Hybrid, Job-Aware, and Preemptive Datacenter Scheduling

THÈSE N^{O} 8892 (2018)

PRÉSENTÉE LE 2 NOVEMBRE 2018

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES D'EXPLOITATION
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Pamela Isabel DELGADO BORDA

acceptée sur proposition du jury:

Prof. P. Ienne, président du jury
Prof. W. Zwaenepoel, directeur de thèse
Prof. P. R. Pietzuch, rapporteur
Prof. G. Muller, rapporteur
Prof. A.-M. Kermarrec, rapporteuse

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2018

"I was taught the method for advancement
is not quick or simple."
— Marie Curie

"All truths are easy to understand once they are discovered;
the point is to discover them."
— Galileo Galilei

To my dearest wonderful family…
Jean-Paul, Aileen & Julie;
Miguel, Roxana, Mónica, Beatriz, Alejandra & Pablo

Acknowledgements

This thesis is the culmination of an exciting yet challenging stage in my life. During this journey I was not alone and received the help, in more than one way, from many people. I thank deeply each and every one of them for being part of, and contributing to, my PhD life.

First and foremost, I want to express my gratitude to Willy. As my supervisor, he not only gave me the opportunity to grow both professionally and personally, but also taught me what it takes to be an outstanding researcher and person. His guidance - full of wisdom and optimism, putting effort where it matters, not compromising quality, and giving confidence when needed - was key to the development of this thesis. I hope to, one day, reflect on those traits too.

While working for my thesis I was fortunate to collaborate with talented researchers from whom I also learned a lot: Florin, Diego and Anne-Marie. This thesis would not be the same without their help.

I am particularly grateful to Anne-Marie, Peter, and Gilles for being part of my committee, for reading my thesis and giving me useful feedback. Thanks to Paolo for serving as a president for my oral defense.

I would like to also thank my colleagues at Microsoft Research Cambridge, in particular, Sergey for hosting and mentoring me during my internship, and Christos for his encouragements and useful feedback. Moreover, I truly appreciate Microsoft Research for generously supporting my research thanks to the Swiss JRC.

Thanks to the DataCenter Observatory for providing me with the tools and necessary equipment to perform my experiments, without them this dissertation would not be possible.

During these years I enjoyed being part of LABOS and being surrounded by extraordinary people that made my PhD life more bearable: Jasmina, Kristina, Maria, Calin, Laurent, Oana, Florin, Diego and Baptiste. I want to thank former lab members as well: Amitabha, Jiaqing, Mihai, Peter and Maciej. I also thank Madeleine, for her endless adminitrative and logistic support. Last but not least, I cannot forget some wonderful friends that accompanied me during these years: Alex, Gorica, Alexandra, Tri, Mehdi, Sonia and Ana.

Throughout these years my family has been amazingly supportive. There are no words to express how much of this thesis and my life I owe to them.

Lausanne, 06 Octobre 2018 Pamela

Abstract

Scheduling in datacenters is an important, yet challenging problem. Datacenters are composed of a large number, typically tens of thousands, of commodity computers running a variety of data-parallel jobs. The role of the scheduler is to assign cluster resources to jobs, which is not trivial due to the large scale of the cluster, as well as the high scheduling load (tens of thousands of scheduling decisions per second).

Additionally to scalability, modern datacenters 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. In such workloads, and especially if the cluster is highly utilized, it is challenging to avoid short running jobs getting stuck behind long running jobs, i.e. head-of-line blocking. Schedulers have evolved from being centralized (one single scheduler for the entire cluster) to distributed (many schedulers that take scheduling decisions in parallel). Although distributed schedulers can handle the large-scale nature of datacenters, they trade scheduling latency for accuracy.

The complexity of scheduling in datacenters is exacerbated by the data-parallel nature of the jobs. That is, a job is composed of multiple tasks and the job completes only when all of its tasks complete. A scheduler that takes into account this fact, i.e. job-aware, could use this information to provide better scheduling decisions.

Furthermore, to improve the quality of their scheduling decisions, most of datacenter schedulers use job runtime estimates. Obtaining accurate runtime estimates is, however, far from trivial, and erroneous estimates may lead to sub-optimal scheduling decisions.

Considering these challenges, in this dissertation we argue the following: (i) that a hybrid centralized/distributed design can get the best of both worlds by scheduling long jobs in a centralized way and short jobs in a distributed way; (ii) such a hybrid scheduler can avoid head-of-line blocking and provide job-awareness by dynamically partitioning the cluster for short and long jobs and by executing a job to completion once it started; (iii) a scheduler can dispense with runtime estimates by sharing the resources of a node with preemption, and load balancing jobs among the nodes.

Keywords: datacenter, scheduling, hybrid, preemptive, job-aware, data center, datacentre, job aware

Résumé

L'ordonnancement dans les centres de données est un problème important mais difficile à résoudre. Les centres de données sont composés d'un grand nombre, typiquement des dizaines de milliers d'ordinateurs, de base exécutant une variété de travaux parallèles aux données. Le rôle de l'ordonnanceur est d'affecter les ressources du cluster aux tâches, ce qui n'est pas négligeable en raison de la grande échelle du cluster, ainsi que de la charge de planification élevée (des dizaines de milliers de décisions d'ordonnancement par seconde).

En plus de l'évolutivité, les centres de données modernes doivent faire face à des charges de travail de plus en plus hétérogènes composées de travaux de longue durée, par exemple l'analyse de données, et de travaux courts sensibles au temps de latence, par exemple l'exploitation de services en contact avec les utilisateurs. Dans de telles charges de travail, et en particulier si le centre de données est très utilisé, il est difficile d'éviter que des travaux de courte durée ne se retrouvent coincés derrière des travaux de longue durée, c'est-à-dire le blocage en tête de ligne. Les planificateurs sont passés d'un système centralisé (avoir juste un seul planificateur pour l'ensemble du centre de données) à un système distribué (avoir des nombreux planificateurs qui prennent des décisions de planification en parallèle). Bien que les planificateurs distribués puissent gérer la nature à grande échelle des centres de données, ils échangent le temps d'ordonnancement pour plus de précision.

La complexité de l'ordonnancement dans les centres de données est exacerbée par la nature parallèle des tâches. C'est-à-dire qu'un travail est composé de tâches multiples et que le travail ne se termine que lorsque toutes ses tâches sont terminées. Un planificateur qui tient compte de ce fait, ce qu'on appelle job-aware, pourrait utiliser cette information pour prendre de meilleures décisions en matière d'ordonnancement.

De plus, pour améliorer la qualité de leurs décisions d'ordonnancement, la plupart des planificateurs de centres de données utilisent des estimations du temps d'exécution des tâches. L'obtention d'estimations précises de la durée d'exécution est cependant loin d'être triviale, et des estimations erronées peuvent conduire à des décisions d'ordonnancement sous optimales.

Compte tenu de ces défis, dans la présente thèse, nous soutenons ce qui suit : (i) qu'une conception hybride centralisée/distribuée peut obtenir le meilleur des deux mondes en planifiant les travaux longs de manière centralisée et les travaux courts de manière distribuée ; (ii) un tel programmateur hybride peut éviter le blocage en tête de ligne et fournir une approche *job-aware*, en partitionnant dynamiquement le cluster pour les travaux courts et longs et en exécutant un travail jusqu'à son achèvement une fois qu'il a commencé ; (iii) un planificateur peut se passer des estimations de temps d'exécution en partageant les ressources d'un nœud

avec des tâches de préemption et d'équilibrage de charge entre les nœuds.
Mots clefs : centre de données, ordonnancement, hybride, préemptif, sensible au travail

Contents

List of Figures

List of Tables

1 Introduction

"I do not know anything, but I do know that
everything is interesting if you go into it deeply enough."
— Richard Feynman

Large-scale datacenters are of paramount importance to operate millions of computations per day [16]. Big data companies (e.g. Google, Yahoo!, Facebook, Cloudera, Twitter, Amazon, Microsoft) run a variety of data-parallel jobs in datacenters that are composed of tens of thousands of commodity machines. A datacenter scheduler is the component in charge of deciding which -and when- resources are assigned to jobs. Datacenter schedulers must provide high-quality placement of jobs while dealing with scalability, high utilization of the resources, and job heterogeneity. Consequently, datacenter scheduling is not an easy task and is an active topic of research. This dissertation aims to improve the state-of-the-art with scheduler designs and techniques that tackle some of the main datacenter scheduling challenges (we explore these in Section 1.2).

In this Chapter, we explore the context of datacenter scheduling in Section 1.1; we elaborate on the scheduling challenges tackled by this dissertation in Section 1.2; we then provide an overview of the research contributions in Section 1.3 and finally we describe the outline of the rest of the thesis in Section 1.4.

1.1 Datacenter scheduling

Datacenter scheduling is an important yet unsolved problem. On the one hand, scheduling has been studied extensively well before the datacenter era in queuing theory [93][1], in operating systems [95], and in early computer science in general [63, 24]. In particular, scheduling is a combinatorial optimization NP-complete problem [39, 61]. On the other hand, there is a need for solutions that match datacenter scheduling requirements. Since datacenter scheduling

[1] Queuing theory dates back its origins in the Copenhagen telephone exchange [93]

cannot be formulated as existing theoretical models, solutions proposed in the literature are mostly a combination of heuristics and techniques inspired by theory. There have been efforts to model the performance of scheduling in systems [48] more formally, however these models are not suitable for a datacenter context. Furthermore, there is not a one size fits all solution. In addition to the unsolved questions addressed by the queueing theory and algorithmic communities, jobs in datacenters present an extra challenge because they are typically composed of more than one task, as we explain below.

To face the ever growing data deluge in the last decades, there has been a paradigm shift from mainframe computing to distributed computing [2]. To user-facing services, resources in a cluster should appear as being one single system and the underlying software they run (i.e. operating systems) is not built for providing such a functionality for a distributed set of machines. Because of this, an ecosystem of tools and frameworks has emerged to manage computation and storage in a distributed way. To process big amounts of data in a distributed and parallel way, Google's MapReduce [26] model was proposed as an alternative to existing data processing models. These data-parallel MapReduce-like jobs are composed of one or more stages that divide the work in many tasks to exploit parallelism.

One example of such an ecosystem is Hadoop [10] and its related frameworks. Hadoop [10] was one of the first frameworks that incorporated the MapReduce model and, due to its open source nature, had a wide and rapid adoption. Examples of frameworks that run on top of Hadoop MapReduce are FlumeJava [17], Hive [97], Pig [76], Tenzing [18], Giraph (an open source counterpart of Pregel [71]).

MapReduce-like jobs, became since then the norm for processing batch jobs in datacenters [108, 100]. Scheduling in such a context is a key component of datacenters and has evolved quickly. The main challenges tackled by schedulers since the beginning were scalability, heterogeneity, locality and fault tolerance [94]. These functionalities were managed by a resource manager component, but later, with scheduling scalability becoming more important, it has been separated from other functions like fault tolerance [100]. Furthermore, locality became less of a worry with the evolution of networking and the greater/cheaper disk capacity . The main difficult requirements that today's datacenter scheduling face are: dealing with heterogeneity of jobs, be scalable, and achieve high resource utilization. Additionally, a datacenter scheduler can take into account hardware constraints such as: memory, network bandwidth, I/O bandwidth, GPUs. Scheduling with constraints is out of the scope of this thesis. A discussion about constraints and how to incorporate them with the designs and techniques presented in this dissertation can be found in Chapter 5.

[2] One in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages [23].

1.1.1 Heterogeneous jobs

Large clusters have to deal with an increasing number of jobs, which can vary significantly in size and have very different requirements with respect to latency [21, 20]. Short jobs, due to their nature, are latency sensitive. Long jobs, such as graph analytics, can tolerate long latencies but suffer more from bad scheduling placement. Efficiently scheduling such heterogeneous workloads in a datacenter is an increasingly important problem. At the same time, maintaining good response times for short jobs becomes harder under high load. A typical datacenter workload [21, 20, 103] is composed mainly (>90%) of short jobs, while most of the resources time (>83.6%) is used running long jobs.

1.1.2 Scale

The unprecedented scale moved the community efforts towards improving scheduling latency. On the one side, data to be processed became predominantly larger. On the other side, to cope with this big data, clusters also became bigger (tens or hundreds of thousands of servers). As a result, datacenter workloads combine long best-effort jobs, that process the ever-growing data (for example web indexing) with low-latency short jobs that face the user (for example web search). Furthermore, the scheduler must deal with a peak load of 20k tasks per second [16].

1.1.3 High utilization

Although the idea of using many machines to run jobs in parallel in principle helps to process big data effectively, resources in datacenters are underutilized and sometimes adding more resources to a single server is better than scaling out [12]. Therefore there have been efforts to improve resource efficiency at scale [108, 69]. Understanding how to run datacenters at high utilization is becoming increasingly important. Resource-efficiency reduces provisioning and operational costs as the same amount of work can be performed with fewer resources [65]. For instance, utilizing cluster resources efficiently translates into direct savings in money and energy, for instance, costs of datacenter servers make up for 57% of the total costs monthly [47]. Moreover, datacenter operators need to be ready to maintain acceptable levels of performance even during peak request rates, which may overwhelm the datacenter.

1.2 Scheduling challenges

Efficiently scheduling jobs in datacenters is an interesting yet challenging research problem. We focus on four main challenges: conflicting goals of scheduling latency versus high-quality placement, head-of-line blocking, the lack of job-awareness and the dependency on runtime estimates. This Section explains what these are and next Section 1.3 shows how we address them in this dissertation.

1.2.1 Conflicting scheduling latency and high-quality scheduling

The first-generation of cluster schedulers, such as the one used in Hadoop [106] or [58], are centralized: all scheduling decisions are made in a single place. A centralized scheduler has near-perfect visibility into the utilization of each node and the demands in terms of jobs to be scheduled. In practice, however, the very large number of scheduling decisions and status reports from a large number of servers can overwhelm centralized schedulers, and in turn lead to long latencies before scheduling decisions are made. This latency is especially problematic for short jobs that are typically latency-bound, and for which any additional latency constitutes a serious degradation. In order to overcome the limitations of centralized schedulers for large clusters with many competing jobs, there has been a movement towards distributed scheduling [33, 88, 79, 16]. Such distributed schedulers are inherently scalable, but may make poor scheduling decisions because of limited visibility into the overall resource usage in the cluster. In particular, under high load, short jobs can fare poorly with such a distributed scheduler. The pros and cons of distributed schedulers are exactly the opposite of centralized ones: scheduling decisions can be made quickly, but by construction they rely on partial, possibly out-dated (or no) information and may therefore lead to inferior scheduling decisions. Therefore, depending on the scheduling design (centralized, distributed), there is a trade-off between scheduling latency and high-quality placement.

1.2.2 Head-of-line blocking

An effect of having heterogeneous jobs in a high-loaded cluster is the head-of-line blocking problem: a latency-sensitive short job gets scheduled behind a long one. As described in Section 1.1, typical datacenters workloads are composed of a small amount of long-lived best-effort jobs and a large portion of latency-sensitive jobs. If this heterogeneity is not taken into account, short jobs will experience head-of-line blocking. In addition, the fact that most of the cluster's time is spent running long jobs, there is a high probability of this happening. This problem is exacerbated with high load, because even with careful online scheduling algorithms there are high chances that the cluster will be entirely occupied by running long jobs hurting thus greatly the short jobs for potentially a long period of time. Short jobs, which make most of the workload, are greatly affected because waiting for long tasks to finish can increase their running time by two orders of magnitude. To circumvent head-of-line blocking, researchers proposed to make use of queue reordering [81] both in the node queues and in a central queue. However, only reordering (without preemption) is not enough to avoid all the head-of-line blocking, because it suffices to have a combination of long tasks started in all of the cluster to harm incoming short tasks.

1.2.3 Lack of job-awareness

Another important datacenter scheduling challenge is the parallel nature of the jobs: the overall completion time of a job is equal to that of its slowest task. Therefore, scheduling

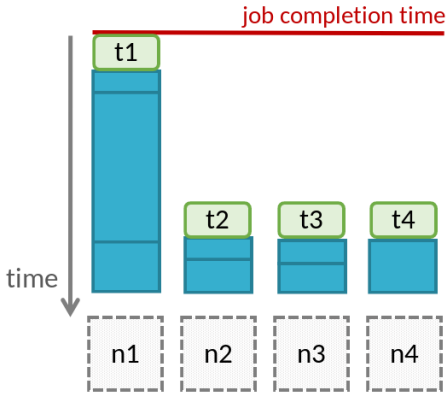


Figure 1.1 – Lack of job-awareness in schedulers leads to bad job completion times.

individual tasks without taking into account other tasks in the same job can result in scheduler induced stragglers.

Figure 1.1 shows an example of scheduler induced stragglers. When a job with four tasks $t1$, $t2$, $t3$ and $t4$ arrives a non job-aware scheduler could schedule tasks like in the picture. However, that is not the best scheduling to reduce the job's completion time. Since distributed schedulers have limited to no visibility, it is easy to see this situation happening.

Most of the centralized schedulers in the literature schedule at the task level 5.3. However, the situation depicted in Figure 1.1 can still happen due to unforeseen situations like misestimations 1.2.4.

