Kairos: Preemptive Data Center Scheduling Without Runtime Estimates

Pamela Delgado\(^1\), Diego Didona\(^1\), Florin Dinu\(^1\) and Willy Zwaenepoel\(^1\)

\(^1\)EPFL

Abstract

The vast majority of data center schedulers use job runtime estimates to improve the quality of their scheduling decisions. Knowledge about the 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 may 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 job runtimes. Kairos introduces a distributed approximation of the Least Attained Service (LAS) scheduling policy. Kairos consists of a centralized scheduler and a per-node scheduler. 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 YARN FIFO scheduler and Big-C, an open source state-of-the-art YARN-based scheduler that also uses preemption. We show that Kairos reduces the median job completion time by 37% (resp. 73%) and the 99-th percentile by 57% (resp. 30%), with respect to Big-C and FIFO. 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% and 85% the for the Google and Yahoo traces respectively.

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. That is, 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\(^1\) to improve the quality of their scheduling decisions in the face of job heterogeneity and data-parallelism [5, 19, 21, 22, 30, 31, 38]. Execution times from prior runs [5] or a preliminary profiling phase [13] are often used for this purpose. The accuracy of such estimates has a significant impact on the performance of these schedulers. For instance, queuing a 1-second job behind a job that is estimated to take 1 second but in reality takes 3 seconds doubles the completion time of the former job. Similarly, scheduling at the same time two jobs estimated to be of equal length may seem to provide excellent load balance, but in fact significant load imbalance occurs if one job turns out to be shorter and the other longer.

Limitations of estimates-based approaches. Unfortunately, obtaining accurate job runtime estimates is far from being trivial. We show in Section 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 [11, 31] – can lead to large errors ( > 100%). Our findings are confirmed by recent work that shows that more sophisticated approaches based on machine learning [30] 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 its execution time [1], 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 [9, 27]. Techniques to tackle these issues, such as queue re-balancing [31] or uncertainty-aware scheduling policies [30] have shown some success in mitigating the impact of misestimations, but they do not fundamentally address the problem.

Kairos. In this paper, we propose an alternative approach

\(^1\)We refer to such estimates as “job runtime estimates”.

to data center scheduling, which does not use job runtime estimates. Our approach draws from the Least Attained Service (LAS) scheduling policy [28]. LAS is a preemptive scheduling technique that executes first the task that has received the smallest amount of service so far. LAS is known to achieve good job completion times when the distribution of job runtimes has a high variance, as is the case in the often heavy-tailed data center workloads.

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 its attendant performance penalties. Task migration would be needed to allow a preempted task to resume its execution on any worker node with available resources.

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 balances the load.

A first challenge in this design is how to ensure high resource utilization in 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 core.

A second challenge is how 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 such a way to induce high variance in the distribution of the runtime of tasks assigned to each node.

We have implemented Kairos in YARN. We compare its performance against the YARN FIFO scheduler and Big-C [6], a state-of-the-art YARN-based scheduler that also uses preemption. We show that Kairos reduces the median job completion time by 37%, resp. 73%, and the 99-th percentile by 57%, resp., 30%, with respect to Big-C and FIFO. We evaluate Kairos at scale by implementing it in the Eagle scheduler [11] and comparing its performance against Eagle using traces from Google and Yahoo. Kairos improves the completion times of latency-sensitive jobs by up to 55% and 85% respectively.

**Contributions.** We make the following 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 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. Section 2 provides the necessary background. Section 3 describes the design of Kairos. Section 4 describes its implementation in YARN. Section 5 evaluates the performance of the Kairos YARN implementation. Section 6 provides simulation results. Section 7 discusses related work. Section 8 concludes the paper.

## 2 Background

### 2.1 Misestimations

**Estimates in existing systems.** Earlier schedulers like Sparrow [29] do not rely on prior information about job runtimes, but in heavy-tailed workloads latency-sensitive short tasks often experience long queueing delays due to head-of-line blocking [12].

Most state-of-the-art data center schedulers rely on job runtime estimates to make informed scheduling decisions [5, 11, 12, 13, 19, 20, 21, 22, 26, 41]. Estimates are used to avoid head-of-line blocking and resource contention, provide load balancing and fairness, and meet deadlines. The accuracy of job 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, or at past executions of similar tasks [5], or by means of on-line profiling [13]. A common estimation technique for the task duration is to take the average of the task durations over previous executions of the job [12, 31]. More sophisticated techniques rely on machine learning [30].

