime distributions, as it may lead to temporary load imbalance scenarios. For example, a worker node may be assigned many short tasks, while another worker node is loaded with longer tasks. Then, the first worker node might complete all of its short tasks and become idle, while some tasks are waiting on the other worker node.

To address this issue, the central scheduler assigns to each worker at most $Q + N$ tasks at any moment in time. This

admission control mechanism bounds the amount of load imbalance possible, since a worker node can host at most Q idle tasks that could have been assigned to other worker nodes with available resources.

Impact and setting of Q. The value of Q determines the trade-off between load balance and effectiveness of LAS. A small value of Q reduces the possibility of load imbalance, but may lead to many short tasks being delayed in the central queue. A high value of Q, on the contrary, may lead to higher load imbalance, but enables more parallelism, benefiting short tasks that can complete quickly by preempting other tasks.

We assess the sensitivity of Kairos to the setting of Q in §5.3.3, where we show that Kairos' performance is not dramatically affected by sub-optimal settings of Q.

3.3.3 Maximizing LAS effectiveness. Kairos implements an LAS-aware policy to break ties in cases in which two or more worker nodes have an equal number of tasks assigned to them. In more detail, it assigns the task to the worker node with the lowest variance in the attained service times of tasks currently placed on that worker node. The hope is that by doing so it can significantly increase the variance of the runtimes of tasks assigned to that node. The rationale behind this choice is that LAS is most effective when the task duration distribution has a high variance. Intuitively, if only short tasks were assigned to a node, the youngest short tasks would preempt older short tasks, hurting their completion times. Similarly, if only long tasks were assigned to a node, all would run in an interleaved fashion, each one hurting the completion time of the others.

The effectiveness of this policy is grounded in previous analysis of SRPT in distributed environments, that shows that maximizing the heterogeneity of task runtimes on each worker node is key to improve task completion times [3, 13]. Unlike previous studies, however, Kairos does not rely on exact knowledge of task runtimes for each worker node, and uses the attained service times of tasks on a worker node to estimate the variability in task runtimes on that worker node.

4 IMPLEMENTATION

We implement Kairos in YARN [17], a widely used scheduler for data-parallel jobs. Figure 3 shows the main building blocks of YARN, their interactions and the components introduced by Kairos.

YARN. YARN consists of a `ResourceManager` residing on a master node, and a `NodeManager` residing on each worker node. YARN runs a task on a worker node within a container, which specifies the node resources allocated to the task. Each worker node also has a `ContainerManager` that manages the containers on the node. Finally, each job has an `ApplicationManager` that runs on a worker node and tracks the advancement of all tasks within the job.

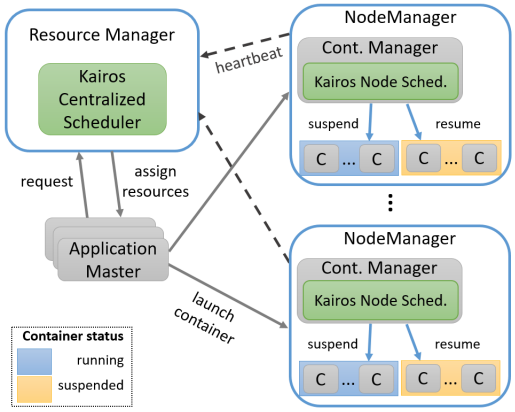


Figure 3: Kairos integration in YARN.

The `ResourceManager` assigns tasks to worker nodes and communicates with the `NodeManagers` on the worker nodes. A `NodeManager` sends periodic heartbeat messages to the `ResourceManager`. The heartbeats describe the node's health and the containers running on it.

Kairos central scheduler. The Kairos central scheduling policy is implemented in the `ResourceManager`. In particular, the Kairos central scheduler extends the CapacityScheduler, to allow more containers to be allocated to a node than there are cores on that mode.

Kairos node scheduler. The `KairosNodeScheduler` is implemented within the `ContainerManager`. It consists of a thread that monitors the status of the containers and implements LAS. The `KairosNodeScheduler` maintains the attained service time of the tasks running within the containers, and implements preemption. It preempts a container by reducing the resources allocated to it to a minimum, and resumes it by restoring the original allocation, similar to what is done in Chen et al. [5]. Reducing the resources to a minimum (rather than to zero) allows the heartbeat mechanism to continue to function correctly when a container is preempted. The `KairosNodeScheduler` sets the timers necessary for implementing the processor sharing window W, and also extends the information sent by the `NodeManager` in the heartbeat messages, by including the standard deviation of the attained service of all containers hosted by the node. In this implementation (not in the general approach) preempted containers still consume memory, Kairos assumes that memory is not a limiting factor.