1.2.4 Dependency on runtime estimates

Many state-of-the-art systems rely on estimates of the runtimes of tasks within a job[3] to improve the quality of their scheduling decisions in the face of job heterogeneity and data-parallelism [16, 42, 44, 45, 81, 99, 80]. Execution times from prior runs [16] or a preliminary profiling phase [31] 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 job behind a job that is estimated to take 1 second but in reality takes 3 seconds, will double 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. In fact, significant load imbalance occurs if one job turns out to be shorter and the other longer.

Unfortunately, obtaining accurate job runtime estimates is far from being trivial. We show in Chapter 4 that using the mean task execution time as a predictor of the execution of all tasks in a job [27, 81] can lead to large errors (> 100%). These findings are confirmed by recent work that shows that more sophisticated approaches based on machine learning [80]

[3] We refer to such estimates as "job runtime estimates".

still exhibit significant estimation errors. Techniques such as queue re-balancing [81] or uncertainty-aware scheduling policies [80] have shown some success in mitigating the impact of misestimations, but they do not fundamentally address the problem.

1.3 Research contributions

Taking into consideration the challenges presented in Section 1.2, this dissertation investigates the following hypotheses:

Thesis statement 1 A datacenter scheduler can get the best of both worlds: high-quality placement and good scheduling latency. Furthermore, if provided with estimates, such a scheduler can be job-aware and avoid head-of-line blocking completely.

Thesis statement 2 A datacenter scheduler can achieve good scheduling latency and job runtimes without the need of any a-priori assumption about an incoming job.

This dissertation proposes a set of scheduling designs and techniques to prove the above statements. We have built three systems: Hawk, Eagle, and Kairos as supporting evidence to prove that those techniques and designs work well in practice. In Hawk we propose a novel hybrid design for a datacenter scheduler. Furthermore, Eagle augments such a hybrid design with techniques to avoid completely the head-of-line blocking and at the same time provide job-awareness. Finally, Kairos proposes a competent new scheduling approach that is not dependent on estimates.

This dissertation makes therefore four principal contributions:

1. The first *hybrid scheduler design that provides both high-quality placement and low scheduling latency* for datacenters. This dissertation argues that the scheduling of long and short jobs should be treated differently because of their nature. By scheduling the long jobs in a centralized way and the short jobs in a distributed way, such a scheduler can get the best of both worlds.
2. A new *technique to completely avoid head-of-line blocking* for distributed and hybrid schedulers. We show that a scheduler that dynamically divides the nodes for the execution of short and long jobs can improve greatly the running times of short jobs by avoiding the head-of-line blocking.
3. A new *technique that provides job-awareness* that, combined with the previous technique and a hybrid scheduling design improves job running times even further. This technique avoids scheduler-induced stragglers by running a job to completion once it has started.
4. The first *estimation-independent scheduler design* that can reach job running times comparable (or better) than estimation-based approaches. In this thesis we argue that a

preemptive datacenter scheduler can achieve competitive job running times without assuming any a-priori knowledge. This is achieved by implementing a distributed approximation of the Least Attained Service (LAS) [75] scheduling policy in the datacenter scheduling context.

1.4 Dissertation outline

In the following three chapters of this dissertation we present the systems built to prove our solution and the experiments conducted in our three main systems.

1.4.1 Hybrid scheduling

Chapter 2 addresses the problem of efficient scheduling of large clusters under high load and heterogeneous workloads. It describes and evaluates a new hybrid centralized/distributed scheduler, called Hawk. In Hawk, long jobs are scheduled using a centralized scheduler, while short ones are scheduled in a fully distributed way. Moreover, a small portion of the cluster is reserved for the use of short jobs. In order to compensate for the occasional poor decisions made by the distributed scheduler, we propose a novel and efficient randomized work-stealing algorithm. In Chapter 2 Hawk is evaluated using a trace-driven simulation and a prototype implementation in Spark [11]. In particular, using a Google trace, we show that under high load, compared to the purely distributed Sparrow scheduler, Hawk improves the 50th and 90th percentile runtimes by 80% and 90% for short jobs and by 35% and 10% for long jobs, respectively. Measurements of a prototype implementation using Spark on a 100-node cluster confirm the results of the simulation.

1.4.2 Job-aware scheduling

Chapter 3 presents Eagle, a new hybrid datacenter scheduler for data-parallel programs. Eagle dynamically divides the nodes of the datacenter in partitions for the execution of long and short jobs, thereby avoiding head-of-line blocking. Furthermore, it provides job-awareness and avoids stragglers thanks to a new technique, called Sticky Batch Probing (SBP).

The dynamic partitioning of the datacenter nodes is accomplished by a technique called Succinct State Sharing (SSS), in which the distributed schedulers are informed of the locations where long jobs are executing. SSS is particularly easy to implement with a hybrid scheduler, in which the centralized scheduler places long jobs.

With SBP, when a distributed scheduler places a probe for a job on a node, the probe stays there until all tasks of the job have been completed. When finishing the execution of a task corresponding to probe P, rather than executing a task corresponding to the next probe P' in its queue, the node may choose to execute another task corresponding to P. We use SBP in combination with a distributed approximation of Shortest Remaining Processing Time

(SRPT) [48] with starvation prevention.

Eagle is implemented as a Spark plugin, and we have measured job completion times for a subset of the Google trace on a 100-node cluster for a variety of cluster loads. We provide simulation results for larger clusters, different traces, and for comparison with other scheduling disciplines. We show that Eagle outperforms other state-of-the-art scheduling solutions at most percentiles, and is more robust against mis-estimation of task duration.

1.4.3 Preemptive scheduling

Chapter 4 presents Kairos, a novel datacenter 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.

Kairos is implemented in YARN [100]. We compare its performance against the YARN FIFO scheduler and Big-C [19], 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.

1.4.4 Related work and conclusions

The remaining of the dissertation describes the contributions of this thesis in the context of the related work in **Chapter 5**. It first explores the common approaches to solve the problems that are main focus of this thesis, then it explores diverse correction mechanisms that datacenter schedulers use to avoid scheduling problems. Chapter 5 also explores other properties that are provided by the state-of-the-art schedulers. Finally, before concluding, we provide an overview of scheduling in other contexts.

Chapter 6 concludes this dissertation and highlights directions for future work.

2 Hawk: Hybrid Datacenter Scheduling

"The normal case must be fast. The worst case must make some progress."
— Butler Lampson, Hints for Computer System Design

2.1 Introduction

This chapter addresses the problem of efficient scheduling of large clusters under high load and heterogeneous workloads. An heterogeneous workload typically consists of many short jobs and a small number of large jobs that consume the bulk of the cluster's resources. To overcome the limitations of centralized schedulers for large clusters with many competing jobs papers in the literature advocate for distributed scheduling. Distributed schedulers achieve good scheduling latency but trade for poor scheduling decisions due to the limited visibility on the cluster usage. In particular, we demonstrate that under high load, short jobs can fare poorly with such a distributed scheduler.

In this chapter, we propose Hawk, a hybrid scheduler, staking a middle ground between centralized and distributed schedulers. Attempting to achieve the best of both worlds, Hawk centralizes the scheduling of long jobs and schedules the short jobs in a distributed fashion. To compensate for the occasional poor choices made by distributed job scheduling, Hawk allows task stealing for short jobs. In addition, to prevent long jobs from monopolizing the cluster, Hawk reserves a (small) portion of the servers to run exclusively short jobs.

The rationale for our hybrid approach is as follows. First, the relatively small number of long jobs does not overwhelm a centralized scheduler. Hence, scheduling latencies remain modest, and even a moderate amount of scheduling latency does not significantly degrade the performance of long jobs, which are not latency-bound. Conversely, the large number of short jobs would overwhelm a centralized scheduler, and the scheduling latency added by a centralized scheduler would add to what is already a latency-bound job. Second, by scheduling long jobs centrally, and by the fact that these long jobs take up a large fraction of

the cluster resources, the centralized scheduler has a good approximation of the occupancy of nodes in the cluster, even though it does not know where the large number of short jobs are scheduled. This accurate albeit imperfect knowledge allows the scheduler to make well-informed scheduling decisions for the long jobs. There is, of course, the question of where to draw the line between short and long jobs, but we found that benefits result for a large range of cutoff values.

The rationale for using randomized work stealing is based on the observation that, in a highly loaded cluster, choosing uniformly at random a loaded node from which to steal a task is very likely to succeed, while finding at random an idle node, as distributed schedulers attempt to do, is increasingly less likely to succeed as the slack in the cluster decreases.

We evaluate Hawk through trace-driven simulations with a Google trace [33] and workloads derived from [20, 21]. We compare our approach to a state-of-the-art fully distributed scheduler, namely Sparrow [79] and to a centralized one. Our experiments demonstrate that, in highly loaded clusters, Hawk significantly improves the performance of short jobs over Sparrow, while also improving or matching long job performance. Hawk is also competitive against the centralized scheduler.

Using the Google trace, we show that Hawk performs up to 80% better than Sparrow for the 50th percentile runtime for short jobs, and up to 90% for the 90th percentile. For long jobs, the improvements are up to 35% for the 50th percentile and up to 10% for the 90th percentile. The differences are most pronounced under high load but before saturation sets in. Under low load or overload, the results are similar to Sparrow. The results are similar for the other traces: Hawk sees the most improvements under high load, and in some cases the improvements are even higher than those seen for the Google trace.

We break down the benefits of the different components in Hawk. We show that both reserving a small part of the cluster and work stealing are essential to good performance for short jobs, with work stealing contributing the most to the overall improvement, especially for the 90th percentile runtimes. The centralized scheduler is a key component for obtaining good performance for the long jobs.

We implement Hawk as a scheduler plug-in for Spark [107], by augmenting the Sparrow plug-in with a centralized scheduler and work stealing. We evaluate the implementation on a cluster of 100 nodes, using a small sample of the Google trace. We demonstrate that the general trends seen in the simulation hold for the implementation.

Contributions. In summary, in this chapter we make the following contributions:

1. We propose a novel hybrid scheduler, Hawk, combining centralized and distributed schedulers, in which the centralized entity is responsible for scheduling long jobs, and short jobs are scheduled in a distributed fashion.

2. We introduce the notion of randomized task stealing as part of scheduling data-parallel jobs on large clusters to "rescue" short tasks queued behind long ones.

3. Using extensive simulations and implementation measurements we evaluate Hawk's benefits on a variety of workloads and parameter settings.

2.2 Motivation

2.2.1 Prevalent workload heterogeneity

Workload heterogeneity is the norm in current datacenters [20, 82]. Typical workloads are dominated by short jobs. Long jobs are considerably fewer, but dominate in terms of resource usage. In this chapter, we precisely address scheduling for such heterogeneous workloads.

To showcase the degree of heterogeneity in real workloads, we analyze the publicly available Google trace [103, 82]. We order the jobs by average task duration. The top 10% jobs account for 83.65% of the task-seconds (i.e., the product of the number of tasks and the average task duration). Moreover, they are responsible for 28% of the total number of tasks, and their average task duration is 7.34 times larger than the average task duration of the remaining 90% of jobs.

Workload	% **Long Jobs**	% **Task-seconds**
Google	10.00%	83.65%
Cloudera-b	7.67%	99.65%
Cloudera-c	5.02%	92.79%
Cloudera-d	4.12%	89.72%
Facebook	2.01%	99.79%
Yahoo	9.41%	98.31%

Table 2.1 – Long jobs in heterogeneous workloads form a small fraction of the total number of jobs, but use a large amount of resources.

We also analyzed additional workloads described in [20, 21]. Table 2.1 shows the percentage of long jobs among all jobs, and the percentage of task-seconds contributed by the long jobs. The same pattern emerges in all cases, even for different providers: the long jobs account for a disproportionate amount of resource usage.

The numbers we provided also corroborate previous findings from several other researchers [5, 83, 106].

2.2.2 High utilization in datacenters

Understanding how to run datacenters at high utilization is becoming increasingly important. Resource-efficiency reduces provisioning and operational costs as the same amount of work

can be performed with fewer resources [65]. Moreover, datacenter operators need to be ready to maintain acceptable levels of performance even during peak request rates, which may overwhelm the datacenter.

Related work has approached the problem from the point of view of a single datacenter server [30, 31]. For a single server, the challenge is to maximize resource utilization by collocating workloads without the danger of decreased performance due to contention. As a result, several isolation and resource allocation mechanisms have been proposed, ensuring that resources on servers are well and safely utilized [96, 100].

Running highly utilized datacenters presents additional, orthogonal challenges beyond a single server. The problem we are targeting consists of scheduling jobs to servers in a scalable fashion such that all resources in the cluster are efficiently used.

2.2.3 Challenges in performing distributed scheduling at high load

We next highlight by means of simulation why a heterogeneous workload in a loaded cluster is a challenge for a distributed scheduler. The main insight is that with few idle servers available at high load, distributed schedulers may not have enough information to match incoming jobs to the idle servers. As a result, unnecessary queueing will occur. The impact of the unnecessary queueing increases dramatically for heterogeneous workloads.

We illustrate this insight in more detail using the Sparrow scheduler, a state-of-the-art distributed cluster scheduler [79]. In Sparrow, each job has its own scheduler. To schedule a job with t tasks, the scheduler sends probes to $2t$ servers. When a probe comes to the head of the queue at a server, the server requests a task from the scheduler. If the scheduler has not given out the t tasks to other servers, it responds to the server with a task. This technique is called "batch probing". More details can be found in the Sparrow paper [79], but the above suffices for our purposes. Sparrow is extremely scalable and efficient in lightly and moderately loaded clusters, but under high load, few servers are idle, and $2t$ probes are unlikely to find them. More probes could be sent, but the paper found that this is counterproductive because of messaging overhead.

We use the same simulator employed by Sparrow [79] to investigate the following scenario: 1000 jobs need to be scheduled in a cluster of 15000 servers. 95% of the jobs are considered short. Each short job has 100 tasks, and each task takes 100s to complete. 5% of the jobs are long. Each has 1000 tasks, and each task takes 20000s. The job submission times are derived from a Poisson distribution with a mean of 50s. We measure the cluster utilization (i.e., percentage of used servers) every 100s. The median utilization is 86%, and the maximum is 97.8%. This suggests that at least 300 servers (2%) are free at any time, enough to accommodate all tasks of any incoming short job.

Figure 2.1 presents the cumulative frequency distribution (CDF) of the runtimes of short jobs. A large fraction of short jobs exhibit runtimes of more than 15000 seconds, far in excess of

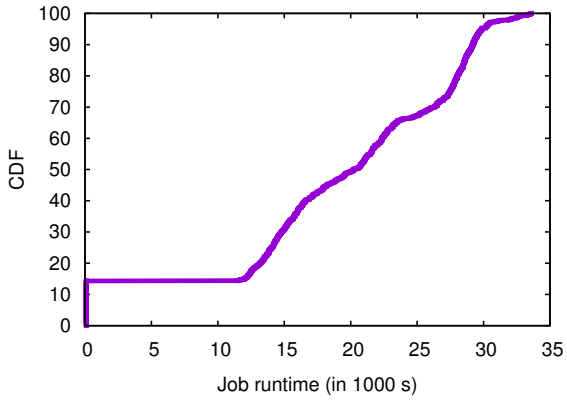


Figure 2.1 – CDF of runtime for short jobs, in a loaded cluster, using Sparrow.

their execution time, which clearly indicates a large amount of queuing, mostly behind long jobs. Given that enough servers are free, an omniscient scheduler would yield job runtimes of 100s for the majority of the short jobs. With Sparrow, if all tasks are 100s long, the impact of queueing is less severe. However, a heterogeneous workload coupled with high cluster load has a strong negative impact on the performance of short jobs.

2.3 The Hawk Scheduler

2.3.1 System model

We consider a cluster composed of server (worker) nodes. A job is composed of a set of tasks that can run in parallel on different servers. Scheduling a job consists of assigning every task of that job to some server. We use the terms long task and short tasks to refer to tasks belonging to long jobs or short jobs respectively. A job completes only after all its tasks finish. Each server has one queue of tasks. When a new task is scheduled on a server that is already running a task, the task is added to the end of the queue. The server queue management policy is FIFO.

2.3.2 Hawk in a nutshell

The previous section demonstrated that *(i)* many cluster workloads consist of a short number of long jobs that take up the bulk of the resources and a large number of short jobs that take up only a small amount of the total resources, and *(ii)* existing distributed cluster scheduling systems, exemplified by Sparrow, do not provide good performance for short jobs in such an environment, due to head-of-line blocking.

In this context, Hawk's goals are:

1. to run the cluster at high utilization,
2. to improve performance for short jobs, which are the most penalized ones in highly

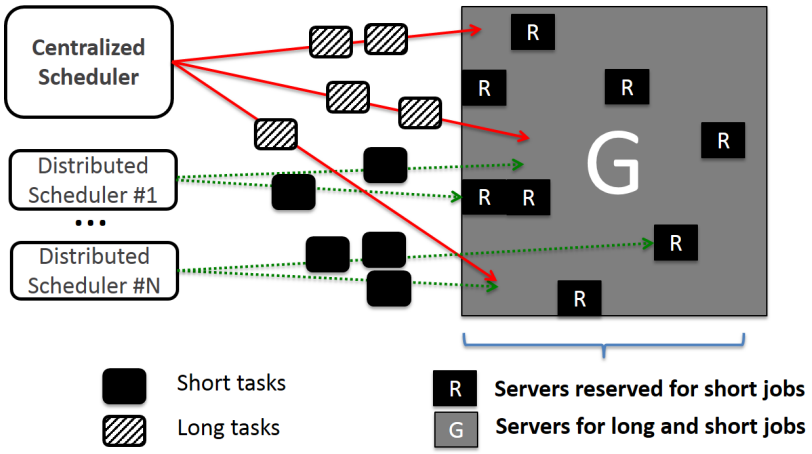


Figure 2.2 – Overview of job scheduling in Hawk.

loaded clusters,

3. to sustain or improve the performance for long jobs.