**Challenges in obtaining accurate estimates.** Unfortunately, obtaining accurate and reliable estimates is a far from easy task. Many factors contribute to the difficulty in obtaining reliable estimates. The scheduler may have limited or no information to produce estimates for new jobs, i.e., jobs that have never been submitted before [31]. Even if jobs are recurring, evidence indicates that changes in the input data set may lead to significant and hard-to-predict shifts in the runtime for a job [11]. Changes in the data placement may cause the job execution time to change. Skew in the input data distribution can lead tasks in the same job to have radically different runtimes [9, 27]. Finally, failures and transient resource utilization spikes may lead to stragglers [2], which not only have an unpredictable duration, but represent outliers in the data-set used to predict future runtimes for the same job.

We provide an example of the estimation errors that can affect job scheduling decisions by analyzing public traces that are widely used to evaluate data center schedulers. In particular, we consider the Cloudera [7], Yahoo [8], Google [32] and Facebook [7] traces. We study the distri-

---

**References**

[1]...

---

**Data**

- [28] LAS policy
- [6] Big-C scheduler
- [12] YARN FIFO
- [13] On-line profiling
- [31] Recurring jobs
- [29] Sparrow
- [26] YARN
- [41] Recent work
Figure 1: Prediction error when estimating the duration of each task in a job as the mean task duration in that job. (a) The PDF of the relative error (the '<100' datapoint includes all under-estimations of more than 100%, and the '>100' data point all over-estimations larger than 100%. (b) The CDF of the absolute relative error (the very tail of this distribution is not shown for the sake of readability).

2.2 Least-Attained-Service

Prioritizing short jobs. Typical data centers workloads are a mix of long and short jobs [8, 7, 32]. Giving higher priority to short jobs improves their response times by reducing head-of-line-blocking. The Shortest Remaining Processing Time (SRPT) scheduling policy [35] 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 [34].

Recent systems have successfully adopted SRPT in the context of data center scheduling [11, 24, 31]. These systems do not support preemption, so they implement a variant of SRPT, where the shortest task executes but, once started, a task 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 [28]. 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 for execution the one with the lowest attained service, i.e., the one that has executed for the smallest amount of time so far. We call such task the youngest one. In case there are \( n \) youngest tasks, 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. The rationale behind the effectiveness of this service demand prediction policy lies in the heavy-tailed service demand distribution.

Coping with misestimations and Kairos approach. Previous work has shown that job runtime misestimation leads to worse job completion times [11], and failure to meet service level objectives [13, 38] or job completion deadlines [38]. Some systems deal with misestimations by runtime correction mechanisms such as task cloning [2] and queue re-balancing [31], or by taking using a distribution of estimates rather than a single value estimates [30]. 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 [28] 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 it is the case in the often heavy-tailed data center workloads.
that is prevalent among production workloads. That is, if a task has executed for a long amount of time, it is likely that it is a large task, and hence it still has much to execute before completion. Hence, it is better to execute younger tasks, as they are more likely to be short tasks.

In addition, if the youngest task in the queue has an attained service time $T$, a new incoming task is going to be the youngest one until it has received a service of $T$ (if no other task arrives in the meantime). Hence, if the task is a short one—which is likely under the assumption of heavy-tailed runtime distribution– then it is likely that the task is going to complete within $T$, thus experiencing no queueing at all.

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 job 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.

Kairos approach to LAS. Kairos implements LAS in an approximate fashion. It uses a two-level scheduling hierarchy consisting of a central scheduler and a node scheduler 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 (see Section 3.2). The central scheduler assigns tasks to worker nodes in such a way to achieve load balancing and to maximize the effectiveness of LAS within each worker node (see Section 3.3).

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 for each task information such as its attained service time and, for running tasks, the start time of their current quantum. Each node scheduler implements LAS taking as input the number of available cores $N$ and the quantum of time $W$ to allow the interleaved execution of tasks with identical attained service times (as described in Section 2.2).

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 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 $t$. If there is a task $t'$ in the node queue with attained service time lower than $T$, $t'$ is scheduled for execution by pre-empting $t$ (Line 31). Otherwise, $t$ continues its execution, and the corresponding timer is reset (Line 39).