We have made the Kairos code publicly available. [1]

[1] https://github.com/epfl-labos/Kairos

Category	input	#maps	#reduces	extraFlops	duration	probability
1 small	4GB	15	15	0	85s	0.32
2 medium small	4GB	15	15	500	201s	0.31
3 medium	8GB	30	30	0	239s	0.31
4 medium long	30GB	112	60	500	308s	0.04
5 long	60GB	224	60	1000	1175s	0.02

Table 1: Job categories composing our workload. Job runtimes follow a heavy-tailed distribution, typical for modern data center workloads.

5 EXPERIMENTAL EVALUATION

5.1 Methodology and baselines

We compare Kairos to the FIFO YARN scheduler and to Big-C, a recent preemptive scheduler based on YARN and for which the source code is available [5].

Big-C uses available runtime estimates to perform task placements, and preemption to prioritize short tasks in case of high utilization. Big-C extends YARN's capacity scheduler. Jobs are partitioned in classes, and each class is assigned a priority and a number of nodes. Tasks can run opportunistically on nodes assigned to a different class, but when a task with a certain priority is ready to run and there are no free nodes in its class, tasks with lower priority are preempted.

Big-C defines two job classes, corresponding to long and short jobs. A job is classified according to available runtime estimates. Short jobs have higher priority and are assigned a large fraction of the nodes. Long jobs running opportunistically on a node assigned to the short job class can be preempted by newly arrived short tasks. We configure Big-C with its default value for the share of resources for short jobs (95%).

5.2 Testbed

Platform. We use a 30-node cluster running over a 10Gb Ethernet. Each node runs 2.6Ghz AMD Opteron 6212 CPUs. We limit the scheduler to 4 CPUs, resulting in 120 cores cluster-wide. We use Hadoop-2.7.1, the same code base as Big-C. Containers use Docker-1.12.1 with the image from `sequenceiq/hadoop-docker`.

We set $Q = 4$, bounding the maximum number of tasks queued per node to the number of cores on a node, and $W = 50s$, which allows the shortest tasks to execute within one quantum of time.

Workloads. We create workloads with specific distributions of job runtimes. In particular, we use Hadoop WordCount jobs, using different input sizes, and with each input consisting of randomly generated 100-character strings. We modify the Hadoop WordCount code so that we can increase a task's runtime by a controllable amount, by inserting a parameterized number of floating point operations in both the map and reduce functions.

The resulting workloads then consist of five categories of jobs, described in Table 1. The number of mappers in each category is equal to the job input size divided by the HDFS block size. The number of reducers is chsoen for optimal performance. We allocate 2GB for map tasks and 4GB for reduce tasks. The HDFS block size used is 256MB for categories 1 to 4, and 1GB for category 5. The HDFS replication factor is 3. The container size is set to <5120 MB,1 vCore>. The durations in Table 1 correspond to the total makespan of a job when running alone in the cluster. When using Big-C, jobs from the first three categories are considered short jobs, and jobs from the remaining two categories as long jobs.

For each experiment we draw 100 jobs from these five categories, according to the probabilities given in Table 1. The probabilities are inspired by the typical heavy-tailed job runtime distribution that characterizes production workloads. The job inter-arrival times follow a Poisson distribution with a mean of 60s. The resulting workload takes roughly 2 hours to run.

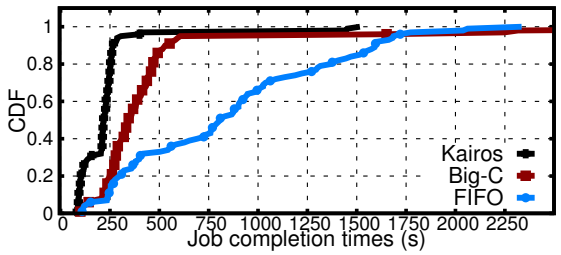
5.3 Results

5.3.1 Job completion times. Figure 4a reports the CDF of job completion times with Kairos, YARN/FIFO and Big-C. Figure 4b shows the CDF of job slowdowns for the same three systems.

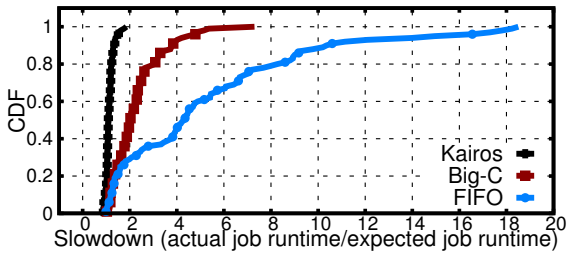
Kairos achieves better job completion times than Big-C and FIFO at all percentiles. Short tasks in Kairos complete more quickly than in Big-C, which can be seen by looking at the lower percentiles. For example, Kairos reduces the 50th percentile of job completion times by 73% with respect to YARN FIFO (241s vs. 808s) and by 37% with respect to Big-C (217s vs. 341s).