To meet these challenges, Hawk relies on the following mechanisms. To improve performance for short jobs, head-of-line blocking must be avoided. To this end, Hawk uses a combination of three techniques. First, it reserves a small part of the cluster for short jobs. In other words, short jobs can run anywhere in the cluster, but long jobs can only run on a (large) subset of the cluster. Second, to maintain low latency scheduling decisions, Hawk uses distributed scheduling of short jobs, similar to Sparrow. Third, Hawk uses randomized work stealing, allowing idle nodes to steal short tasks that are queued behind long tasks.

Finally, Hawk uses centralized scheduling for long jobs to maintain good performance for them, even in the face of reserving a part of the cluster for short jobs. The rationale for this choice is to obtain better scheduling decisions for long jobs. Since there are few long jobs, they do not overwhelm a centralized scheduler, and since they use a large fraction of the cluster resources, this centralized scheduler has an accurate view of the resource utilization at various nodes in the cluster, even if it does not know the location of the many short jobs. Figure 2.2 presents an overview of the Hawk scheduler.

2.3.3 Differentiating long and short jobs

The main idea behind Hawk is to process long jobs and short jobs differently. Two important questions are 1) how to compute a per-job runtime estimate, and 2) where to draw the line between the two categories.

Hawk uses an estimated task runtime for a job and computes it as the average task runtime for all the tasks in that job. This allows Hawk to easily classify jobs with variations in task

runtime [66] without having to deal with per-task estimates. Moreover, the average task runtime is relatively robust in the face of a few outliers tasks.

Hawk compares the estimated task runtime against a cutoff (threshold). The value of the cutoff is based on statistics about past jobs because the relative proportion of short and long jobs in a cluster is expected to remain stable over time. Jobs for which the estimated task runtime is smaller than the cutoff are scheduled in a distributed fashion. This estimation-based approach is grounded in the fact that many jobs are recurring [35] and compute on similar input data. Thus, task runtimes from a previous execution of a job can inform a future run of the same job [35].

2.3.4 Splitting the cluster

Hawk reserves a portion of the servers to run exclusively short tasks. Long tasks are scheduled on the remaining (large) part. Short tasks may be scheduled on the whole set of servers. This allows short tasks to take advantage of any idle servers in the entire cluster. Henceforth we use the term *short partition* to refer to the set of servers reserved for short jobs and the term *general partition* to refer to the set of servers that can run both types of tasks.

If long tasks were scheduled on any server in the cluster, this may severely impact short jobs when short tasks end up queued after long tasks. A particularly detrimental case occurs when a long job has more tasks than servers or when several long jobs are being scheduled in rapid succession. In this case, every server in the cluster ends up executing a long task, and short tasks have no choice but to queue after them.

Hawk sizes the general partition based on the proportion of time that cluster resources are used by long jobs. For example, from table 2.1 Hawk uses the percentage of task-seconds.

2.3.5 Scheduling short jobs

Hawk maintains low-latency scheduling for short jobs by relying on a distributed approach. Typically, each short job is scheduled by a different scheduler. For scalability reasons, these distributed schedulers have no knowledge of the current cluster state and do not interact with other schedulers or with the centralized component.

Distributed schedulers schedule tasks on the entire cluster. The first scheduling step is achieved as in Sparrow. To schedule a job with t tasks, a distributed scheduler sends probes to $2t$ servers. When a probe comes to the head of a server's queue, the server requests a task from the scheduler. If the scheduler has not given out the t tasks to other servers, it responds to the server with a task. Otherwise, a cancel is sent.

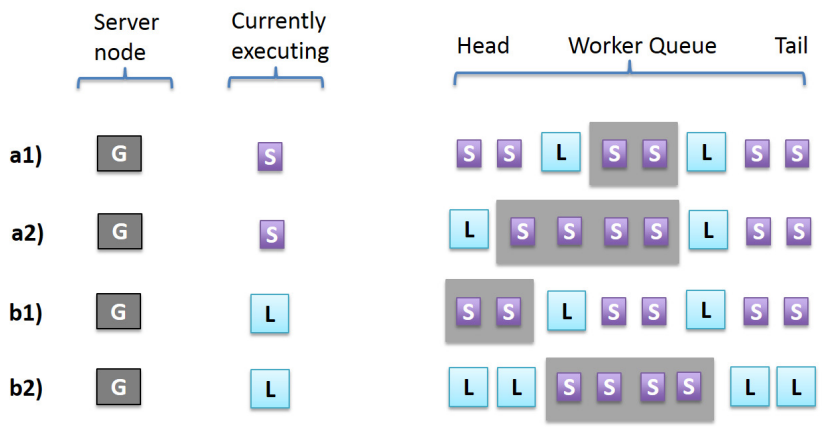


Figure 2.3 – Task stealing in Hawk. L = Long task, S = Short task. Stolen tasks are on the dark background.

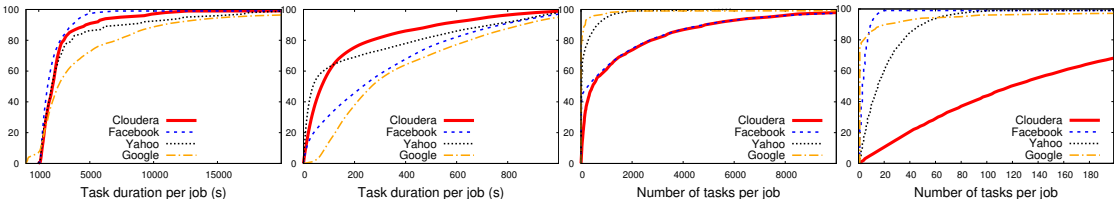
2.3.6 Randomized task stealing

Hawk uses task stealing as a run-time mechanism aimed at mitigating some of the delays caused by the occasionally suboptimal, distributed scheduling decisions. Since the distributed schedulers are not aware of the content of the server queues, they may end up scheduling short tasks behind long tasks. In a highly loaded cluster, the probability of this event happening is fairly high. Even if a short job is scheduled using twice as many probes as tasks, if more than half of the probes experience head-of-line blocking, then the completion time of the short job takes a big hit.

Hawk implements a randomized task stealing mechanism, that leverages the fact that the benefit of stealing arises in highly loaded clusters. In such a cluster a random selection very likely returns an overloaded server. Indeed, if 90% of the servers are overloaded, a uniform random probe has 90% probability of returning an overloaded server from which tasks are stolen.

The cluster might reach a point where many servers in the general partition are occupied by long tasks and also have short tasks in their queues, while other servers lie idle. Hawk allows such idle servers to steal tasks from the over-subscribed ones. This works as follows: whenever a server is out of tasks to execute, it randomly contacts a number of other servers to select one from which to steal short tasks. Both the servers from the general partition and the servers from the short partition can steal, but they can only steal from servers in the general partition, because that is where the head-of-line blocking is caused by long jobs.

Task stealing in Hawk proceeds as follow: The first consecutive group of short tasks that come after a long task is stolen. To see this in more detail, consider Figure 2.3. In cases a1) and a2) a server currently is executing a short job. The short tasks that it provides for stealing come after the first long job in the queue. In cases b1) and b2) the server is executing a long task. The short tasks stolen come immediately after that long task. Even though that long task is being



(a) Long jobs. (b) Short jobs. (c) Long jobs. (d) Short jobs.

Figure 2.4 – Workload properties. CDFs of average task duration and number of tasks per job.

executed already and has made some progress to completion, it is still likely that it will delay the short tasks queued behind it.

With our design we want to increase the chance that stealing actually leads not only to an improvement in task runtime but also in job runtime. Consider a job that has completed all but two of its tasks. Stealing just one of these tasks improves that task's runtime, but the job runtime is still determined by the completion time of the last task (the one not stolen). As shown in Figure 2.3, Hawk steals a limited number of tasks and starts from the head of the queue when deciding what to steal. Thus, stealing focuses on a few short jobs, increasing the chance that the runtime of those jobs benefits. If short tasks were stolen from random positions in server queues that would likely end up focusing on too many jobs at the same time while failing to improve most.

2.3.7 Scheduling long jobs

The final technique used in Hawk is to schedule long jobs in a centralized manner. Long jobs are only scheduled in the general partition, and the centralized component has no knowledge of where the short tasks are scheduled. This centralized approach ensures good performance for long jobs for three reasons. First, the number of long jobs is small, so the centralized component is unlikely to become a bottleneck. Second, long jobs are not latency-bound, so they are largely unaffected even if a moderate amount of scheduling latency occurs. Third, by scheduling long jobs centrally and by the fact that these long jobs take up a large fraction of the cluster resources, the centralized component has a timely and fairly accurate view of the per-node queueing times regardless of the presence of short tasks.

The centralized component keeps a priority queue of tuples of the form $< server, waiting\ time >$. The priority queue is kept sorted according to the waiting time. The waiting time is the sum of the estimated execution time for all long tasks in that server's queue plus the remaining estimated execution time of any long task that currently may be executing. When a new job is scheduled, for every task, the centralized allocation algorithm puts the task on the node that is at the head of the priority queue (the one with the smallest waiting time). After every task assignment, the priority queue is updated to reflect the waiting time increase caused by the job that is being scheduled. The goal of this algorithm is to minimize the job completion time

for long jobs.

2.3.8 Implementation

We implement Hawk as a scheduler plug-in for Spark [11], by augmenting the Sparrow scheduler with a centralized scheduler and work stealing. To realize work stealing we enable the Sparrow node monitors to communicate and send tasks to each other. The node monitors communicate via the Thrift RPC library.

2.4 Evaluation

We compare Hawk with Sparrow, a state-of-the-art fully distributed scheduler. We show that in loaded clusters Hawk outperforms Sparrow for both long and short jobs. The benefits hold across all workloads. We also show that Hawk compares well to a centralized scheduler.

2.4.1 Methodology

Workloads

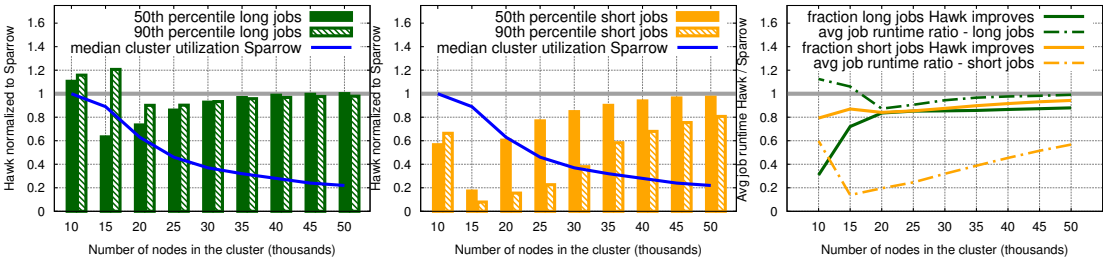
We use the publicly available Google trace [103, 82]. After removing invalid or failed jobs and tasks we are left with 506460 jobs. Task durations vary within a given job. The estimated task execution time for a job is the average of its task durations.

We create additional traces using the description of the Cloudera C and Facebook 2010 workloads from [20] and Yahoo 2011 workload from [21]. We only consider the mapper tasks from these workloads, since many jobs do not have reducers. In [20, 21] the workloads are described as k-means clusters, and the first cluster is deemed composed of short jobs. We consider the rest of the clusters to be long jobs. For each cluster we derive the centroid values for the average number of tasks per job and the duration of the tasks by combining the information on task-seconds from [20, 21] with the job to mapper duration ratios in [106]. We then use the derived centroid values as the scale parameter in an exponential distribution in order to obtain the number of tasks and the mean task duration for each job. Given the mean task duration we derive task runtimes using a Gaussian distribution with standard deviation twice the mean, excluding negative values.

Figures 2.4a, 2.4b, 2.4c and 2.4d show the CDFs of the duration of tasks and the number of tasks per job for both long and short jobs. Table 2.2 shows additional trace properties. The trace properties differ from trace to trace. This is expected, as workload properties are known to vary depending on the provider [5, 20, 21].

Workload	% Long Jobs	Total number jobs
Google	10.00%	506460
Cloudera-c	5.02%	21030
Facebook	2.01%	1169184
Yahoo	9.41%	24262

Table 2.2 – Percentage of long jobs and total number of jobs in workloads.



(a) Google trace long jobs. (b) Google trace short jobs. (c) Google trace additional metrics.

Figure 2.5 – Google trace. Hawk normalized to Sparrow. Figure (c) shows two additional metrics: (1) percentage of jobs for which Hawk is equal or better to Sparrow and (2) average job runtime.

Simulator

We augment the event-based simulator used to evaluate Sparrow [79]. The input traces contain tuples of the form: (jobID, job submission time, number of tasks in the job, duration of each task). Network delay is assumed to be 0.5ms. The scheduling decisions and the task stealing do not incur additional costs.

Real cluster run

We use a 100-node cluster with 1 centralized and 10 distributed schedulers. We use a subset of 3300 jobs from the Google trace. To obtain task runtimes proportional to the ones in the Google trace, we scale down task duration by 1000x (i.e., sec. to msec.) and use these durations in a sleep task. We also scale down the number of tasks per job by keeping constant the ratio between the cluster size and the largest number of tasks in a job. When we scale down the number of tasks in a job, we compensate by proportionally increasing the duration of the remaining tasks in order to keep the same task-seconds ratio as the original trace. We vary the cluster load by varying the mean job inter-arrival rate as a multiple of the mean task runtime. We use this mean to generate job inter-arrival times according to a Poisson distribution.

Parameters

By default, in Hawk, a node performs task stealing by randomly contacting 10 other nodes and stealing from the first node that has short tasks eligible for stealing. We compare against Sparrow configured to send two probes per task because the authors of Sparrow [79] have found two to be the best probe ratio. Each simulated cluster node has 1 slot (i.e., can execute only one task at a time). This is analogous to having multi-slot nodes with each slot served by a different queue. Following the task-second proportion between long and short jobs, the short partition comprises 17% of the nodes for the Google trace and 9%, 2% and 2% for the Cloudera, Facebook and Yahoo traces, respectively.

Metrics

When comparing Hawk to another approach X, we mostly take the ratio between the 50th (or 90th) percentile job runtime for Hawk and the 50th (or 90th) percentile job runtime time for X. Consequently, our results are normalized to 1. We do this separately for short and long jobs. Additional metrics are explained with the corresponding results. In all figures lower values are better.

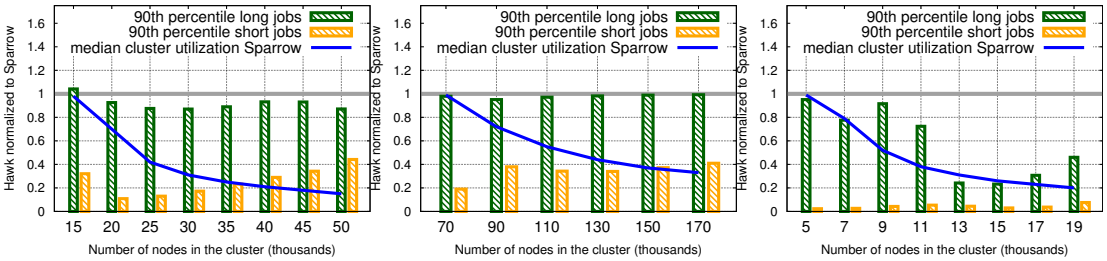
Repeatability of results

The results for the 50th and 90th percentiles are stable across multiple runs, and for this reason we do not show confidence intervals. We have seen variations in the maximum job runtime for short tasks. This is expected, as failing to steal one task can make a big difference in job runtime.

2.4.2 Overall results on the Google trace

We take the Google trace and vary the number of server nodes in order to vary cluster utilization. We find that Hawk consistently outperforms Sparrow, especially in a highly loaded cluster. Figures 2.5a and 2.5b illustrate the improvements in job runtime for long jobs and short jobs, respectively as a function of the number of machines in the cluster. The cluster utilization is based on snapshots taken every 100s.

Hawk shows significant improvements when the cluster is highly loaded but not overloaded (i.e., 15000 - 25000 nodes), since both the centralized scheduler and the task stealing algorithm make efficient use of any idle slots. In the best cases, Hawk improves the 50th and 90th percentile runtimes by 80% and 90% for short jobs and by 35% and 10% for long jobs. Hawk improves short job runtime at the 90th percentile more than at the 50th percentile, because these jobs are more affected by queueing. Stealing a few (even one) short tasks experiencing head-of-line blocking can greatly improve short job completion time.



(a) Cloudera trace. (b) Facebook trace. (c) Yahoo trace.

Figure 2.6 – Cloudera, Facebook and Yahoo traces. Long and short jobs. Hawk normalized to Sparrow.

Figure 2.5c presents additional metrics: the fraction of jobs for which Hawk provides performance better than or equal to Sparrow and the average job runtime for Hawk vs. Sparrow. The average job runtime for short jobs is significantly better for Hawk and is as low as a factor of 7. For 15000 nodes we present additional details, not all pictured: Hawk improves the runtime of 68% of short jobs, while for 59% of short jobs the improvement is more than 50%. Overall, for 86% of short jobs, Hawk is better or equal to Sparrow. For long jobs, Hawk improves 51% of jobs and is better or equal to Sparrow for 72% of jobs.