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

The node scheduler implements an anti-starvation

![Figure 2: Kairos architecture. Nodes schedulers implement LAS locally (Section 3.2) and periodically send information to the central scheduler, such as the number of tasks and the variance of their attained service times. The central scheduler uses this information to assign tasks to worker nodes in such a way to achieve load balancing (Section 3.3.2) and to maximize the effectiveness of LAS within each worker node (Section 3.3.3).](image)
Algorithm 1 Node scheduler

1: Set $\langle$TaskEntry $\rangle$ $\rightarrow$ IdleTasks, RunningTasks $\triangleright$ Track suspended/running tasks
2: upon event Task $t$ arrives do
3: TaskEntry te
4: te.task $\leftarrow$ $t$
5: te.attained $\leftarrow$ 0
6: te.start $\rightarrow$ now()
7: RunningTasks.add(te)
8: if (IdleCores.size() < $N$) then $\triangleright$ Free core can execute $t$
9: core $c$ $\leftarrow$ IdleCores.pop()
10: $\triangleright$ Preempt oldest running task
11: $t_p$ $\leftarrow$ argmax$_{t \in \text{RunningTasks}}$($t$.attained)
12: $t_p$.attained$+=$ now() $-$ $t_p$.start
13: $c$ $\leftarrow$ core serving $t_p$
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 (IdleCores.isEmpty()) then $\triangleright$ Run youngest suspended task
22: TaskEntry $t_e$ $\leftarrow$ argmin$_{t_i \in \text{IdleTasks}}$($t_i$.attained)
23: RunningTasks.add($(t_e)$)
24: assign $t_e$ to $c$
25: $t_e$.start $\leftarrow$ now()
26: $c$.startTimer(W)
27: Start $t_e$.task
28: else
29: IdleCores.push($c$)
30: upon event Timer fires on core $c$ running task $t$ do
31: TaskEntry $t_s$ $\leftarrow$ TaskEntry $c$ $::$ $c$.task $=$ $t$
32: $t_s$.attained$+=$ now() $-$ $t_s$.start
$\triangleright$ Find youngest suspended task
33: TaskEntry $t_m$ $\leftarrow$ argmin$_{t_i \in \text{IdleTasks}}$($t_i$.attained)
34: if ($t_m$.attained $\leq$ $t_s$.attained) then $\triangleright$ 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 $\leftarrow$ now()
40: place $t_m$.task on $c$
41: Start $t_m$.task
42: else $\triangleright$ Continue running $t$
43: $t_s$.start $\leftarrow$ now()
44: $c$.startTimer(W)
45: upon event Every $\Delta$ do
46: Heartbeat HB
47: $\text{HB}.num$ $\leftarrow$ IdleTasks.size() + RunningTasks.size()
48: $\text{HB}.var$ $\leftarrow$ var$_{t_i \in \text{IdleTasks} \cup \text{RunningTasks}}$
49: send HB to the central scheduler

Algorithm 2 Central scheduler

1: Queue $\text{CentralQueue}$ $\triangleright$ Queue where incoming tasks are placed
2: Node[numNodes] Nodes $\triangleright$ Entries track # tasks and attained service times
3: upon event New job $J$ arrives do
4: for task $t \in J$ do
5: Queue.push($t$)
6: upon event Heartbeat HB from Node $i$ arrives do
7: Nodes[i].var $\leftarrow$ HB var
8: Nodes[i].numTasks $\leftarrow$ HB.numTasks
9: procedure MAINLOOP
10: while (true) do
11: for $i = 0, . . . , N + Q$ do
12: $S_i$ $\leftarrow$ $\{\text{Node } m \in \text{Nodes} : m$.numTasks $=$ $i\}$
13: while ($S_i$ ! isEmpty() $\land$ CentralQueue.isEmpty()) do
14: Node $m$ $\leftarrow$ argmin$_{\text{Node } n \in S_i}$($n$.numTasks)
15: Task $t$ $\leftarrow$ CentralQueue.pop()
16: Assign $t$ to $m$
17: $S_i$ $\leftarrow$ $S_i$ \ { $m$ }
18: Sleep($\Delta$)

mechanism to avoid that long jobs can be preempted indefinitely (not shown in the pseudocode). Each task is associated with a counter that tracks how many times the task has been preempted. If a task is preempted more than a given number of times, then it has the right to run for a quantum of time, 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 execution window. However, it may also lead a preempted task in the node queue to wait for a long time before being it can run again, an undesir-
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) Enforcing that resources do not get wasted, i.e., there are no cores idle if there is any tasks in some queue (either the eligible idle tasks in the central or in any worker queue). This leads to high resource utilization and implies balancing the load among worker nodes (Section 3.3.2).

2) Maximizing LAS effectiveness, e.g., by improving the chances that short tasks bypass long tasks and task do not hurt each other response times by excessive interleaved executions (Section 3.3.3).