The reason for this improvements is that worker nodes in Kairos accept Q more tasks than what they can process, allowing short tasks to be placed on a busy node and execute immediately thanks to LAS. Instead, in Big-C a short task t_s cannot preempt another short task t'_s, even if t_s is shorter than t'_s. Hence, t_s has to wait for some node to have free resources before starting.

Kairos is also more effective in achieving low completion times for longer jobs, which is visible at the right end of the CDF. Kairos reduces the 99th percentile of job completion times by 30% with respect to FIFO (1452s vs. 2061s) and by 57% with respect to Big-C (1452s vs. 3368s). Kairos achieves better job completion times at the high percentiles by not restricting the share of resources for long jobs, and by enhancing the effectiveness of LAS by its task-to-node assignment policy. Big-C achieves worse tail latency than FIFO, because

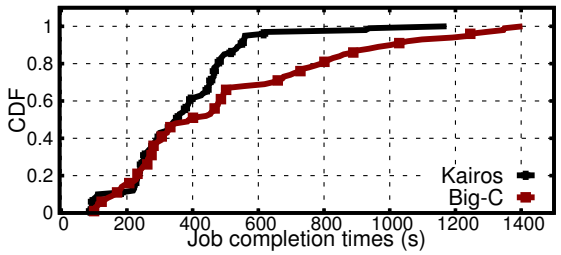


(a) CDF of job completion times.

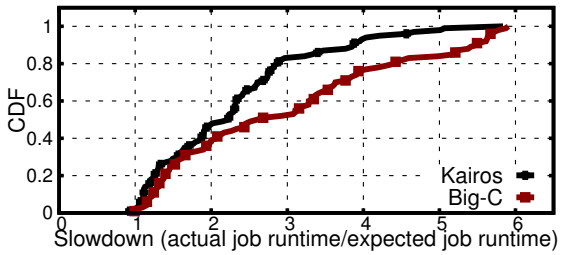


(b) Slowdown.

Figure 4: CDFs of job completion times and slowdown in Kairos, Big-C and YARN/FIFO. Heavy tailed workload. Tail of Big-C omitted for visibility in (a). Worst tail job completion time for Big-C in (a) is 3624s.



(a) CDF of job completion times.



(b) Slowdown.

Figure 5: CDFs of job completion times and slowdown in Kairos, Big-C. Uniform workload.

long jobs can be delayed due to frequent preemptions and their low priority in Big-C.

As illustrated in Figure 4b, the job slowdown is lower in Kairos for all jobs. Moreover, all jobs in Kairos are slowed down by a comparable amount – a desirable property, because it provides performance predictability and a degree of fairness. In contrast, in YARN FIFO and Big-C some jobs are slowed down much more than others. Even worse, some of the largest slowdowns in FIFO and Big-C (5.14x and 18.23x, respectively) occur for the shortest jobs in the workload.

We also create a uniform workload to test how Kairos behaves in a setting that is not ideal for LAS. We show the results in Figure 5. For this workload we only use the first 3 categories in Table 1 with probabilities 0.35, 0.35 and 0.3. The job inter-arrival times follow a Poisson distribution with a mean of 30s. Big-C is configured to assign a 70% share to categories 1 and 2, and 30% for category 3. Since the ratio short/long in Big-C is workload dependent, we did our best to adjust it based on the types of jobs in the uniform workload. As expected, the job runtimes and the slowdown for Kairos deteriorate compared to the heavy-tailed scenario, but they are still better than Big-C.

5.3.2 LAS-aware task dispatching. Figure 6 reports the CDF of job completion times in Kairos with different policies used to choose where to place a task, when there are multiple worker nodes with the same number of tasks already assigned. Besides the LAS-aware policy describe in §3.3.3 and denoted by `Var` in Figure 6, we implement two additional policies, `Random` and `Sum`. `Random` assigns the task to a randomly chosen node. The `Sum` policy assigns a task to the node whose tasks have the lowest cumulative attained service time. The rationale is that by using attained service time as an estimation of remaining runtime, the `Sum` policy tries to assign a task to the least loaded node.