Small clusters (10000 nodes) tend to be overwhelmed by the high job submission rate in the trace. As a result, the node queues become progressively longer and waiting times keep increasing. We do not believe that any cluster should be run at this overload, but the case is nevertheless interesting to understand. Hawk is just slightly worse for long jobs, as the long jobs in Hawk are scheduled only in the general partition, while in Sparrow they can be scheduled across the entire cluster. Conversely, Hawk is better for short jobs because of the randomized stealing, but the improvement is small. The short partition is overloaded, and its nodes have few opportunities to steal short tasks experiencing head-of-line blocking in the general partition. As the cluster size increases (40000+ nodes), the benefits of Hawk decrease as the cluster becomes mostly idle. Any scheduler is likely to do well in that case.

2.4.3 Overall results on additional traces

Figures 2.6a, 2.6b and 2.6c show the results for the workloads derived from Facebook, Cloudera and Yahoo data. Hawk's benefits hold across all traces. At the median (not pictured), Hawk also improves on Sparrow across all simulated cluster sizes.

The most important difference compared to the Google trace is the larger improvement for short jobs. This can be traced back to the utilization of the short partition. In the Facebook, Cloudera and Yahoo traces the short partition is less utilized compared to the Google trace so there are more chances for stealing.

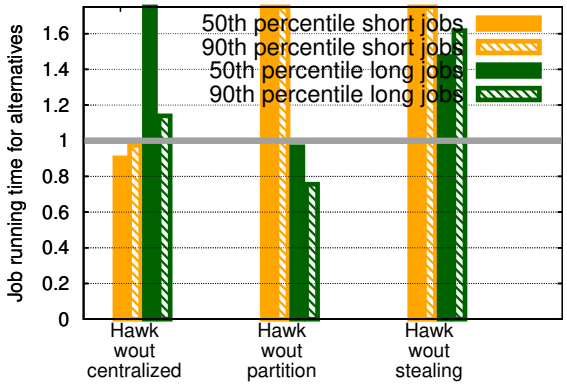


Figure 2.7 – Break-down of Hawk's benefits normalized to Hawk. 15000 nodes. Google trace.

2.4.4 Breaking down Hawk's benefits

This subsection analyzes the impact of each of the major components of Hawk: work stealing, reserving cluster space for short jobs and using centralized scheduling for the long jobs. We find that the absence of any of the components reduces the performance of Hawk for either long or short jobs.

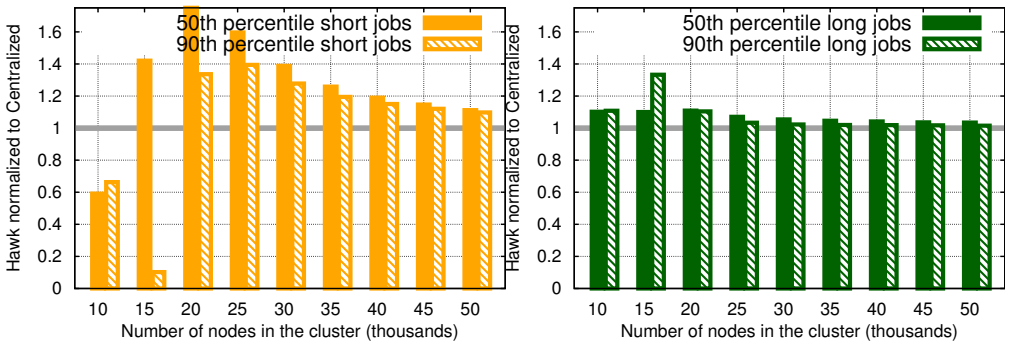
Figure 2.7 shows the results of the Google trace normalized to Hawk with all components enabled. Without centralized scheduling for long jobs the performance of long jobs takes a significant hit, as tasks of different long jobs queue one after the other. The performance of short jobs improves due to the decrease in the performance for long jobs. As the placement of long jobs is not optimized in the general partition, fewer short tasks encounter queueing there.

Without partitioning the cluster, the short jobs are impacted, because they can be stuck behind long tasks on any node. For long jobs, the performance slightly increases, because they can be scheduled on more nodes. Without task stealing both short and long jobs suffer. The short jobs are greatly penalized, because some of their tasks are stuck behind long tasks. The long tasks are penalized, because they share the queues with more short tasks.

2.4.5 Hawk vs. a fully centralized approach

We next look at the performance of Hawk compared to an approach that schedules all jobs (long and short) in a centralized manner. We find that Hawk is competitive, while not suffering from the scalability concerns that plague centralized schedulers.

This centralized scheduler does not reserve part of the cluster for short jobs and does not use work stealing. It uses the algorithm we presented in subsection 2.3.7 for all jobs. Figures 2.8a and 2.8b show Hawk normalized to the centralized scheduler's performance using the Google trace.



(a) Short jobs.

(b) Long jobs.

Figure 2.8 – Hawk normalized to a centralized approach. Google trace.

The centralized scheduler penalizes short jobs (Figure 2.8a), when the cluster is heavily loaded (10000-15000 nodes). This is because in periods of overload the centralized scheduler does not have many options and queues short tasks behind long ones. This is especially the case when long jobs are present in every node in the cluster. In Hawk short tasks benefit from stealing and from running on reserved nodes. As the cluster utilization decreases, the centralized scheduler does an increasingly better job for short jobs. When the cluster becomes lightly loaded (50000 nodes), the results for both approaches begin to converge.

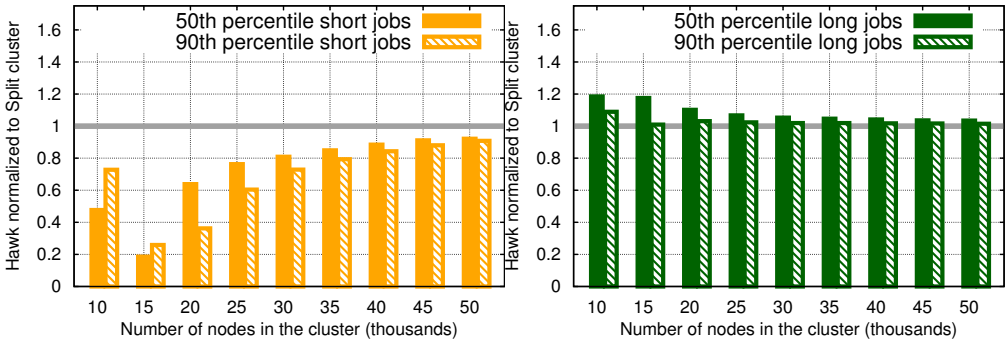
For long jobs the centralized approach performs slightly better (Figure 2.8b), because they can use the entire cluster. In Hawk they only use the general partition.

2.4.6 Hawk compared to a split cluster

We now compare Hawk to a split cluster, in which a long partition only runs long jobs and a short partition only runs short jobs. In other words, there is no general partition, in which both short and long jobs can execute. Hawk fares significantly better for short jobs, while being competitive for long jobs.

We use the Google trace. The split cluster uses 17% of the cluster for the short partition, and the remaining 83% is reserved for long jobs (long partition). The split cluster uses centralized scheduling for the long partition and distributed scheduling for the small one.

Figures 2.9a and 2.9b show the results. For long jobs, the split cluster performs slightly better, because the short jobs do not take up the space in the general partition. However, this comes at the cost of greatly increasing runtime for short jobs. For short jobs, for small cluster sizes, the relative degradation for the split cluster is smaller, because both approaches suffer from significant queueing delays. In the other extreme, for a large cluster, both approaches do well. In between, the split cluster shows extreme degradation, because short tasks cannot leverage the general partition nodes.



(a) Short jobs.

(b) Long jobs.

Figure 2.9 – Hawk normalized to split cluster. Google trace.

(a) Short jobs.

(b) Long jobs.

Figure 2.10 – Effect of varying cutoff, Hawk normalized to Sparrow. 15000 nodes. Google trace.

2.4.7 Sensitivity to the cutoff threshold

Next we vary the cutoff point between short and long jobs. Hawk yields benefits for a range of cutoff values, showing that it does not depend on the precise cutoff chosen.

The cluster size is 15000 nodes in this experiment, and we use the Google trace. Figures 2.10b and 2.10a show the results for long and short jobs, respectively. The percentage of short jobs increases as the cutoff increases. Thus, for the smaller cutoffs, Hawk improves the most on Sparrow because the short partition is underloaded and can steal more tasks. The percentage of long jobs increases as the cutoff decreases. For the smaller cutoffs the 90th percentile long job runtime is affected more for Hawk compared to Sparrow, because Sparrow is able to relieve some of the queueing among long jobs by scheduling them over the entire cluster.

2.4.8 Sensitivity to task runtime estimation

Hawk's centralized component schedules long jobs according to an estimate of the average task runtime for that job. We next analyze how inaccuracies in estimating the average affect

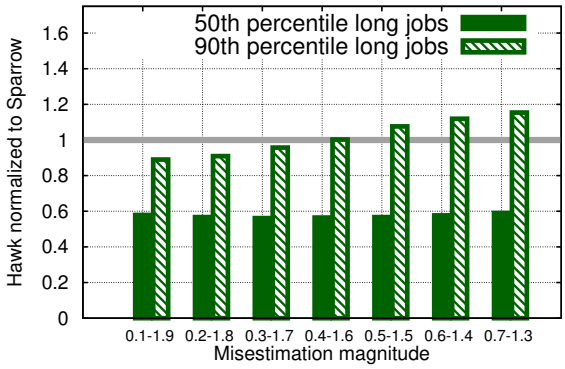


Figure 2.11 – Hawk with varying mis-estimation magnitude normalized to Sparrow, long jobs. 15000 nodes. Google trace.

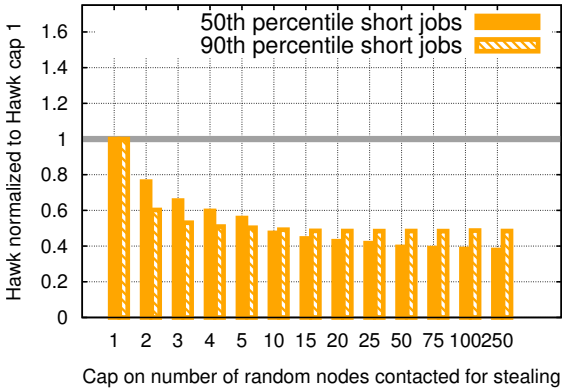
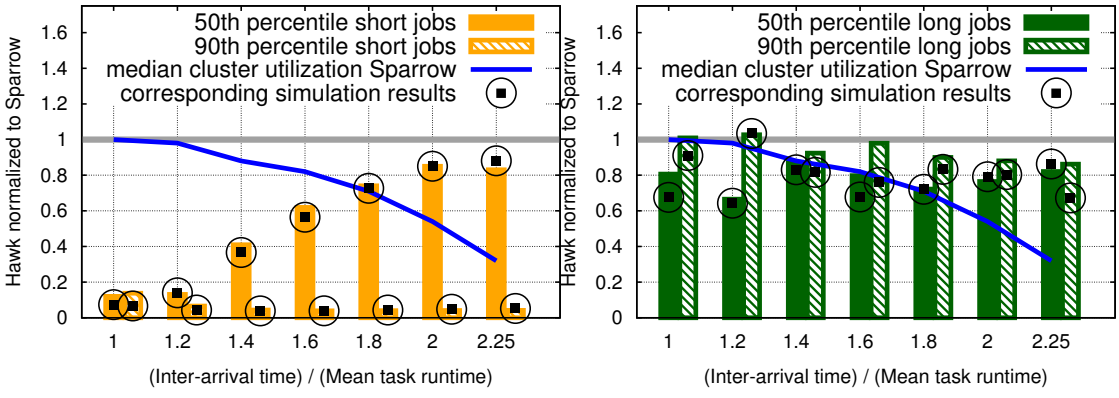


Figure 2.12 – Hawk with varying number of stealing attempts normalized to Hawk capped at 1 attempt, short jobs. 15000 nodes. Google trace.

the results. For each job, to obtain the inaccurate estimate, we multiply the correct estimate with a random value, chosen uniformly within a range given as a parameter (e.g., 0.1-1.9). Figure 2.11 shows the job runtimes normalized to Sparrow for the set of jobs classified as long when no mis-estimations are present. These results are averaged over ten runs.

Hawk is robust to mis-estimations. The mis-estimation results in some long jobs being classified as short and vice-versa. This is more likely to happen for long and short jobs for which the estimation is comparable to the cutoff. Since these jobs are fairly similar in nature, the two opposing mis-classifications (long as short and short as long) tend to cancel each other. Moreover, most jobs are not mis-classified, because their estimation significantly differs compared to the cutoff. In Figure 2.11, long jobs perform better at the 90th percentile as the mis-estimation magnitude increases because more long jobs are classified as short. At 15000 nodes the short partition is less loaded than the general partition so the long jobs classified as short benefit from the additional, less-loaded nodes in the short partition.

Short jobs are not directly impacted by mis-estimations, since their scheduling does not rely on estimations. Short jobs can be indirectly impacted by the changes in the scheduling of the



(a) Short jobs.

(b) Long jobs.

Figure 2.13 – Implementation vs simulation. 3300 job sample from the Google trace.

long jobs. In the experiments, we only see minute variations for the results for short jobs (not pictured).

2.4.9 Sensitivity to stealing attempts

We now vary the maximum number of nodes that an idle node can contact for stealing. We find that performance increases with an increase in the cap value, but even a low value (e.g., 10) gives significant benefit.

Figure 2.12 shows the results normalized to Hawk using a cap of 1. As expected, increasing the cap also increases performance, as it increases the chance for successful stealing. At high cap values there is also a slight increase in the performance of long jobs (not pictured), because they wait behind fewer short tasks. The improvement for long jobs is small, because of the large relative difference between the resource usage of long jobs compared to short jobs.

2.4.10 Implementation vs. simulation

Figures 2.13a and 2.13b show the results for a 3300-job sample of the Google trace. In the implementation, Hawk schedules 3000 short jobs in a distributed way (300 per each of the 10 distributed schedulers) and 300 long jobs in a centralized fashion. The simulation and implementation experiments agree and show similar trends. Hawk is best at high loads, when it significantly improves on Sparrow for short jobs, while maintaining good performance for long jobs. As load decreases, the 50th percentiles for Hawk and Sparrow become similar, as fewer jobs suffer from queueing. Even at medium load, the 90th percentile is still considerably better for Hawk for short jobs, since those jobs suffer from queueing in Sparrow but not in Hawk.

The simulation and implementation results do not perfectly match, because the simulation

does not model overheads for scheduling or stealing. Moreover, some Spark tasks sleep very little (a few msec) and are sensitive to slight inaccuracies in sleeping time and to various system overheads (message exchanges, network delays).

2.5 Summary

In this chapter we address the problem of efficient scheduling in the context of highly loaded clusters and heterogeneous workloads composed of a majority of short jobs and a minority of long jobs that use the bulk of the resources. We propose Hawk, a hybrid scheduling architecture. Hawk schedules only the long jobs in a centralized manner, while performing distributed scheduling for the short jobs. To compensate for the occasional poor choices made by distributed job scheduling, Hawk uses a novel randomized task stealing approach. With a Spark-based implementation and with large scale simulations using realistic workloads we show that Hawk outperforms Sparrow, a state-of-the-art fully distributed scheduler, especially in the challenging scenario of highly loaded clusters.

3 Job-Aware Scheduling in Eagle: Divide and Stick to Your Probes

"A chain is no stronger than its weakest link."
— proverb

3.1 Introduction

Datacenter scheduling is a challenging problem for a variety of reasons. The first issue is the heterogeneity of the workload. A typical workload consists of long and short jobs. The long jobs tend to be latency-insensitive and, while small in number, they consume the bulk of the resources. Vice versa, there are many short jobs, they are latency sensitive, but consume only limited resources [82, 29]. Therefore, the scheduler has to take care to avoid head-of-line blocking, i.e., placing a short job behind a long one, especially under high load. The second issue is the parallel nature of the jobs: the overall completion time of a job is equal to that of its slowest task. Therefore, the scheduler has to be job-aware, considering all tasks of a job rather than individual tasks in isolation. The final issue stems from the scale of the datacenter. A very large number of jobs must be scheduled on a very large number of nodes. At this scale, centralized schedulers can exhibit high scheduling latency [100], and as a result distributed [79, 81] or hybrid centralized/distributed [60, 29] schedulers have been developed.

This chapter introduces a new hybrid scheduler, called Eagle. Eagle divides the datacenter's nodes in partitions for the execution of short and long jobs to avoid head-of-line blocking, and introduces sticky batch probing to achieve better job-awareness. We describe these techniques next, motivating them by fundamental results from queueing theory.

The Pollaczek-Khinchine formula states, under rather general conditions, that the expected completion time of jobs served by a node is proportional to the variance of the job execution times [64]. This observation has led to so-called Size Interval Task Assignment (SITA) scheduling policies, that statically divide compute nodes into different partitions for executing jobs of different lengths [48]. SITA as such is not practical in a datacenter because variations over

time in the resource demands of long and short jobs make a static division inefficient. Instead, we develop a distributed and dynamic variant of SITA, by providing the schedulers for short jobs with information about where long jobs are currently executing, through a technique we call Succinct State Sharing (SSS).