3.3.2 Load Balancing

The central scheduler aims to balance the load across worker nodes by enforcing that each of them is assigned an equal number of tasks. Hence, the first outstanding task in the central queue is placed on the worker node with the smallest amount 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 its short tasks and becomes idle while some tasks lie idle on the other worker node, waiting to receive service time.

To address this issue, the central scheduler enforces that each worker is assigned 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.

Kairos task-to-node dispatching policy achieves load balancing and high resource utilization, and is cheap enough to lead to low-latency scheduling decisions. This allows Kairos to sustain high job arrival rates without incurring long scheduling delays.

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 amount of possible load imbalance, but may lead many short tasks to sit in the central queue, instead of being assigned to a worker node, execute by preempting a previous task and potentially complete quickly. A high value of \( Q \), on the contrary, may lead to higher load imbalance, but enables more parallelism.

We assess the sensitivity of Kairos to the setting of \( Q \) in Section 5.3.3, where we show that Kairos performance are not dramatically affected by sub-optimal settings of \( Q \).

3.3.3 Maximizing LAS effectiveness.

Kairos implements a 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 the worker node, in the hope that by doing so it can significantly increase the variance on 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 show that maximizing the heterogeneity of task runtimes on each worker node is key to improve task completion times [4, 14]. Unlike previous studies, however, Kairos does not rely on an exact knowledge of the runtimes of the tasks on 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 Kairos implementation

We implement Kairos as part of YARN [18], 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.

The ResourceManager assigns tasks to worker nodes and communicates with the NodeManagers on the worker nodes. A NodeManager communicates with the ResourceManager by means of periodic heartbeat messages. These heartbeats contain information about 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 the possibility of a worker node to be allocated more containers than available cores.

**Kairos node scheduler.** The node scheduler of
Kairos is implemented within the ContainerManager by the KairosNodeScheduler component. The KairosNodeScheduler 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. [6]. 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.

5 Evaluation on a Real Cluster

5.1 Methodology and Baselines

We evaluate our Kairos prototype integrated in YARN by means of experiments on a medium-sized cluster using Hadoop jobs. We will release the source code of Kairos.

We compare Kairos to Big-C [6], the most recent scheduling system based on YARN whose source code is publicly available, and to the FIFO YARN scheduler.

Background on Big-C. Big-C uses available runtime estimates to perform task placements, and preemption as a mechanism to prioritize short tasks in case of high utilization. Big-C extends the capacity scheduler of YARN. Jobs are partitioned in classes, and each class is assigned a priority, which determines the share of resources dedicated to jobs in that class. When, according to its priority, a job should run and there is not enough resource available, tasks belonging to lower priority jobs are preempted.

Big-C defines two job classes, corresponding to long and short jobs. A job is classified according to available runtime estimates. A high share of the resources is assigned to short jobs, so as to prioritize them over long ones. Long tasks can opportunistically use more than their share of resources, if there is unused capacity. Such long tasks can be preempted by newly incoming short tasks.

We configure Big-C to use its standard value for the share of resources for short jobs (95%) and we classify as short the jobs in category 1, 2 and 3 (which represent 94% of the total jobs in our workload).

5.2 Test-bed

Platform. We use a cluster composed of 30 nodes interconnected by an 10Gbps Ethernet network. Each node is equipped with 4 CPUs per node, having a total of 120 cores to schedule on. We use Hadoop-2.7.1 to have the same code base as in Big-C. Containers use Docker-1.12.1 and their images were downloaded from the online repository sequenceig/hadoop-docker.

We set \(Q = 4\), corresponding to hosting on a node a number of tasks twice as high as the number of available cores, and \(W = 50s\), which allows the shortest tasks to execute within one quantum of time. We provide a sensitivity analysis to the setting of these parameters in Section 5.3.3.

Workloads. We consider a workload composed of 100 WordCount jobs, taken randomly from five categories of jobs described in Table 1 given their respective probabilities. The job inter-arrival times follow a Poisson distribution with mean 60s. The workload is inspired by the
Figure 4: CDF of the job completion times in Kairos, Big-C and stock FIFO from YARN. Kairos avoids high waiting times of short tasks by means of LAS. In Big-C, instead, the shortest tasks can wait for the completion of other longer tasks classified as short.

5.3 Experimental results

5.3.1 Job completion times.

Figure 4 reports the CDF of job completion times with Kairos, Big-C and stock FIFO scheduling in YARN.