Figure 6 shows that the `Var` policy delivers better job completion times at all percentiles. The biggest gains over `Random` and `Sum` are around the 30th percentile and towards the tail of the distribution. The benefit at the 30th percentile indicates that the shortest jobs, which account for 30% of the total (see Table 1), are effectively prioritized. The benefit at higher percentiles shows that `Var` is also able to effectively use LAS to improve the response time of larger jobs as well.

5.3.3 Sensitivity analysis. We now show that Kairos maintains performance better than or comparable to Big-C even for sub-optimal settings of the parameters W and Q. To

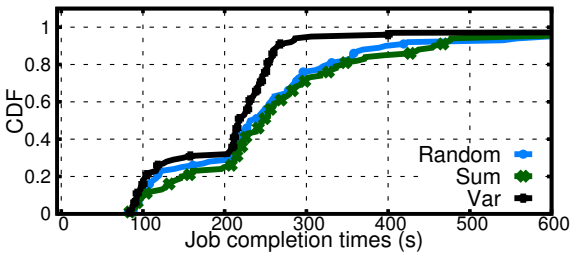


Figure 6: Comparison of alternative task-to-node dispatching policies in Kairos. Tail omitted for visibility. Worst tail job completion times are 1633s for Random, 1651s for Sum and 1452s for Var.

this end, we study how the performance of Kairos varies with different settings for W and Q. When studying the sensitivity of Kairos to the setting of one parameter, we keep the other one to its default value.

Sensitivity to Q. Figure 7a shows the CDF of job response times in Kairos with $Q = 2, 4, 8$ and in Big-C. $Q = 8$ and $Q = 2$ perform slightly worse than the default value of $Q = 4$ we use in Kairos, but still deliver better performance than Big-C at each percentile.

The shape of the CDFs for different values of Q matches our analysis of §3.3. If Q is too low, then sometimes short tasks are kept in the central queue, thus preventing them from going to the nodes, where they could execute rightaway by virtue of LAS. This effect is visible at the 20th percentile of the CDF, where $Q = 2$ is worse than $Q = 8$. If Q is too high, instead, short tasks more often preempt longer ones, increasing their response times and thus leading to worse tail latencies.

Sensitivity to W. Figure 7b shows the CDF of job response times in Kairos with $W = 10, 50, 100$ and in Big-C. Similar to what is seen for Q, setting W too high or too low has some negative impact on the performance of Kairos, but Kairos maintains its performance lead over Big-C.

Comparing the performance achieved with the $W = 10$ and $W = 100$ we see that too low a value for W has the effect that the execution of tasks on a worker node is much interleaved. This phenomenon penalizes the longest jobs, i.e., the very tail of the completion times distribution, but leads to better values for lower percentiles. The dual holds for $W = 100$. The longest jobs can use big quanta, improving their completion times at the detriment of shorter jobs.

Trace	Total # jobs	% Long jobs	% Task-Seconds long jobs
Yahoo [7]	24262	9.41	98
Google [31]	506460	10.00	83

Table 2: Job heterogeneity in the traces. *% Task-seconds long jobs* is the sum of the execution times of all long tasks divided by the sum of the execution times of all tasks.

6 SIMULATION

6.1 Methodology and baseline

We evaluate Kairos in larger data centers by means of a simulation study using the popular Yahoo [7] and Google [31] traces. We compare Kairos to Eagle [10], the most recent system whose design is implemented in a simulator.[2] We have integrated the Kairos design in the Eagle simulator, and we have made the simulation code publicly available. [3] We report average values of 10 runs for the Yahoo trace, and 5 runs for the Google trace.

Background on Eagle. Eagle partitions the set of worker nodes in two sub-clusters, one for long jobs and one for short jobs. The nodes of the data center are divided between the two sub-clusters proportionally to the expected load posed by short and long jobs. Hence, in the traces we consider, the majority of the resources is assigned to long jobs, as they consume the bulk of the resources. Short tasks are allowed to opportunistically use idle nodes in the partition for long jobs. By this workload partition technique, Eagle avoids head-of-line-blocking altogether. In addition, short jobs are executed according to a distributed approximation of SRPT that does not use preemption. In other words, Eagle aims to first execute the tasks of shorter jobs, but tasks cannot be suspended once they start. Eagle uses task runtime estimates to classify jobs as long or short, and to implement the SRPT policy.

We configure Eagle with the same parameters as in its original implementation (which vary depending on the target workload trace). These parameters include sub-cluster sizes, cutoffs to distinguish short jobs from long ones and parameters to implement SRPT.