Furthermore, Little's law states, again under rather general conditions, that the expected completion time of a job is inversely proportional to the number of jobs present in the system [68]. To optimize job completion times, one must therefore optimize the rate at which entire jobs leave the system, and avoid that straggler tasks delay job completion. Eagle introduces Sticky Batch Probing (SBP) to deal with this problem. In contrast to conventional probe-based schedulers [79, 29], in SBP a probe does not represent a single task, but rather the whole job. A probe can trigger the execution of as many tasks of a job as required to prevent stragglers. In combination with SBP, Eagle implements a variant of the Shortest Remaining Processing Time (SRPT) scheduling policy [48], which further improves job-awareness and job completion times.

We evaluate Eagle through simulation on datacenter traces from Cloudera, Facebook and Google, and through implementation and measurement on a cluster with 100 nodes. Eagle compares favorably to earlier datacenter schedulers. In particular, we demonstrate that it does better at avoiding head-of-line blocking than other probe-based schedulers that rely on work stealing [29]. Furthermore, we show that Eagle's distributed job-awareness provides better completion times than schedulers relying on local job-awareness [81]. Finally, we quantify Eagle's superior robustness against misestimations of job execution times.

Contributions. The key contributions of this chapter are:

1. A technique for dividing a datacenter, dynamically and in a distributed fashion, into partitions for executing long and short jobs, thereby reducing head-of-line blocking.

2. A technique for bringing job-awareness to distributed scheduling in a datacenter.

3. The implementation and evaluation, through simulation and implementation, of a hybrid datacenter scheduler, Eagle, that embodies these techniques.

The outline of the rest of this chapter is as follows. Section 3.2 introduces the system and the workloads targeted by Eagle. Section 3.3 shows how Eagle avoids head-of-line blocking using SSS. Section 3.4 shows how SBP provides job-awareness in Eagle. Section 3.5 describes the evaluation methodology. Section 3.6 presents experimental results obtained by means of extensive trace-driven simulations. Section 3.4.3 presents experimental results obtained by deploying and running a prototype of Eagle. Section 3.8 concludes the chapter.

3.2 System Model

We consider a datacenter composed of worker nodes. A job consists of a set of tasks that can run in parallel on different workers. Scheduling a job requires assigning every task of a job to a worker. When a new task is scheduled on an idle worker, the task starts executing immediately. When there is already a task running on the worker, the new task is appended to the queue on the worker.

The completion time of a task is the time from the submission of the job that contains the task to the time when the task finishes execution. A job completes when all of its tasks finish. The job completion time is then the maximum of the task completion times of all its tasks. The scheduling time of a task is the time between the submission of the job of which that task is part until the time the task is queued at a worker. The queueing time of a task is the time the task spends in the queue (zero if the worker is idle when the task is assigned). The execution time of a task is the time the job spends running. The task completion time is the sum of the scheduling time, the queueing time and the execution time. The execution time of a job is the sum of the execution times of its tasks.

Consistent with observations made in many papers about datacenter workloads [62, 82, 29], we assume that the workload consists of a small number of long jobs that consume a large fraction of the datacenter's resources, and a large number of short jobs that consume only a small fraction of the datacenter's resources. We refer to long (short) tasks as the tasks of a long (short) job.

Similarly to other schedulers [16, 29, 81], Eagle leverages the availability of estimated tasks execution times for an incoming job. The estimated task execution time for a given job is computed as the average execution time across all the tasks in a given job. A job is classified as long (short) if the average execution time for its tasks falls above (below) a given threshold. The rationale underlying this approach for identifying long and short jobs is grounded in the fact that jobs in modern datacenters are typically recurring [35, 25], and execute against similar input data. This allows Eagle to compute task runtime estimates looking at previous executions of the same job. Similarly, the relative proportion of short and long jobs is expected to remain stable over time. This enables the implementation of a simple yet accurate threshold-based classification of short vs long jobs.

3.3 Divide

3.3.1 Design

The problematic situation for short tasks in a datacenter highly loaded with long tasks is the so-called head-of-line blocking: a short task is enqueued behind a long task (either in the queue or running) and has to wait a long time to run. Since the majority of resources in typical datacenter workloads is taken up by long jobs, head-of-line blocking is one of the main causes

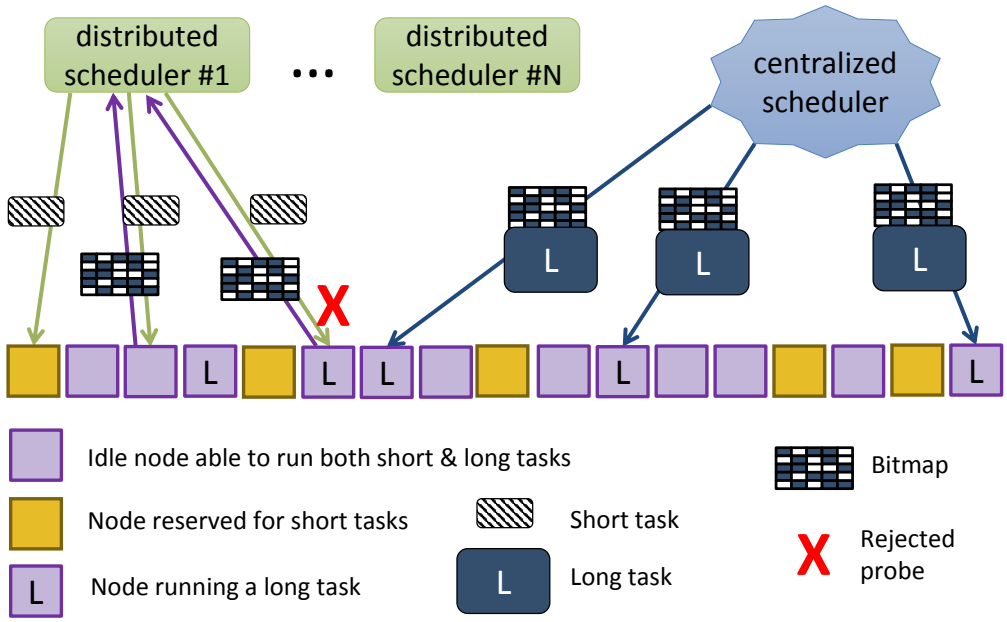


Figure 3.1 – Overview of Eagle and Succinct State Sharing.

of poor performance for short, latency-sensitive tasks [29].

Eagle provides a solution to the head-of-line blocking problem by means of a novel approach grounded in queueing theory. The Pollaczek-Khinchine formula states that the average completion time of the jobs executed by a node is proportional to the variance in the job execution times [64]. This result implies that, in a datacenter, reducing the variability of the execution times of jobs assigned to a single node yields a reduction in their average completion time.

SSS reduces the variability in the execution times of tasks assigned to nodes by enforcing that a short task is never enqueued behind a long one. SSS has a static and a dynamic component. The datacenter is statically split into two partitions. The smaller of the two, referred to as the *short* partition, is reserved for short jobs. The bigger one, referred to as the *general* partition, is primarily dedicated to long jobs, but may on occasion execute short jobs, guided by the dynamic component of SSS, as follows. SSS informs the short jobs schedulers in a low-overhead fashion about the placement of long jobs in the general partition, allowing them to opportunistically place short tasks on nodes in the general partition that are not currently serving a long job.

SSS achieves two principal goals: (i) it completely eradicates the head-of-line blocking problem, by avoiding short tasks to be enqueued behind long ones, and (ii) it achieves high resource utilization by dynamically allowing short jobs to run in the general partition.

3.3.2 Benefits Over Previous Designs

Head-of-line blocking is a primary concern for datacenter schedulers, and a number of solutions have been proposed to address it.

The Hawk scheduler [29] has separate schedulers for short and long jobs and reserves a small portion of the datacenter for short jobs. Hawk, however, allows short tasks to be enqueued behind long ones, to prevent resource under-utilization. To compensate for the resulting head-of-line blocking, Hawk implements randomized work stealing. When a node is idle, it contacts some nodes at random to steal probes that are enqueued behind a long task. As we shall show in Section 3.6, work stealing, as implemented in Hawk, only partially removes head-of-line blocking.

Mercury [60] mitigates head-of-line blocking by means of load shedding. In more detail, tasks from overloaded nodes are periodically relocated to underloaded nodes. Similarly to work stealing, load shedding does not operate at the scheduling level, but it is a runtime correction mechanism.

Hawk and Mercury only partially avoid head-of-line blocking. Some tasks may execute on a node after having experienced head-of-line blocking. Even if they are relocated, they may have waited behind a long task. SSS, instead, pro-actively eradicates the head-of-line blocking problem by avoiding that a short task ever gets enqueued behind a long one.

The Pollaczek-Khinchine formula is also at the basis of the so called Size Interval Task Assignment (SITA) scheduling policies [50, 48]. According to such policies, each node i in a datacenter serves only jobs whose estimated execution time falls in a specific range $[S_i, S'_i]$. A limitation of SITA policies, however, is their *static nature*. It is possible for a subset of the nodes to be temporarily (over)loaded, while having other nodes, assigned to other execution time ranges, idle or under-loaded [49, 50, 48]. SSS leverages the Pollaczek-Khinchine formula in a similar fashion to SITA policies. Unlike SITA policies, however, SSS preserves high resource utilization by dynamically and opportunistically allowing short tasks to occupy worker nodes in the general partition.

3.3.3 Implementation

Eagle is a hybrid scheduler, in which a centralized scheduler handles long jobs and distributed schedulers handle short jobs [29]. The rationale for this design is the following. Long jobs are relatively few, but consume the bulk of the resources, so their centralized scheduling allows for a good placement of the most demanding jobs, while not introducing a scalability bottleneck. The distributed scheduling of short jobs enables their placement on worker nodes with low latency and in a scalable fashion. Figure 3.1 shows the hybrid nature of Eagle and provides an overview of SSS.

The centralized scheduler implements the Least Work Left (LWL) scheduling policy [48] to

place long jobs on nodes in the general partition, as in Hawk [29]. This policy places each task on the node corresponding to the smallest expected queueing time for such task.

When the centralized scheduler assigns a (long) task to a worker node, it piggybacks on the message a timestamp and a succinct copy of its state, consisting of a bitvector of length equal to the number of workers, with bit i indicating whether or not worker i currently has a long task assigned to it (either running or enqueued). The worker node stores this bitvector, together with the timestamp received in the message. The arrival of a new long task causes the old bitvector and timestamp to be replaced with the newly received values.

The distributed schedulers are based on probing [73, 79]. For a job with t tasks, a distributed scheduler sends probes to $max\{K, 2t\}$ worker nodes, with K being a tunable parameter. A distributed scheduler sends a minimum of K probes to improve the completion times of short jobs with very few tasks. Otherwise, such jobs would result in a very small number of probes being sent, reducing the likelihood that at least one probe lands on an unloaded node.

A distributed scheduler selects the targets of probes for a given job uniformly at random among all nodes. A probe can reach a node N to which a long task is currently assigned. In this case, N rejects the probe and responds to the distributed scheduler with its bitvector and corresponding timestamp.

The distributed scheduler then re-schedules the rejected probes. To do so, it uses the freshest available bitvector to identify the set of nodes to which currently no long jobs are assigned. For the re-scheduling phase, target nodes are drawn uniformly at random from the set of nodes that are not currently serving a long job according to the selected bitvector. Some probes might be rejected again, due to stale bitvectors or the concurrent arrival of long jobs. Probes rejected during re-scheduling are assigned uniformly at random to nodes in the short partition.

Probes are scheduled at first by contacting nodes uniformly at random, even if the distributed scheduler already has a bitvector available. We have experimentally verified that this design leads to better results than using an old bitvector in the first scheduling attempt, because sampling nodes at random gives a better approximation of the current utilization of the datacenter than a possibly stale bitvector.

3.4 Stick to Your Probes

3.4.1 Design

A key characteristic of data-parallel jobs is that a job completes when all its tasks finish. The overall completion time of a job is therefore equal to that of its slowest task.

Little's law, a fundamental result in queueing theory [68], states that, given an arrival rate of jobs to a system, the average job completion time is inversely proportional to the number of

jobs in the system.

Applied to the data-parallel jobs case, Little's law indicates that a scheduler needs to optimize the completion time of jobs as a whole, and not the completion time of their individual tasks. In other words, a good scheduling policy must be job-aware.

Eagle uses LWL to schedule long jobs. Therefore, the centralized scheduler in Eagle places long tasks on nodes aiming to optimize the completion time of the whole job.

Eagle introduces SBP to provide job-awareness for the distributed scheduling of short jobs.

In SBP, a probe does not represent a single task of a job but a whole job. In other words, a single probe can lead to the execution of multiple tasks of the corresponding job.

When a task of a job J finishes at a given worker node, rather than relinquishing the node to the next task in the local queue, the worker may contact the distributed scheduler of J to request another task of J. In this way worker nodes are allowed to quickly remove all tasks of a job from the system once that job starts, thus avoiding stragglers.

SBP implements the latest possible form of task-to-node binding, by assigning a task to a node only when the node has available resources. If more nodes storing a probe for J have available resources, more tasks of J can be executed in parallel. Thus, SBP gracefully adapts the degree of parallelism of jobs execution to resources availability.

The following example shows how SBP augments probing with job-awareness. Suppose we have a datacenter with 4 nodes, with queue lengths of 100 at nodes $n1$, $n2$, and queue lengths of 10 at nodes $n3$ and $n4$. We have to schedule a job with four tasks, each with duration 10. One probe lands on each node, making the expected execution time of the overall job 110. With SBP, instead, at time 20 node $n3$ and $n4$ are able to pull from the distributed scheduler another task each, thus achieving a job completion time of 30.

SBP does not restrict worker nodes to process probes in FIFO order. It is well known that SRPT achieves optimal average completion time by executing the job with smallest remaining execution time first (in the single-task job scenario with preemption) [67, 48]. Aiming to further reduce short job completion times, Eagle implements an approximate variant of the SRPT scheduling policy on top of SBP. This variant does not need support for preemption, works for data-parallel jobs, and is augmented with an anti-starvation measure.

SBP can only implement the approximated variant of SRPT, as it would be implemented by a centralized scheduler, because of its probing-oriented nature. If J is the job to run next according to SRPT, J can only be executed on nodes which host a probe for it. Therefore, even if some other node has available resources, J cannot be run there.

3.4.2 Benefits Over Previous Designs

We now show the limitations of existing scheduling systems in implementing job-awareness.

The work stealing implemented by Hawk, being randomized, is not job-aware. With reference to the previous example, node $n4$, once idle, could in vain try to steal a probe from $n3$, and vice versa. In general, a stealing attempt can fail, or it can target a job with no straggler tasks. Mercury is job-unaware as well, and its load shedding technique re-balances queues based only on their backlogs and not on the job status of enqueued tasks.

Yaq [81] implements job-awareness by supporting different local queue reordering policies. Yaq performs early binding of tasks to nodes, thus re-introducing the issue of straggler tasks. We refer to the example used in 3.4.1, where nodes $n1$, $n2$, $n3$ and $n4$ have queue lengths of 100, 100, 10 and 10 respectively. Supposing that the queue length for $n1$ and $n2$ is mis-estimated to be 10 instead of 100, a scheduler in Yaq would place the four tasks of the example job, one in every node. With a task execution time of 10, this would result in an actual completion time of 110. In Eagle, assuming the initial scheduling is the same, nodes $n3$ and $n4$ will execute the two straggler tasks enqueued at $n1$ and $n2$ at time 20, resulting in an actual completion time of 30 instead.

In general, Eagle's SBP is able to pull tasks to the "best" node on which a probe is located, while Yaq may, as a result of misestimation, locate a task on a less desirable node, without any possibility of recovering from that choice other than performing job-unaware load shedding.

3.4.3 Implementation

We only discuss the scheduling of short jobs, as they are the only jobs affected by SBP. To further simplify the presentation, we first explain an SBP implementation in connection with FIFO scheduling. We then augment that implementation with Eagle's variant of SRPT, and complete the description with the anti-starvation measure.

In the case of FIFO, when a probe P arrives at the head of the queue on node N, N contacts the distributed scheduler of P. The scheduler replies to N with one task T of the job J corresponding to P. N does not remove the selected probe from the queue. Once T terminates its execution, N requests another task of J, until all tasks of J are executed.

In the case of SRPT, N selects from its queue the probe to run according to SRPT instead of according to FIFO. In order to make this selection, N needs to know the remaining execution times for the jobs for which it has probes in its queue. To this end, when a distributed scheduler sends a probe P to N, it also communicates the number of tasks composing the job and their expected average execution time. Upon assigning a new task to a node, the distributed scheduler updates the number of remaining tasks at all nodes where it has located probes. This information suffices for N to decide which probe to select according to SRPT.