Kairos achieves better job completion time than Big-C and FIFO at all percentiles. Short tasks in Kairos complete more quickly than in Big-C, which is visible by looking at lower percentiles. For example, Kairos reduces the 50-th percentile of job completion times by 37% with respect to Big-C (217 seconds vs 341) and by 73% with respect to FIFO (241 seconds vs 808). The reason for the performance improvement over Big-C is two-fold. First, short tasks can execute on any worker node at any moment in time in Kairos, while in Big-C a share of the resources is destined to long jobs. Second, worker nodes in Kairos accept \( Q \) more tasks than what they can process. This allows short tasks to be placed on a busy node and execute thanks to LAS. Instead, in Big-C a short task \( t_s \) cannot preempt another short task \( t_s' \), even if this \( t_s' \) is actually longer than \( t_s \). Hence, \( t_s \) has to wait for some node to have free resources before being scheduled for execution.

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

5.3.2 LAS-aware task dispatching

Figure 5 reports the CDF of job completion times in Kairos with different policies used to choose where to place a task. These policies are implemented when there are multiple worker nodes with the minimum number of tasks already assigned. The \textit{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 \textit{Sum} policy tries to assign a task to the least loaded node. \textit{Random} assigns the task to one node at random. The \textit{Var} policy is the one implemented by Kairos (see Section 3.3.2).

The plots show that the \textit{Var} policy is able to deliver better job completion times at all percentiles. The biggest gains over \textit{Random} and \textit{Sum} are at around the 30-th per-
centile and towards the tail of the distribution. The benefit at the 30-th percentile indicates that the shortest jobs, which account for 30% of the total (see Table 1), are effectively prioritized. The benefit at higher percentiles show that 

5.3.3 Sensitivity Analysis

We now show that Kairos maintains performance that are better than or comparable to Big-C even in case of sub-optimal setting of the parameters $W$ and $Q$. To this end, we study how the performance of Kairos vary 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 6a 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 $Q = 4$ we use in Kairos, but still deliver better performance than Big-C at each percentile.

The shape of the CDFs for different values of $Q$ match our analysis of Section 3.3. If $Q$ is too low, sometimes short tasks are kept in the central queue, thus preventing them to execute right away by means of LAS. This is visible for at the 20-th percentile of the CDF, where $Q = 2$ is worse than $Q = 8$. If $Q$ is too high, instead, short tasks can more often preempt longer ones, increasing their response times and thus leading to worse tail latencies.

Sensitivity to $W$. Figure 6b 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 negatively impacts 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 a too low value for $W$ has the effect that the execution of tasks on a worker node is much interleaved. This 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 have big quanta of times to use. This improves their completion times at the detriment of shorter jobs.

6 Large Scale Simulation Study

6.1 Methodology and baseline

We now evaluate Kairos on large-scale data centers by means of a simulation study using the popular Yahoo [8] and Google [32] traces. We compare Kairos to Eagle [11], the most recent system whose design is implemented in a simulator. We integrate Kairos design in the Eagle simulator and will make the source code of the simulator available. We do not compare our prototype of Kairos with Eagle because Eagle is built on top of Spark’s scheduler. We report average values of 10 runs for the Yahoo trace, and 5 runs for the Google trace.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Total # jobs</th>
<th>% Long jobs</th>
<th>% Task-Seconds long jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yahoo [8]</td>
<td>24262</td>
<td>9.41</td>
<td>98</td>
</tr>
<tr>
<td>Google [32]</td>
<td>506460</td>
<td>10.00</td>
<td>83</td>
</tr>
</tbody>
</table>

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.
tically 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. That is, Eagle aims to execute first 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 to use the same parameters as in its original implementation (which vary depending on the target workload trace). These include sub-cluster sizes, cutoffs to distinguish short jobs from long ones and parameters to implement SRPT.

6.2 Simulated Test-bed

Platform. We simulate different large-scale data centers with 15000-23000 worker nodes for the Google trace and 4000-8000 nodes for the Yahoo trace. We keep constant the job arrival rates to the values in the traces, so increasing the number of worker nodes reduces the load on the worker nodes. 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 trace and 2% for Yahoo. These values determine the size of the partition 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 = 10000 time units for the Google trace. The anti-starvation counter for Kairos is set to 3 for both traces.
6.3 Experimental results

Figure 7 and Figure 8 report the 50th, 90th and 99th percentiles of job completion times for Kairos normalized to the ones obtained by Eagle for the Yahoo and Google traces respectively. The plots on the left report short job completion times; the plots on the right report long job completion times. In addition, we report the average cluster utilization for both Kairos and Eagle as a function of the number of worker nodes in the cluster.