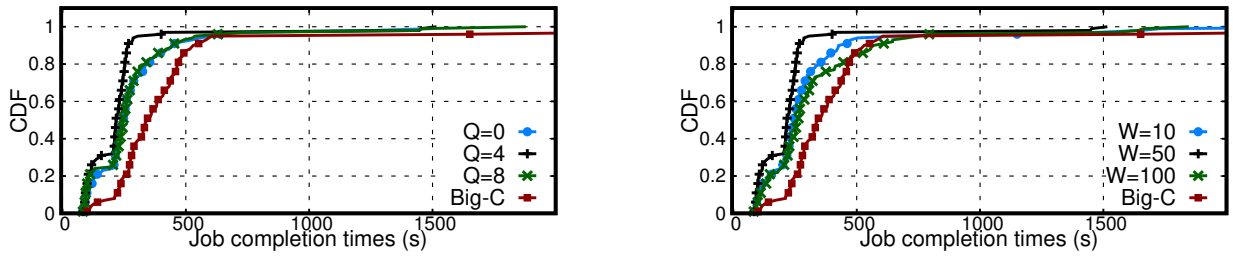
6.2 Simulated testbed

Platform. We simulate data centers with 15,000 to 23,000 worker nodes using the Google trace, and with 4,000 to 8,000 nodes using the Yahoo trace. We keep the job arrival rates constant at the values in the traces, so increasing the number of worker nodes reduces the load. We set the network delay to 0.5 milliseconds, and we do not assign any cost to making scheduling decisions.

Workloads. Table 2 shows the total number of jobs, the percentage of long jobs and the percentage of task-seconds for long jobs for the two traces. The percentage of the execution times (task-seconds) of all short jobs is 17% in the Google

[2] We do not compare our prototype of Kairos with Eagle because Eagle is built on top of Spark's scheduler.

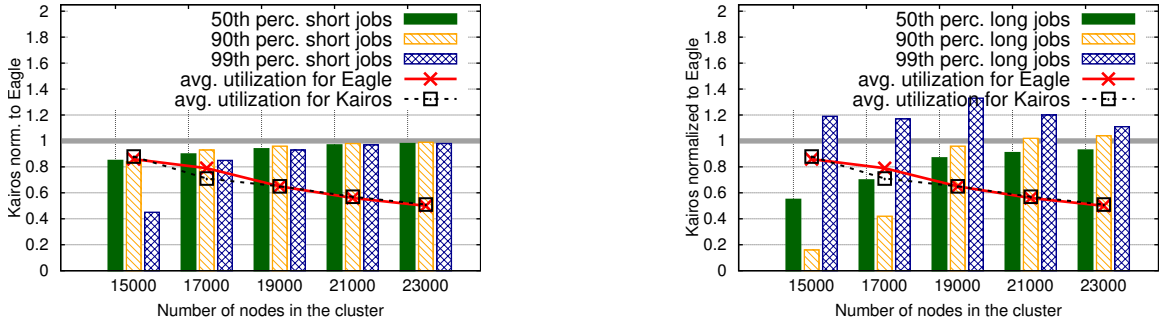
[3] https://github.com/epfl-labos/kairos



(a) Varying Q (max number of tasks queued per node).

(b) Varying W (size of quantum of time).

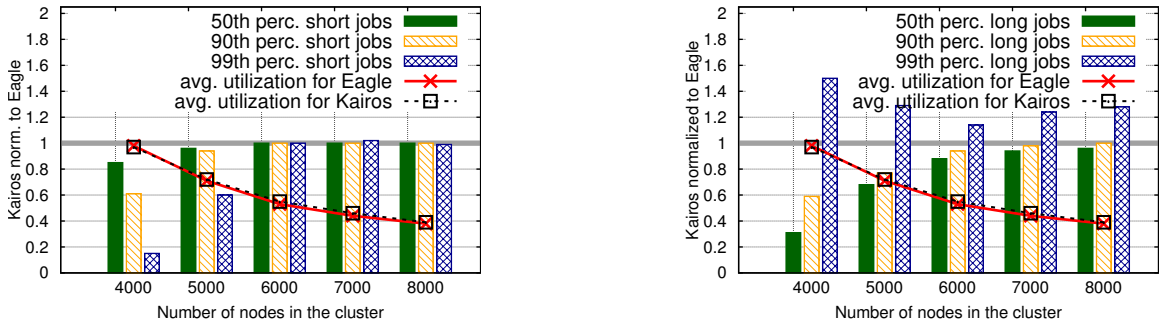
Figure 7: Sensitivity analysis to parameters Q (queue size) and W (time quantum).



(a) Kairos short jobs normal. to Eagle. Google trace.

(b) Kairos long jobs normal. to Eagle. Google trace.