Algorithm 1 Sticky Batch Probing + SRPT

```
1: procedure MAINLOOP
2:     while (true) do
3:         task ← GetNextTaskToExecute()
4:         ExecuteTask(task)
5:         Send(task.scheduler, finishedTask, task.id)
6: procedure GETNEXTTASKTOEXECUTE
7:     p ← GetProbeFromQueue()
8:     if (p.isLong) then
9:         queue.pop(p)
10:        return p.task
11:    else
12:        reply ← Send(p.scheduler, getTask)
13:        return reply.task
14: procedure GETPROBEFROMQUEUE
15:    shortest ← infinite
16:    chosen ← void
17:    for p in queue do
18:        if (p.isLong) then
19:            break
20:        if (p.estJobLeftRuntime < shortest) then
21:            if (CanBypass(p.estTaskRuntime)) then
22:                shortest ← p.estJobLeftRuntime
23:                chosen ← p
24:    if (chosen == void ∧ queue.size() > 0) then
25:        return queue.first
26:    else
27:        return chosen
```

Because SRPT does not respect FIFO order, starvation can occur: a probe that has joined N's queue at time t can be bypassed indefinitely often by tasks whose probes have joined N's queue at a time $t' > t$. To prevent starvation, whenever a probe P is about to be bypassed by a task T, P's "bypass counter" is incremented by the estimated execution time of T.

Eagle does not allow T to bypass P if the estimated execution time of T would make the bypass counter of P exceed a threshold value.

In summary, when choosing the next short task to execute, N picks the probe corresponding to the enqueued job with the shortest remaining execution time that is allowed to bypass all the probes in front of it. If there is no such probe, then N selects the next probe in FIFO order.

The pseudocode in Algorithm 1 depicts this protocol, reporting the operations performed by a worker node. For simplicity, the initial scheduling of probes and the message exchanges to update the remaining execution time of the jobs are not shown.

The main loop ran by a worker node is embedded in the function `MainLoop`, which performs three main steps: (i) retrieval of the next task to run (Line 3), (ii) execution of the task (Line 4) and (iii) notification of task completion to the distributed scheduler (Line 5).

The function `GetNextTaskToExecute` implements SBP. It first invokes the `GetProbeFromQueue` function to determine whether the next task to execute belongs to a long or to a short job. In the former case, `GetNextTaskToExecute` removes the corresponding placeholder from the queue (Line 9). In the latter case, the function contacts the distributed scheduler corresponding to

(a) Short jobs. Cloudera trace. (b) Short jobs. Google trace. (c) Short jobs. Yahoo trace.

(d) Long jobs. Cloudera trace. (e) Long jobs. Google trace. (f) Long jobs. Yahoo trace.

Figure 3.2 – Eagle normalized to Hawk. 50/90/99th percentiles of job completion times.

the returned probe to pull a new task (Line 12).

The `GetProbeFromQueue` function embeds the logic of Eagle's job-aware policy. If the worker is in the general partition and there are no probes, the function picks a long task from the queue in FIFO order. Otherwise, the function implements Eagle's variant of SRPT described before and returns the next short task to be executed.

3.5 Evaluation Methodology

We evaluate Eagle using the Google workload trace as the primary workload [82], and we additionally use traces from Cloudera [20] and Yahoo [21]. A detailed description of these traces can be found in [29]. For completeness, Table 3.1 shows for each of the three traces the total number of jobs, the percentage of long jobs and the percentage of task-seconds for long jobs.

We assess Eagle's effectiveness by means of a twofold methodology. We evaluate, against Hawk, a prototype integrated in Spark [107], on a 100-node deployment running a subset of the Google trace. In addition, we provide simulation results for larger clusters and for all the traces, comparing with other scheduling policies.

We vary the number of worker nodes to simulate both high and low load conditions. Each worker node has one queue. The size of the small partition is 17, 9 and 2 percent of the nodes, for the Google, Cloudera and Yahoo traces, respectively. These numbers correspond to the percentage of the execution times (task-seconds) of all short jobs in the respective traces. The

Trace	Total # jobs	% long jobs	% task-seconds long jobs
Cloudera [20]	21030	5.02	91
Yahoo [21]	24262	9.41	98
Google [82]	506460	10.00	83

Table 3.1 – 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.

choice of these values is aimed at allocating an amount of worker nodes to short jobs that is proportional to their computational demands.

We set the network delay to 0.5 milliseconds, and we do not assign any cost to making scheduling decisions. To prevent starvation, as specified in Section 3.4.3, we set the starvation threshold value to 5 times the estimated duration of a single task in a job, for all experiments involving SRPT. In all the Eagle experiments, unless otherwise stated, the minimum number of probes sent per job is 20.

We use as main metrics the 50th, 90th and 99th percentiles of the job completion time distribution. The results are averages over 5 runs. We do not plot error bars, since the results of different simulations are consistent across runs.

3.6 Simulation Results

In Section 3.6.1 we evaluate the benefits of Eagle against Hawk, a state-of-the-art hybrid scheduler. Next, in Section 3.6.2, we compare Eagle against a Distributed Least Work Left (DLWL) scheduler with node-local Shortest Remaining Processing Time (SRPT) queue reordering. In Section 3.6.3 we show a breakdown of Eagle's components. A comparison against an omniscient centralized scheduler is shown in Section 3.6.4. Finally, Section 3.6.5 includes an analysis of Eagle's robustness to misestimations.

3.6.1 Comparing Eagle Against Hawk

We compare the job completion time distributions for short and long jobs between Hawk and Eagle for the three considered traces. Work stealing in Hawk is configured according to the default settings [29]: idle nodes in the general partition perform ten attempts to steal probes from other nodes in that partition. If successful, they steal the first batch of short tasks that are enqueued behind a long one.

Figure 3.2 (top) shows, as a function of the number of worker nodes, the 50th, 90th, and 99th percentiles of the job completion time distribution for short jobs for Eagle, normalized to those same values in Hawk. In addition, we report the average node utilization with the given number of nodes. Figure 3.2 (bottom) reports the same results for long jobs.

Trace (# nodes)	# probes behind long task		# short tasks exec after long		# re-scheduled probes	
	Hawk	Eagle SSS	Hawk	Eagle SSS	Hawk (steal)	Eagle SSS
Google (15K)	17.30M	0	0.63M	0	15.70M	16.98M
Yahoo (4K)	0.87M	0	18K	0	0.80M	0.86M
Cloudera (15K)	5.60M	0	55K	0	5.50M	5.65M

Table 3.2 – Head-of-line blocking statistics: Eagle vs Hawk.

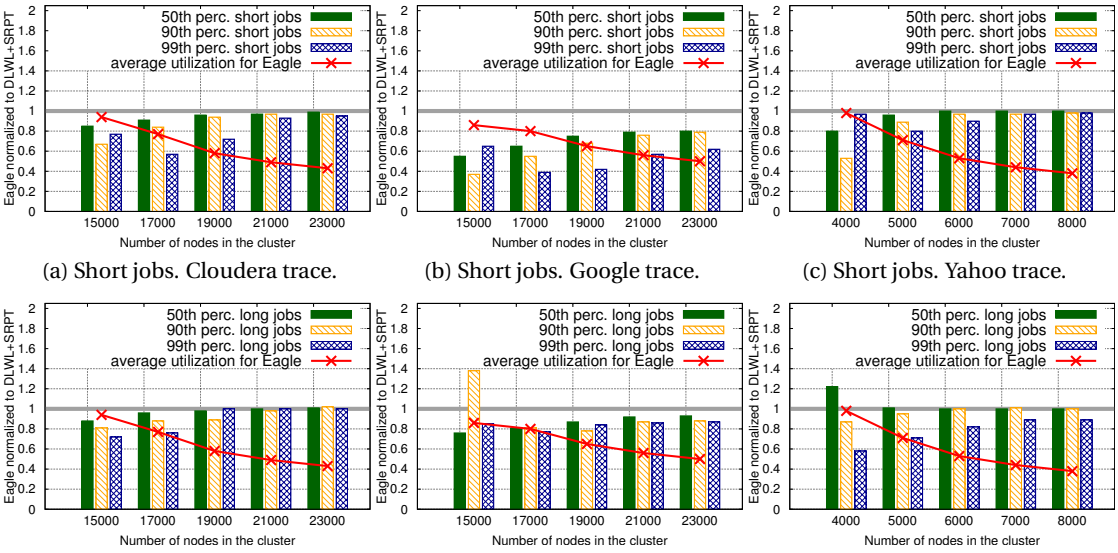
From Figure 3.2 we see that for short jobs Eagle achieves better job completion times than Hawk at all percentiles, for all load conditions and in all traces. The improvements brought about by Eagle are more evident at higher loads, since the impact of the better resource utilization achieved by SSS and SBP is higher when resources are scarce. At the highest load, Eagle achieves a speedup that ranges between a minimum factor of 3 and a maximum factor of 10 (Figure 3.3c). As the load goes down, the benefits of Eagle tend to decrease, as the abundance of available computational resources makes scheduling decisions less important.

Figure 3.2 shows that long jobs in Eagle are not negatively affected by the execution of short jobs in the general partition. On the contrary, in some cases, the completion times of long jobs are better in Eagle than in Hawk. This result showcases Eagle's ability to opportunistically allow short jobs to take advantage of computational resources in the general partition without impairing the completion times of long jobs.

We now provide some detailed data to compare the effectiveness and the efficiency of the strategies that Eagle and Hawk implement to combat head-of-line blocking, i.e., SSS and work stealing, respectively. To this end, we show the frequency of head-of-line blocking in both systems, and the message overhead they occur to avoid it. We focus on this aspect of the two systems because Hawk does not provide support for job-awareness.

Table 3.2 reports the following metrics: i) number of probes that are initially scheduled behind a long task; ii) number of short tasks that execute after experiencing head-of-line blocking, i.e., that are not "rescued" by stealing; iii) number of re-scheduled probes, namely stolen probes for Hawk and probes re-assigned by Eagle during its re-scheduling phase. The data reported in Table 3.2 are collected under the highest load condition for all three traces. We are going to analyze the Table by looking at its columns from left to right.

The first result that the Table reveals is that, thanks to SSS, Eagle never schedules a probe behind a long task and, as a consequence, totally avoids head-of-line blocking. In contrast, Hawk initially schedules, depending on the trace, from 0.87 to 17.3 million probes behind long jobs, which correspond to 84%, respectively, 67% of the total number of probes sent. Among these probes, 18 thousand to 0.63 million, again depending on the trace, experience head-of-line blocking before they get to the head of their queue and are able to execute a task. For the Google trace, these tasks constitute roughly 5% of the total number of short tasks.



(a) Short jobs. Cloudera trace. (b) Short jobs. Google trace. (c) Short jobs. Yahoo trace.

(d) Long jobs. Cloudera trace. (e) Long jobs. Google trace. (f) Long jobs. Yahoo trace.

Figure 3.3 – 50/90/99th percentiles of job completion times. Eagle normalized to DLWL+SRPT.

The following column reports the number of probes that are re-scheduled in the two systems, either by work stealing (in Hawk) or by retrying the probe placement (in Eagle). We see that a large fraction of the probes initially scheduled behind a long task is successfully stolen in Hawk (from 90 to 98% depending on the trace), thus mitigating the impact of head-of-line blocking. The number of probes that are re-scheduled with SSS is only marginally higher than the number of probes stolen in Hawk (up to 8% more).

Despite this marginal increase in number of re-scheduled probes, Figure 3.2 demonstrates that Eagle's scheduling design is more effective than Hawk's. The work stealing in Hawk is a reactive scheme, that is triggered whenever a node becomes idle. This scheme has two main drawbacks. On the one hand, the likelihood of a node being idle is inversely proportional to the load, thus reducing the number of times that work stealing is triggered under high load. On the other hand, whenever a task is stolen, it has probably already experienced some head-of-line blocking, negatively impacting the corresponding job's completion time. In contrast, the proactive nature of Eagle's probe re-scheduling avoids scheduling short tasks behind long ones altogether. This is crucial for task-parallel jobs, especially looking at high percentiles in their completion time distribution, because even failing to steal one of the probes of a job might have a huge impact on the job's completion time.

3.6.2 Comparing Eagle Against DLWL+SRPT

We compare Eagle to the distributed version of the recent Yaq scheduler [81]. We refer to this design as DLWL+SRPT. Queue reordering in DLWL+SRPTis implemented on top of a distributed version of an LWL-like scheduler, in which the estimated queue waiting times

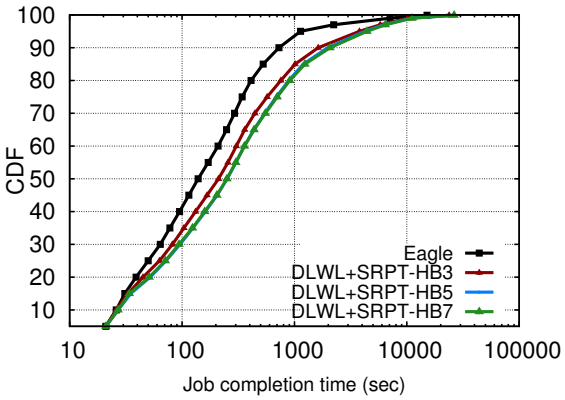


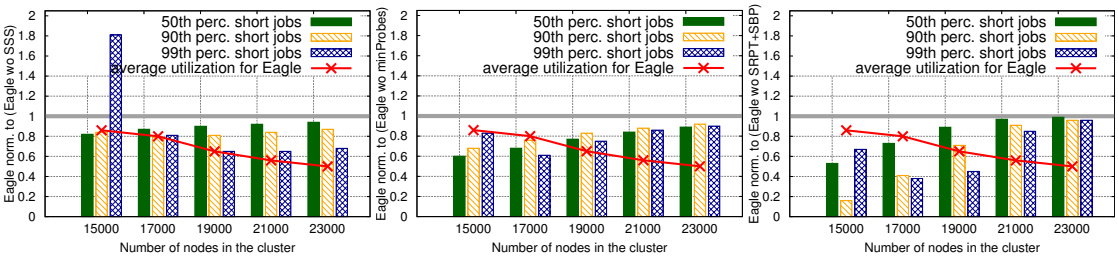
Figure 3.4 – DLWL+SRPTwith 3s, 5s and 7s heartbeat interval compared to Eagle. Short jobs, Google trace, 15000 nodes.

are made available to the distributed schedulers through periodic updates (hereby referred to as heartbeats). For the Google trace, with an average job inter-arrival time of 1s, we use a heartbeat interval of 3s, a value commonly used in the industry. For the Yahoo and Cloudera traces the heartbeat intervals are 7s and 12s. To reduce the chance of conflicts by several distributed schedulers picking the same workers with a low advertised load, we add a small random number (smaller than the heartbeat interval) to each queue waiting time, as done in Apollo [16]. We use the same relative threshold as in Eagle to prevent starvation, and we also reserve the same small partition of the datacenter for short jobs. Doing so improves DLWL+SRPT's performance, although Yaq's design does not include it. Finally, we use SRPT as a common queue reordering policy for both systems.

Figure 3.3 (top) shows, as a function of the number of worker nodes, the 50th, 90th, and 99th percentiles of the job completion time distribution for short jobs for Eagle, normalized to those same values in DLWL+SRPT. In addition, we report the average node utilization with the given number of nodes. Figure 3.3 (bottom) reports the same results for long jobs.

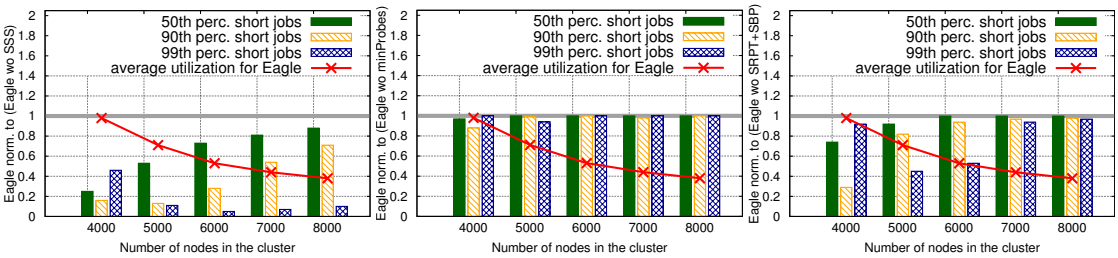
For short jobs, Eagle performs better than DLWL+SRPT, because SBP improves SRPT's ability to quickly remove jobs from the system. Moreover, Figure 3.3 (bottom) shows that Eagle also improves the completion times of long jobs in the vast majority of the cases. The reason is that Eagle does not enqueue short tasks after long ones, so a long task can start executing after all short probes in front of it are serviced. In contrast, in DLWL+SRPT, short tasks can queue after long ones and bypass them due to SRPT, resulting in an additional delay for the long jobs.

DLWL+SRPTis sensitive to the value of heartbeat interval. For the Google trace, Figure 3.4 shows on a logarithmic scale how DLWL+SRPTis affected by an increase in the heartbeat interval. Increasing the heartbeat interval from 3s to 5s results in a 4.8% increase for the 90th percentile of the short job completion time, and a 2.2% increase for the 99th percentile. When increasing the interval from 3s to 7s, the increases in the 90th and 99th percentiles are 6.8% and 4.2%.



(a) Eagle vs Eagle without SSS. (b) Eagle vs Eagle without min probes. (c) Eagle vs Eagle without SBP+SRPT.

Figure 3.5 – Breakdown of Eagle's benefits. Eagle normalized to Eagle without one of its components. Short jobs. Google trace.



(a) Eagle vs Eagle without SSS. (b) Eagle vs Eagle without min probes. (c) Eagle vs Eagle without SBP+SRPT.

Figure 3.6 – Breakdown of Eagle's benefits. Eagle normalized to Eagle without one of its components. Short jobs. Yahoo trace.