Figure 7 (a) and Figure 8 (a) show that Kairos improves the short job completions times significantly at high loads (up to 55% for Google and 85% for Yahoo). Kairos improvement are due to the fact that, when the load is very high, short jobs in Eagle are confined in the portion of the 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 small completion times. As the load decreases, the two systems achieve increasingly similar performance.

Long jobs exhibit different dynamics. Kairos reduces the completion times of most of the long jobs with respect to Eagle, when the load is at least 50%. This is visible looking at the 50-th and 90-th percentiles in Figure 7 (b) and Figure 8 (b). Kairos improves the completion times of most long jobs because it interleaves their executions, leading to better completion times for the shortest among the long jobs. In Eagle, instead, the absence of preemption may lead a relatively short task among the long ones to wait for the whole execution of a longer task. As what is seen for the short jobs, the differences in performance at the 50-th and 90-th percentiles level out as the load decreases.

Kairos achieves a slightly worse 99-th percentile with respect to Eagle (between 14% and 50% in Yahoo and between 11% and 33% for Google). This is due to the fact that Kairos frequently preempts the longest jobs to prioritize the shorter ones. This is an unavoidable, and we argue favorable, trade-off that Kairos makes to improve the performance for the vast majority of the jobs, especially latency-sensitive-ones, without requiring a priori knowledge on job runtimes.

Finally, Kairos and Eagle achieve the same resource utilization in both workloads and for all cluster sizes. This result showcases the capability of Kairos of achieving the same high resource utilization as approaches that rely on prior knowledge on the job 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 of the state-of-the-art scheduling systems rely on runtime estimates to perform informed scheduling decisions. These systems differ in how such estimates are integrated into the scheduling policy.

Apollo [5], Yaq [31] and Mercury [26] 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 [12], Eagle [11] and Big-C [6] 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 [12, 6] or eliminate [11] head-of-line-blocking.

Tetrisched [38], Rayon [10], Firmament [19], Quincy [25], Tetris [20], 3Sigma [30] and Medea [16] 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 [15] uses a simulator to speculate on the evolution of the system and accordingly decides the task-to-node placement. Graphene [22] 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 [21] exploits temporary relaxations of the fairness guarantees to allow a job to use resources destined to another one.

As opposed to these systems, Kairos eschews the need for any a priori information about job runtimes. Instead, Kairos infers the expected remaining runtime of tasks by looking at 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. Borg [39] and Mercury [26] kill low-priority jobs to reallocate the resources they are using to higher-priority jobs. In Hawk, if a node becomes idle, it steals tasks from other nodes. Yaq [31] and Mercury [26] migrate tasks that have not started yet to re-balance the load. LATE [40], Mantri [3], Dolly [2], Hopper [33] and DieHard [37] use
techniques like restarting or cloning tasks to cope with stragglers due to misestimations or to unexpected worker nodes slowdowns or failures. Tetrisched [38], 3Sigma [30], Rayon [10] and Jockey [15] periodically re-evaluate the scheduling plan and change it accordingly 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 [10], 3Sigma [30] and Big-C [6] make use of 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 [29] 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 $2t$ that have been contacted.

Tyrex [17] 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 $t_i$.

The system in [23] aims 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 [36], Sparrow [29], Apollo [5] and Yaq [31], or hybrid schedulers such as Mercury [26], Hawk [12] and Eagle [11] 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 [5] and Yaq [31], or shared among the distributed schedulers, e.g., as in Omega [36]. Existing techniques like randomly perturbing the state communicated to different schedulers [5] and atomic transactions over the shared view of the cluster [36] could be used to limit or avoid concurrent conflicting scheduling decisions by different schedulers.

### 8 Conclusion

We present Kairos, a new data center scheduler that makes no use of a priori job runtime estimates. Kairos achieves a good quality of the scheduling decisions, hence attaining low latency and high resource utilization, by employing in synergy two techniques. First, a lightweight use of preemption aimed to prioritize short tasks over long ones and avoid head-of-line-blocking. Second, a novel task-to-node assignment that employs in combination an admission control policy, aimed to reduce load imbalance among worker nodes, and assigns tasks to nodes so as to improve the chances that they complete quickly.

We evaluate Kairos by means of a experiments on a cluster with a full fledge prototype in YARN, and by means of large scale simulations. We show that Kairos achieves better job latencies than state-of-the-art approaches that use a priori job runtime estimates.
References