Figure 8: Kairos normalized to Eagle short (a) and long (b) jobs. Google trace.



(a) Kairos short jobs normal. to Eagle. Yahoo trace.

(b) Kairos long jobs normal. to Eagle. Yahoo trace.

Figure 9: Kairos normalized to Eagle short (a) and long (b) jobs. Yahoo trace.

trace and 2% for the Yahoo trace. These values determine the size of the partitions for short jobs in Eagle.

Each simulated worker node has one core. Kairos uses $Q = 2$ for both workloads, $W = 100$ time units for the Yahoo trace and $W = 10,000$ time units for the Google trace. The starvation prevention counter for Kairos is set to 3 for both traces.

6.3 Results

Figures 8 and 9 report the 50th, 90th and 99th percentiles of job completion times for Kairos normalized to Eagle for the Yahoo and Google traces, respectively. The plots on the left (right) report short (long) job completion times. We also report the average cluster utilization for Kairos and Eagle as a function of the number of worker nodes in the cluster.

Figures 8a and 9a show that Kairos improves the short job completion times significantly at high loads (up to 55% for the Google trace and 85% for the Yahoo trace). When the load is very high, short jobs in Eagle are confined to the sub-cluster reserved from them. Hence, short jobs compete for the same scarce resources. In Kairos, instead, short jobs can run on any node, and preempt long jobs to achieve fast completion times. As the load decreases, the two systems achieve similar performance.

Long jobs exhibit different dynamics. Looking at the 50th and 90th percentiles in Figures 8(b) and Figure 9(b), we see that, when the load is at least 50%, Kairos reduces the completion times of most of the long jobs with respect to Eagle. Kairos interleaves the execution of long jobs, leading to better completion times for the shorter among the long jobs. In Eagle, instead, the absence of preemption may cause a relatively short task among the long ones to wait for the entire execution of a longer task to complete. Similar to what is seen for short jobs, the performance differences between the two systems at the 50th and the 99th percentile level off as the load decreases.

Kairos achieves a worse 99th percentile than Eagle (between 14% and 50% in the Yahoo trace and between 11% and 33% for the Google trace), due to the fact that Kairos frequently preempts the longest jobs to prioritize the shorter ones. This tradeoff is unavoidable, and in our opinion the right one. Kairos improves the performance for the vast majority of the jobs, especially latency-sensitive ones. The price to pay for that is slightly worse performance for the longest jobs.

Finally, Kairos and Eagle achieve the same resource utilization in both workloads and for all cluster sizes. This result showcases Kairos's ability to achieve the same high resource utilization as approaches that rely on prior knowledge of task runtimes.

7 RELATED WORK

We compare Kairos to existing systems first focusing on scheduling policy, and then on scheduler architecture.

7.1 Scheduling policies

7.1.1 Scheduling with runtime estimates. Most state-of-the-art scheduling systems rely on runtime estimates to make informed scheduling decisions. These systems differ in *how* such estimates are integrated into the scheduling policy.

Apollo [4], Yaq [30] and Mercury [25] disseminate information about the expected backlog on worker nodes. Tasks are scheduled to minimize expected queueing delay and to equalize the load. Yaq also uses per-task runtime estimates to implement queue reordering strategies, aimed at prioritizing short tasks.

Hawk [11], Eagle [10] and Big-C [5] use runtime estimates to classify jobs as long or short. In Eagle and Hawk, the set of worker nodes is partitioned in two subsets, sized proportionally to the expected load in each class. Then, tasks of a job are sent to either of the two sub-clusters depending on their expected runtime. Big-C gives priority to short jobs by assigning a higher priority to them in the YARN capacity scheduler. Workload partitioning and short job prioritization aim to reduce [5, 11] or eliminate [10] head-of-line-blocking.

Tetrisched [35], Rayon [9], Firmament [18], Quincy [24], Tetris [19], 3Sigma [29] and Medea [15] formalize the scheduling decision as a combinatorial optimization problem. The resulting Mixed-Integer Linear Program is solved either exactly, or an approximation is computed by means of heuristics.

Jockey [14] uses a simulator to speculate on the evolution of the system and accordingly decides the task-to-node placement. Graphene [21] uses estimates to decide first the placement of the job with the most complex requirements, and then packs other jobs depending on the remaining available resources. Carabyne [20] temporarily relaxes fairness guarantees to allow jobs to use resources destined to others.

As opposed to these systems, Kairos eschews the need for any *a priori* information about task runtimes. Instead, Kairos infers the expected remaining runtime of tasks from the amount of time they have already executed, and uses preemption and a novel task-to-node assignment policy to avoid head-of-line blocking and achieve high resource utilization.