3.6.3 Breakdown of Eagle's Benefits

In this Section we evaluate the benefits of each of Eagle's components separately. To do so, we compare Eagle to three variants of Eagle, each being stripped of one of Eagle's features: SSS, SBP+SRPT and the minimum number of probes per job.

Figures 3.5 and 3.6 report, for the Google and Yahoo traces, respectively, the 50th, 90th and 99th percentiles of short job completion times for the full-fledged Eagle implementation, normalized to the same values for Eagle's variants. The lower the value reported in the graph for a variant without a given feature, the more that feature contributes to Eagle's overall performance. Figures 3.5 and 3.6 also report the datacenter utilization corresponding to each simulated deployment. We do not report results for long jobs, because Eagle's features are largely aimed at improving short job response times.

Divide

Figures 3.5a and 3.6a showcase the performance of Eagle against Eagle without its SSS component. We see that SSS is a key ingredient to Eagle's success. SSS shows greater benefit for the Yahoo trace, because in that trace there are more tasks in a job, and so without SSS, there is a higher chance that at least one task of a job is affected by head-of-line blocking. Such blocking

is more likely to occur at high load, so for the Yahoo trace that is precisely where SSS shows the greatest benefit. The benefit of SSS is more pronounced for the higher percentiles, as that is where the impact of stragglers is best visible.

For the Google trace, the addition of SSS leads to improvements in all cases, except for the 99th percentile for the highest load. In this extreme case the short partition becomes highly loaded. It thus becomes better to enqueue a probe after a long task in the general partition than after many short tasks in the short partition. We verify this hypothesis by increasing the size of the short partition from 17% to 20% of the workers. This lessens the load in the short partition, and as a result the 99th percentile becomes better for Eagle with SSS.

Stick to Your Probes

Figure 3.5c shows that SBP+SRPT significantly helps Eagle for the Google trace. Comparing to Figures 3.5a and 3.5b we see that SBP+SRPT is the feature that contributes the most to Eagle's performance, because one-task short jobs are abundant in the Google trace and SRPT allows them to bypass other tasks in the queue. SBP+SRPT is also effective for the Yahoo trace, as depicted in Figure 3.6c. As expected, the benefits of SRPT+SBP diminish at lower loads, when enough workers are idle.

Minimum Number of Probes per Job

Figure 3.5b and Figure 3.6b show the effectiveness of sending a minimum number of probes to help jobs with very few tasks. As expected, this feature has a much higher impact in the Google trace than in the Yahoo trace, because of the much higher number of jobs with few tasks in the Google trace. For the Google trace, at high load (15000 and 17000 nodes), setting a minimum number of probes for short jobs achieves a speedup up to 30-40% at all the considered percentiles. Conversely, improvements for the Yahoo trace are more modest, reaching 10% in the higher percentiles. These performance gains come at the cost of a negligible amount of extra messaging overhead. Sending 20 probes, in fact, corresponds to contacting only 0.05% of the nodes in the system in the worst considered case (4000 nodes).

3.6.4 Comparison Against an Omniscient Scheduler

In order to assess the relative benefits of knowing only the location of long jobs, as provided by SSS, compared to having complete information about the estimated work left in each worker node queue, we compare SSS, combined with a minimum number of probes per job, to an omniscient scheduler that uses this complete information to schedule all tasks (short and long) in LWL fashion. The latter provides an upper bound on the performance that can be achieved by using this complete information.

To obtain this upper bound, the omniscient scheduler is enhanced with a (static) datacenter

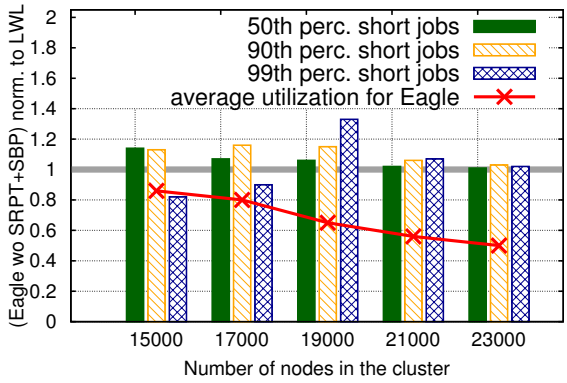


Figure 3.7 – Eagle without SBP+SRPT against an omniscient scheduler. Google trace, 15000 nodes.

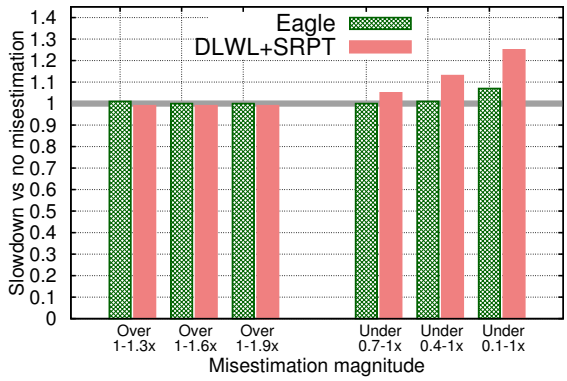


Figure 3.8 – Sensitivity of Eagle and DLWL+SRPT to under and over-estimation of task execution time. Google trace, 15000 nodes, 99th percentile short job completion time.

partitioning scheme with the same configuration as Eagle. Without such a partitioning, LWL is not able to match Eagle, because, especially at high load, several long jobs can occupy most of the datacenter's resources.

Figure 3.7 shows that Eagle delivers performance very close to that achieved by an omniscient LWL scheduler. In other words, with only the information about the location of long jobs, Eagle performs almost as well as having the up-to-date estimated queue lengths of all workers. In some cases Eagle is better than LWL, because of late binding as a result of SBP, and because the information on which Eagle relies, namely the location of tasks of long jobs, is smaller and easier to keep up-to-date.

3.6.5 Sensitivity to Misestimation

We analyze the impact of task length misestimation on the performance delivered by the job-aware scheduling policies of Eagle and DLWL+SRPT. To do so, we multiply the *estimated* task execution time of every job by a random value, chosen uniformly at random within the

range $[0.x, 1]$ for under-estimation and $[1, 1.x]$ for over-estimation. The actual task execution times remain the same, only the estimate used by the scheduler changes. In other words, over-estimation (under-estimation) means that tasks complete faster (slower) than the scheduler expects. We set x to 3, 6 and 9.

Figure 3.8 shows the slowdown for Eagle and DLWL+SRPT, when misestimations occur. The results are for the 99th percentile of the short job completion time. For both systems, the impact of the misestimations on the 50th and the 90th percentiles is minimal.

Figure 3.8 shows that both systems are similarly robust with respect to over-estimation. However, Eagle is more robust to under-estimation than DLWL+SRPT. SBP is the reason behind the robustness. The early binding performed by DLWL+SRPTleads to stragglers, while the late binding in SBP avoids stragglers. This effect is exacerbated by under-estimation.

3.6.6 Simulation summary

We conclude this Section by showing that, although they may seem orthogonal techniques, SSS, SBP and minProbes are instead complementary, and represent synergistic building blocks of a unified, principled design.

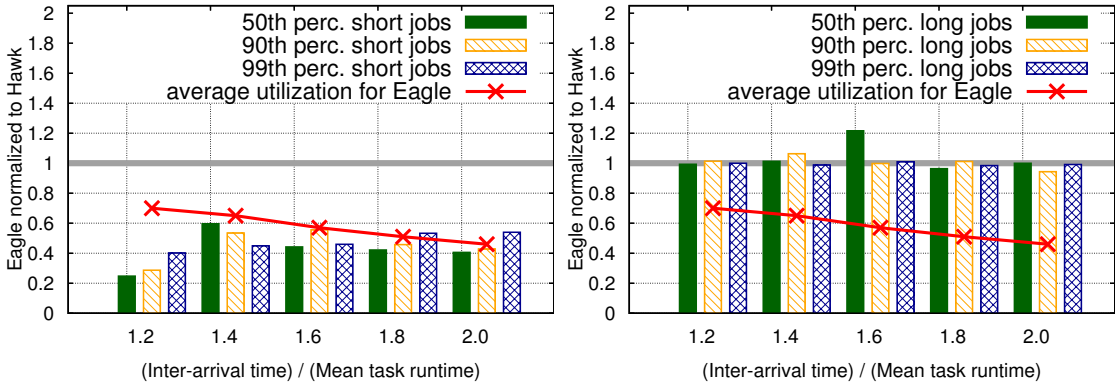
Applying standard SRPT (or any policy based on preemption) in Eagle's target system model is not possible, because tasks cannot be preempted. Therefore, SBP uses the time when a task finishes as natural evaluation point to determine the next job to be executed and to pick a task from that job. If we were to allow the entire datacenter to become overwhelmed with long tasks, then such evaluation points would occur at low frequency, leaving only few opportunities to favor short jobs. Instead, with SSS, even in the case where the general partition is full, short tasks complete at a high rate in the short partition. The result is an implementation of SRPT without preemption, but at a granularity that is fine enough to benefit from it.

The minProbes enhancement further amplifies the gains brought about by the other two techniques. It increases the chances for a probe of a job with few tasks to land on an unloaded node. SSS plays a complementary role in the effectiveness of minProbes, since, in a datacenter crowded with long tasks, any number of probes would still likely suffer head-of-line blocking, if no nodes were reserved for short tasks.

3.7 Implementation Results

We implement Eagle in the Spark framework [107]. We run an Eagle daemon that runs in each worker node to manage their queue, and a scheduler client as a plug-in for Spark.

We deploy this implementation of Eagle on a 100-node cluster, using a centralized scheduler and ten distributed schedulers. To evaluate its performance, we compare it against an implementation of Hawk, an earlier hybrid scheduler [29].



(a) Short jobs

(b) Long jobs

Figure 3.9 – 50/90/99th percentiles of the job completion times in Eagle, normalized to Hawk for a 3300-job sample of the Google trace.

To keep the time to run experiments tractable, we use a scaled-down version of the Google trace. We use a 3,300-job sample of the trace, and we reduce the duration of each task in the sample by a factor of 1,000 (from seconds to milliseconds). We also reduce the number of tasks in a given job by the ratio between the number of nodes in the original trace and in the sampled-down trace. In order to keep the task-seconds ratio between long and short jobs the same as in the original trace, we increase the task duration in the affected jobs by the same ratio. Job arrival follows a Poisson process, and we vary the cluster load by varying the mean job inter-arrival time as a multiple of the mean task runtime.

Figure 3.9 presents the 50th, 90th and 99th percentiles of execution times when running with Eagle, normalized to the same values for Hawk, for various ratios of mean inter-arrival time to average job completion time. As this ratio get bigger, the cluster load decreases. In Figure 3.9a we see that Eagle is better across the board for short jobs. The higher the load, the more improvement we get. For example, the improvement reaches 80% for a ratio of 1.2. As the load decreases, the gains of Eagle over Hawk stabilize, but remain non-negligible, up to 60% for the 50th percentile.Figure 3.9b presents the same results for long jobs and shows that the performance delivered by Eagle and Hawk are comparable. This result showcases the ability of Eagle to improve short job completion times without hurting the performance of long jobs.

3.8 Summary

In this chapter we present two new techniques to improve scheduling of data-parallel jobs in datacenters. First, we dynamically divide the nodes into different partitions for executing long and short jobs, thereby avoiding head-of-line blocking, the queueing of a short job behind a long one. This division is implemented by a technique called succinct state sharing (SSS), by which distributed schedulers are informed about where long jobs are queued or executing. Second, sticky batch probing (SBP) allows a probe for a short job to request further tasks of

that job, thereby avoiding straggler tasks. In combination with SRPT, it favors the expedient completion of short jobs.

We have incorporated these techniques in the Eagle hybrid scheduler, in which the centralized component schedules the long jobs, and the distributed schedulers take care of the short jobs. With such a hybrid scheduler, it becomes particular easy to implement SSS, by simply piggybacking the location of long jobs on scheduling messages. We have implemented Eagle as a Spark plug-in, and measured the performance of this implementation. We have also extensively evaluated Eagle by means of simulation. Our results indicate that Eagle avoids head-of-line blocking altogether, and outperforms various state-of-the-art schedulers.

4 Kairos: Preemptive Scheduling Without Runtime Estimates

"The most sublime act is to set another before you.."
— William Blake

4.1 Introduction

The vast majority of datacenter 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.

In this chapter, we introduce an alternative approach to datacenter scheduling, which does not use task runtime estimates. Our approach draws from the Least Attained Service (LAS) scheduling policy [75]. 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 datacenter workloads that are common in datacenters.

The main challenge is to find a good approximation for LAS in a datacenter 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 [19]. 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 [27]. 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. We make the following contributions:
1) We demonstrate good datacenter 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 chapter is as follows. Section 4.2 provides the necessary background. Section 4.3 describes the design of Kairos. Section 4.4 describes its implementation in YARN. Section 4.5 evaluates the performance of the Kairos YARN implementation. Section 4.6 provides simulation results. Section 4.7 summarizes the chapter.

4.2 Background

4.2.1 Estimating task runtimes

Estimates in existing systems. Most state-of-the-art datacenter schedulers rely on task runtime estimates to make informed scheduling decisions [16, 27, 29, 31, 42, 43, 44, 45, 60, 109]. 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

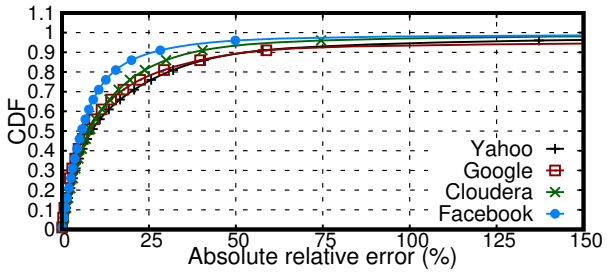


Figure 4.1 – Prediction error for estimating the duration of each task in a job as the mean task duration in that job.

from past executions of the same task, if any, from past executions of similar tasks [16], or by means of on-line profiling [31]. A common estimation technique for the task duration is to take the average of the task durations over previous executions of the job [29, 81]. More sophisticated techniques rely on machine learning [80].

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 [81]. Even if jobs are recurring, changes in the input data set may lead to significant and hard-to-predict shifts in task runtimes [2]. 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 [22, 66]. Finally, failures and transient resource utilization spikes may lead to stragglers [4], which not 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 datacenter schedulers. In particular, we consider the Cloudera [20], Facebook [20], Google [82] and Yahoo [21] 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 4.1. 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 [80].

Coping with misestimations. Previous work has shown that task runtime misestimation leads to worse job completion times [27], and failure to meet service level objectives [31, 99] or job completion deadlines [99]. Some systems deal with misestimations by runtime correction

mechanisms such as task cloning [4] and queue re-balancing [81], or by using a distribution of estimates rather than single-value estimates [80]. 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 [75] to a datacenter 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 datacenter workloads.

4.2.2 Least Attained Service

Prioritizing short jobs. Datacenter workloads typically consist of a mix of long and short jobs [21, 20, 82]. Giving higher priority to short jobs improves their response times by reducing head-of-line-blocking. The Shortest Remaining Processing Time (SRPT) scheduling policy [86] 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 [85].

Recent systems have successfully adopted SRPT in the context of datacenter scheduling [27, 54, 81]. 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 [75]. 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.

4.3 Kairos

4.3.1 Design overview

Challenges of LAS in a datacenter. LAS is an appealing starting point to design a datacenter 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 datacenter 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 4.2. The node schedulers implement LAS locally on each worker node (§4.3.2) 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 (§4.3.3).

4.3.2 Node scheduler

Each worker node has N cores, which can run N concurrent tasks, and a queue, in which preempted tasks are placed. Algorithm 2 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.

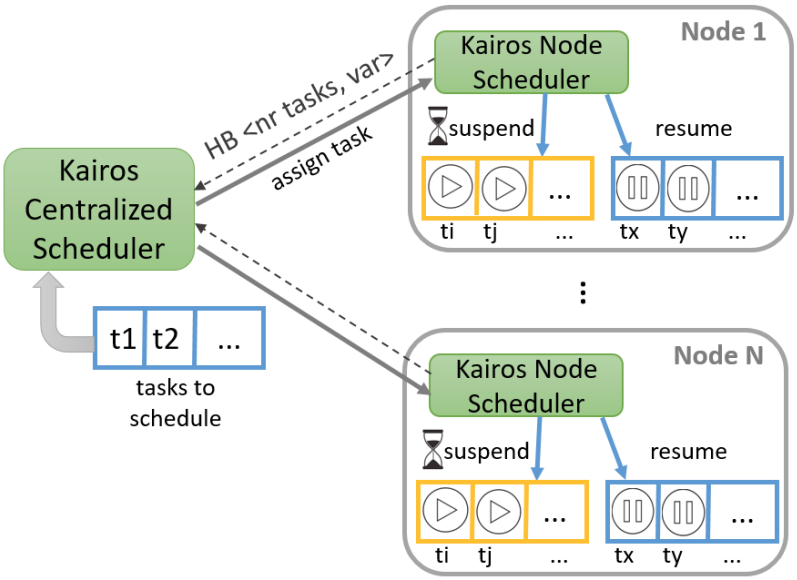


Figure 4.2 – Kairos' two-level scheduling architecture. Node schedulers implement LAS locally. The central scheduler assigns tasks to nodes.

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 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 §4.3.3.

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.