Correction mechanisms. The systems that rely on task runtime estimates also encompass several techniques to cope with unavoidable misestimations.

Tetrisched [35], 3Sigma [29], Rayon [9] and Jockey [14] periodically re-evaluate the scheduling plan in case tasks take longer than expected to complete.

By contrast, Kairos uses preemption and limits the amount of queue imbalance by means of admission control. Kairos can integrate speculative execution or queue re-balancing techniques techniques at the cost of introducing heuristics to detect stragglers (e.g., based on their progress rate) and support for task migration (e.g., based on checkpointing).

Some systems like Rayon [9], 3Sigma [29] and Big-C [5] use preemption to correct the scheduling decision in case a new job arrives that must use resources already allocated. The difference with the use of preemption in Kairos is twofold. First, Kairos uses preemption to *avoid* the need for runtime estimates, which makes Kairos suitable also for environments with highly variable runtimes across several executions of the same job or where data on previous runs of the jobs is not available. Second, preemption in Kairos, in addition to allowing short tasks to get served quickly, also allows longer tasks to take turns to execute, thereby ensuring progress.

7.1.2 Scheduling without runtime estimates. Sparrow [28] avoids the use of runtime estimates by means of *batch sampling*. A job with t tasks sends $2t$ *probes* to $2t$ worker nodes, where the probes are enqueued. One task of the job is served when one of the probes reaches the head of its queue. Sparrow improves response times because the t tasks in a job are executed by the least loaded t worker nodes out of the $2t$ contacted. We have not compared Kairos to Sparrow directly, because several other schedulers significantly outperform Sparrow. For example, Hawk [11] improves the 50th and 90th percentile job runtimes by 80% and 90% for short jobs compared to Sparrow. Also for short jobs, Eagle improves on Hawk between 30% and 90% for all percentiles. We therefore compare Kairos against Eagle.

Tyrex [16] aims to avoid head-of-line blocking by partitioning the workload in classes depending on task runtimes, and by assigning different classes to disjoint partitions of worker nodes. Because runtimes are not known *a priori*, workload partitioning is achieved by initially assigning all tasks to partition 1, and then migrating a task from partition i to $i+1$ when the task execution time has exceeded a threshold t_i.

Hu et al. [22] aim to prioritize short jobs by organizing jobs in priority queues depending on the cumulative time its tasks have received so far. Jobs in higher-priority queues are assigned more resources than those in lower-priority queues. Tasks are hosted in a system-wide queue on a centralized scheduler, and are assigned to worker nodes depending on the priority of the corresponding job.

Unlike Kairos, in all these systems there is no support for preemption, and tasks, once started, run to completion. Hence, latency-sensitive tasks may incur head-of-line blocking and suffer from high waiting times in case of high utilization. In contrast, Kairos uses preemption to allow an incoming task to run as soon as it arrives on a worker node, offering short tasks the possibility of completing with limited or no waiting time, even in high-utilization scenarios.

7.2 Scheduler architecture

Kairos can be classified as a centralized scheduler, because all tasks are dispatched by a single component, although the worker nodes also perform local scheduling decisions. There is a recent trend towards distributed schedulers, such as Omega [34], Sparrow [28], Apollo [4] and Yaq [30], or hybrid schedulers such as Mercury [25], Hawk [11] and Eagle [10] to achieve low scheduling latency under high job arrival rates.

Kairos can sustain high load and achieve low scheduling latency despite being centralized, because *i*) it effectively distributes the burden of performing scheduling decisions between the central scheduler and the worker nodes and *ii*) the task-to-node assignment policy is very lightweight.

Because of these characteristics, we argue that Kairos could also be implemented as a distributed scheduler. The state of the worker nodes could be gossiped across the system, e.g., as in Apollo [4] and Yaq [30], or shared among the distributed schedulers, e.g., as in Omega [34]. Existing techniques like randomly perturbing the state communicated to different schedulers [4] and atomic transactions over the shared view of the cluster [34] could be used to limit or avoid concurrent conflicting scheduling decisions by different schedulers.

8 CONCLUSIONS AND FUTURE WORK

In this paper we present Kairos, a new data center scheduler that makes no use of *a priori* task runtime estimates. Kairos achieves low latency and high resource utilization, by employing in synergy two techniques. First, it uses a lightweight form of preemption to prioritize short tasks over long ones and to avoid head-of-line-blocking. Second, it employs a novel task-to-node assignment that reduces load imbalance among worker nodes and assigns tasks to nodes so as to improve the chances that they complete quickly.

We evaluate Kairos experimentally on a small scale using a full-fledged prototype in YARN, and on a larger scale by means of simulation. We show that Kairos achieves better job completion times than state-of-the-art approaches that use *a priori* task runtime estimates.

As part of future work we plan to extend Kairos to make it more aware of other container resources. Currently, Kairos assumes a slot-based allocation system where each container occupies a slot, defined in terms of a fixed number of cores. In future work, we first plan to make Kairos memory-aware. Currently, Kairos assumes that the CPU is the bottleneck resource in the cluster and that memory is not a limiting factor. Second, we plan to make Kairos handle heterogeneous CPU allocations, where different containers can be allocated different number of cores.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd Kenneth Yocum for their feedback. We also thank Baptiste Lepers, Calin Iorgulescu, Kristina Spirovska, and Konstantinos Karanasos for the discussions, feedback and help. Additional thanks go to the Big-C authors for making their code available. This research has been supported in part by a grant from Microsoft Research Cambridge and by an EcoCloud post-doctoral research fellowship.

REFERENCES

[1] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 469–482, Berkeley, CA, USA, 2017. USENIX Association.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[3] N. Avrahami and Y. Azar. Minimizing total flow time and total completion time with immediate dispatching. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'03, pages 11–18, New York, NY, USA, 2003. ACM.

[4] 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 *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 285–300, Broomfield, CO, Oct. 2014. USENIX Association.

[5] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'17, pages 251–263, Santa Clara, CA, 2017.

[6] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, Aug. 2012.

[7] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS'11, pages 390–399, Washington, DC, USA, 2011. IEEE Computer Society.

[8] E. Coppa and I. Finocchi. On data skewness, stragglers, and mapreduce progress indicators. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, SoCC'15, pages 139–152, New York, NY, USA, 2015. ACM.

[9] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'14, pages 2:1–2:14, New York, NY, USA, 2014. ACM.

[10] 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*, number EPFL-CONF-221125, 2016.

[11] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'15, pages 499–510. USENIX Association, July 2015.

[12] 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*, ASPLOS'14, pages 127–144, New York, NY, USA, 2014. ACM.

[13] D. G. Down and R. Wu. Multi-layered round robin routing for parallel servers. *Queueing Systems*, 53(4):177–188, Aug. 2006.

[14] 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*, EuroSys'12, pages 99–112, New York, NY, USA, 2012. ACM.

[15] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the 13th European Conference on Computer Systems*, EuroSys'18, pages 4:1–4:13, 2018.

[16] B. Ghit and D. H. J. Epema. Tyrex: Size-based resource allocation in mapreduce frameworks. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*, pages 11–20, 2016.

[17] 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 Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[18] I. Gog, M. Schwarzkopf, A. Gleave, R. M. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proc. of OSDI*, 2016.

[19] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, Aug. 2014.

[20] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 65–80, Berkeley, CA, USA, 2016. USENIX Association.

[21] 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 Conference on Operating Systems Design and Implementation*, OSDI'16, pages 81–97, Savannah, GA, 2016. USENIX Association.

[22] Z. Hu, B. Li, Z. Qin, and R. S. M. Goh. Job scheduling without prior information in big data processing systems. In *Proceedings of ICDCS*, 2017.

[23] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *Proceedings of the 6th Symposium on Cloud Computing*, SoCC'15, pages 111–124, New York, NY, USA, 2015. ACM.

[24] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP'09, pages 261–276, 2009.

[25] 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 Conference on Usenix Annual Technical Conference*, USENIX ATC'15, pages 485–497, Berkeley, CA, USA, 2015. USENIX Association.

[26] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. Skewtune: mitigating skew in mapreduce applications. In K. S. Candan, Y. C. 0001, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *SIGMOD Conference*, pages 25–36. ACM, 2012.

[27] M. Nuyens and A. Wierman. The foreground-background queue: A survey. *Performance evaluation*, 65(3-4):286–307, Mar. 2008.

[28] 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*, SOSP'13, pages 69–84, New York, NY, USA, 2013. ACM.

[29] 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 13th European Conference on Computer Systems*, EuroSys'18, page 2. ACM, 2018.

[30] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys'16, page 36. ACM, 2016.

[31] 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 3rd ACM Symposium on Cloud Computing*, SoCC'12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[32] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[33] L. E. Schrage and L. W. Miller. The queue m/g/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, Aug. 1966.

[34] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th European Conference on Computer Systems*, EuroSys'13, pages 351–364, New York, NY, USA, 2013. ACM.

[35] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys'16, page 35. ACM, 2016.

[36] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. n. Goiri, and R. Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 755–770, Berkeley, CA, USA, 2016. USENIX Association.