Algorithm 2 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
```

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

Algorithm 3 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 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 §4.5.3, where we show that Kairos is relatively robust to sub-optimal settings of W.

4.3.3 Central scheduler

Algorithm 3 presents the data structures maintained by the central scheduler and its operations.

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 [16, 81]. 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.

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 (§4.3.3).

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 (§4.3.3).

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 §4.5.3, where we show that Kairos' performance is not dramatically affected by sub-optimal settings of Q.

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 [13, 34]. 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.

4.4 Kairos implementation

We implement Kairos in YARN [41], a widely used scheduler for data-parallel jobs. Figure 4.3 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` 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

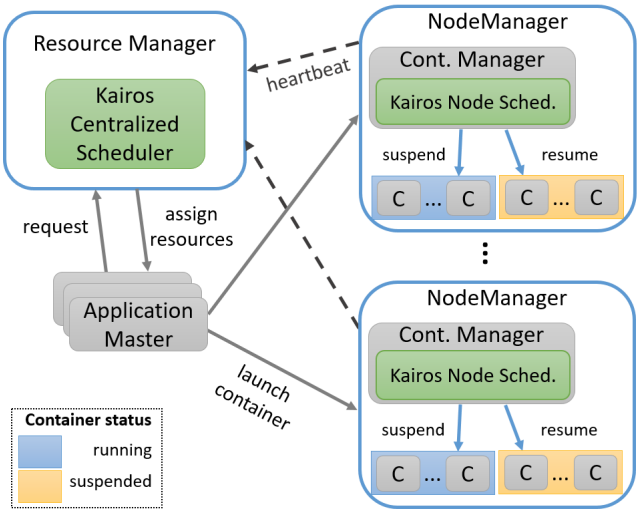


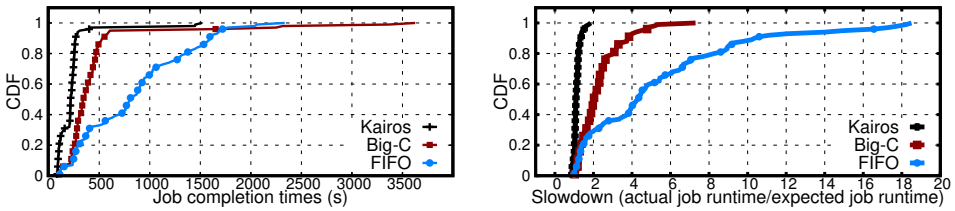
Figure 4.3 – Kairos integration in YARN.

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 4.1 – Job categories composing our workload. Job runtimes follow a heavy-tailed distribution, typical for modern datacenter workloads.

`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. [19]. 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.

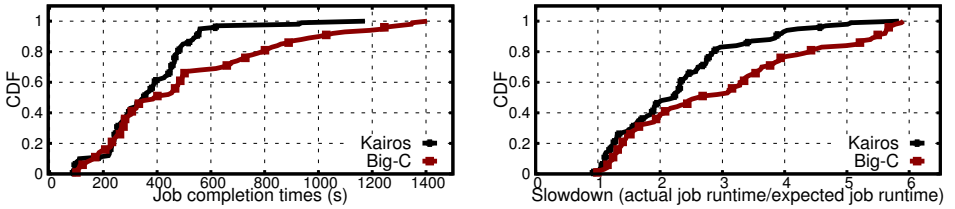
We have made the Kairos code publicly available. [1]



(a) CDF of job completion times.

(b) Slowdown.

Figure 4.4 – 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 4.5 – CDFs of job completion times and slowdown in Kairos, Big-C. Uniform workload.

4.5 Experimental Evaluation

4.5.1 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 [19].

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%).

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

4.5.2 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 4.1. 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 4.1 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 4.1. 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.

4.5.3 Results

Job completion times

Figure 4.4a reports the CDF of job completion times with Kairos, YARN/FIFO and Big-C. Figure 4.4b 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 long jobs can be delayed due to frequent preemptions and their low priority in Big-C.

As illustrated in Figure 4.4b, 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 4.5. For this workload we only use the first 3 categories in Table 4.1 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.

LAS-aware task dispatching

Figure 4.6 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 §4.3.3 and denoted by `Var` in Figure 4.6, 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 4.6 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 4.1), are effectively prioritized. The benefit at higher percentiles shows that `Var` is also able to effectively use LAS to improve the response time of

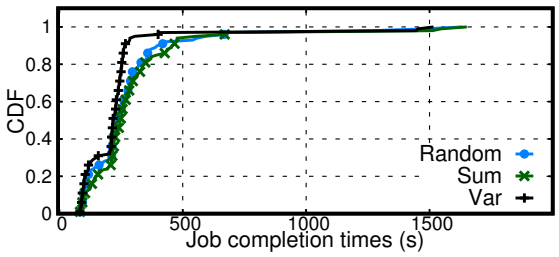
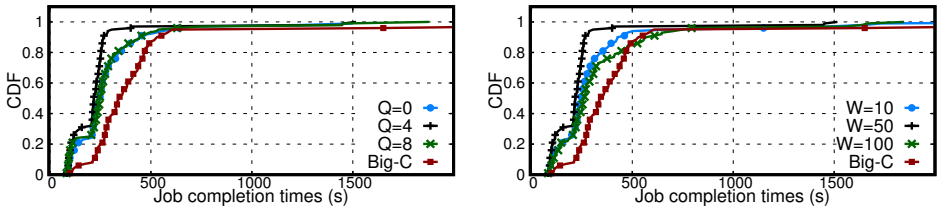


Figure 4.6 – Comparison of alternative task-to-node dispatching policies in Kairos. Tail omitted for visibility. Worst tail job completion times are 1633s for Random, 1651s for Sum and 1452s for Var.



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

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

Figure 4.7 – Sensitivity analysis to parameters Q (queue size) and W (time quantum).

larger jobs as well.

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 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.

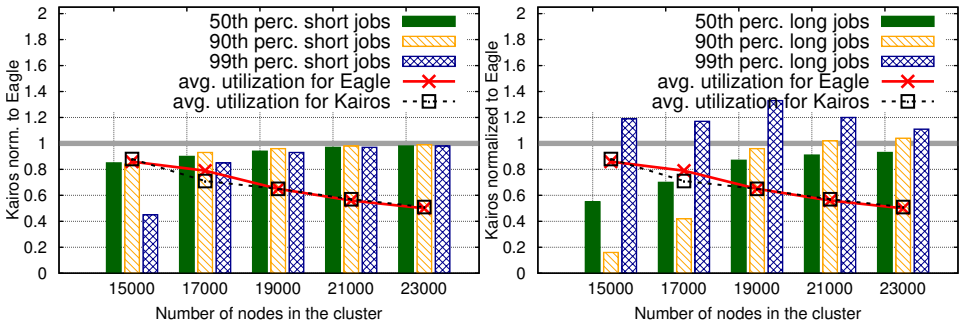
Sensitivity to Q. Figure 4.7a shows the CDF of job response times in Kairos with $Q = 2, 4, 8$ and in Big-C. $Q = 8$ and $Q = 2$ perform slightly worse than the default value of $Q = 4$ we use in Kairos, but still deliver better performance than Big-C at each percentile.

The shape of the CDFs for different values of Q matches our analysis of §4.3.3. 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 4.7b shows the CDF of job response times in Kairos with $W =$

Trace	Total # jobs	% Long jobs	% Task-Seconds long jobs
Yahoo [21]	24262	9.41%	98%
Google [82]	506460	10.00%	83%

Table 4.2 – 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.



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

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

Figure 4.8 – Kairos normalized to Eagle short (a) and long (b) jobs. Google trace.

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.

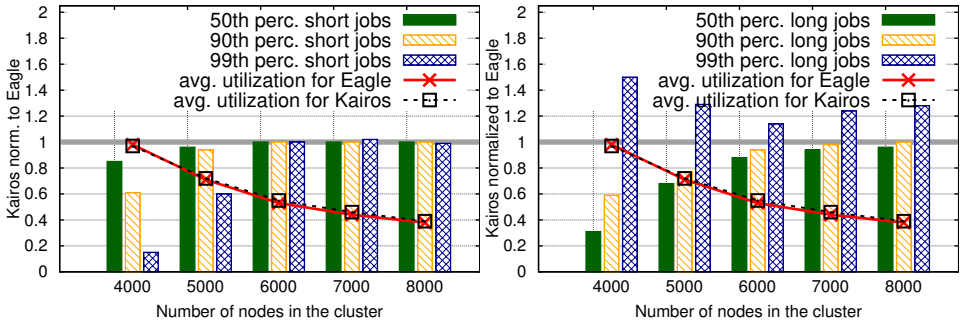
4.6 Simulation

4.6.1 Methodology and baseline

We evaluate Kairos in larger datacenters by means of a simulation study using the popular Yahoo [21] and Google [82] traces. We compare Kairos to Eagle [27], 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

[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) Kairos short jobs normal. to Eagle. Yahoo trace.

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

Figure 4.9 – Kairos normalized to Eagle short (a) and long (b) jobs. Yahoo trace.

long jobs and one for short jobs. The nodes of the datacenter 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.

4.6.2 Simulated testbed

Platform. We simulate datacenters 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 4.2 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 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.

4.6.3 Results

Figures 4.8 and 4.9 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 4.8(a) and 4.9(a) 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 4.8(b) and Figure 4.9(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.

4.7 Summary

We present Kairos, a new datacenter 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.

5 Related Work

"Perfection must be reached by degrees; she requires the slow hand of time."
— Voltaire

Schedulers in datacenters have been an active topic of study in the community and have evolved rapidly over the past ten years. Table 5.1 gives an overview, ordered by year of publication, of the main schedulers that are deeply studied in this chapter (other schedulers can also be found in the development of pertinent sections). The table also reflects on the importance of this research field: papers were published in top-tier systems conferences, the frequency of publications shows the rapid evolution and need for scheduling solutions, and the last column shows that many of these techniques were regarded as best papers and/or were adopted for use in production.

Datacenter schedulers can be compared and analyzed according to different criteria, however this chapter discusses mainly and foremost the four challenges covered by this dissertation (Chapter 1) and how the state-of-the-art helped addressing them.

Challenge 1. Section 5.1 showcases the *conflicting goals* of high-quality scheduling placement and low scheduling latency that centralized and distributed *scheduler designs* have to choose from, and how hybrid schedulers can help to reconcile both goals.

Challenge 2. Another aspect that this dissertation focuses on is the *head-of-line blocking* problem, explored in Section 5.2. While there exist other problems (Section 5.6) that datacenter schedulers tackle, head-of-line blocking translates directly to increased job runtime for short jobs by one or two orders of magnitude (e.g. waiting behind a long job).

Challenge 3. Section 5.3, develops on the *lack of job-awareness* in current solutions and how that incurs into inferior job runtimes.

Challenge 4. Furthermore, an analysis of the *dependency on assumptions* that state-of-the-art schedulers make about is presented in Section 5.4

Scheduler	Conference	Year	Merits
Quincy [58]	SOSP	2009	
Delay [106]	Eurosys	2010	
Mesos [51]	NSDI	2011	Adopted in production
Omega [88]	Eurosys	2013	Best student paper
Yarn [100]	SoCC	2013	Best paper and adopted in production
Sparrow [79]	SOSP	2013	
Apollo [16]	OSDI	2014	Adopted in production
Borg [102]	Eurosys	2015	Adopted in production
Mercury [60]	Usenix ATC	2015	
Hawk [29]	Usenix ATC	2015	
Yaq [81]	Eurosys	2016	
Tetrisched [99]	Eurosys	2016	Best student paper
Eagle [27]	SoCC	2016	
Firmament [42]	OSDI	2016	
Big-C [19]	ATC	2017	
Medea [38]	Eurosys	2018	Adopted in production
Kairos [28]	Technical report	2018	

Table 5.1 – List of main datacenter schedulers studied thoroughly in this chapter. Additional scheduling papers are overviewed in the different sections of this chapter.

Table 5.2 describes in a nutshell scheduler designs, scheduling policies, and node queue management techniques that address challenges 1 and 2. Challenges 3 and 4 are not comprised in the table because current datacenter schedulers do not focus primarily on these problems.

Additionally, we make a study of different scheduling correction mechanisms (Section 5.5). Section 5.6 explores other properties provided by schedulers. The way the schedulers are benchmarked is described in Section 5.7. This chapter concludes with a brief overview on scheduling in other contexts such as HPC and Operating systems (Section5.8).

5.1 Enhancing scheduler design

Scheduler design has changed over the years in order to deal with the different challenges posed by datacenter evolution. Initially, design followed a centralized architecture [58, 106, 102]. A second generation of schedulers, meta-schedulers [51, 88] were designed to increase the flexibility of scheduling policies, or decouple scheduling from other functionalities [100]. Later, to enhance the scheduling latency to a maximum, the design of schedulers evolved to fully distributed [79, 16]. However, as described in Chapter 1, these schedulers suffer from not making optimal scheduling decisions because they trade having an accurate view of the resources usage in the cluster. To get the best of both worlds, hybrid scheduling was proposed as a design combination of one centralized scheduler and many distributed schedulers [60, 29, 27].

Table 5.2 depicts different scheduling designs adopted by the community, along with the scheduling policy implemented in their centralized and distributed components. Furthermore, distributed schedulers need to deal with conflicts (concurrently scheduling in the same resource) by either avoiding them with a coordination mechanism or solving them with a conflict-solving mechanism, this is also depicted in the table.

5.1.1 Centralized schedulers

A datacenter scheduler is centralized when there is only one single entity in charge of making all of the scheduling decisions. Among the early centralized schedulers we find Quincy [58], Delay Scheduling [106] for Hadoop [10] and Dominant Resource Fairness [41]. The main focus of these papers is to conciliate locality and fairness. The trade-off is: if simple fair scheduling is applied jobs will tend to be assigned the same "sticky slot" [105] which is not good for locality because data files are spread across all nodes. Quincy maps scheduling needs (locality, fairness, starvation-freedom) to a cost function in a graph data structure and then solves it as an optimization problem. In delay scheduling if a job, scheduled in a fair way, cannot launch locally a task, it is delayed for a small amount of time. Dominant Resource Fairness [41] provides an algorithm that generalizes max-min fairness to multiple resources. This logic is ultimately part of the CapacityScheduler in Yarn [100]. In this paper, the focus is also to increase resource utilization. Quincy was made for Dryad [57] DAG-based programming model; Delay scheduling and Dominant resource fairness schedule Hadoop jobs.

As high utilization became more important in datacenters, designs focused on high utilization were studied. For instance, the resource manager at Google, Borg [102], translates resource requirements specified by users to scores. Scheduling is done then in two phases: 1. a scoring fit that aims to provide better packing [101] and, if this fails, 2. tasks with lower priorities can be killed. Single datacenter server schedulers like Quasar [30] and Paragon [31] also aim to increase resource utilization in a datacenter by collocating workloads without the danger of decreased performance due to contention.

The idea of scheduling long jobs differently from short jobs to provide better scheduling latency was first introduced in Hawk [29]. Some recent centralized schedulers adopt this notion. Big-C [19] uses different capacities (Yarn CapacityScheduler) as a way to give priority to short jobs over long jobs, using preemption when necessary. Medea [38] schedules long running applications using MILP and short ones using a task-based approach.

Combinatorial optimization Finally, a class of schedulers formalize the scheduling decision as a combinatorial optimization problem: Quincy [58], Rayon [25], Tetris [43], Tetrisched [99], Firmament [42], Medea [38] and 3Sigma [80]. The resulting Mixed-Integer Linear Program (MILP) is solved either exactly, or an approximation is computed by means of heuristics. In Quincy, the scheduling problem is formulated as an minimum-cost maximum-flow problem by using locality and fairness as the main conflicting goals for jobs. Firmament borrows the

idea of stating scheduling preferences of different jobs as a minimum-cost maximum-flow model from Quincy. Unlike Quincy, Firmament is scalable thanks to the proposed relaxation algorithm combined with a cost-scaling technique. Rayon and Tetrisched are schedulers that allow reservations: jobs scheduled to execute in the future. While Rayon is a reservation system designed to guarantee resource availability in the long term, Tetrisched targets short-term job placement. In Tetrisched introduces a new Space-Time Request Language (STRL) that allows jobs to express their preferences in space-time manner, expressions in STRL are compiled and converted to an MILP formulation. Medea distinguishes the scheduling for long running applications from latency sensitive jobs. Medea formulates the placement of long running applications with constraints as an integer linear program and solve it as an online optimization problem. 3Sigma proposes to schedule not only based on a single-point estimate but to use the entire distribution instead, it also uses MILP to represent and solve each job's resource request. Computing the solution to such problems yields very good scheduling decisions, but is computationally expensive and needs to be performed in a centralized fashion.

Given the growth in scale of datacenters, centralized scheduling became quickly the source of bottleneck concerns [88]. The following sections 5.1.2, 5.1.3, and 5.1.4 explore on schedulers proposed to solve the scalability problem.

5.1.2 Meta-schedulers

In this dissertation we call meta-schedulers the scheduling frameworks that propose a common way for many independent schedulers to coordinate and share resources in a cluster. In this category we include Mesos [51], Omega [88] and Yarn [100].

While centralized scheduling approaches can enforce global scheduling policies and achieve high-quality placements, they suffer from scalability constraints. Moreover, centralized schedulers are tightly coupled with other resource manager functionalities, such as task scheduling, speculative execution or failure handling, which further impacts scalability.

Mesos [51] proposes a two-level scheduler design: at the top level there are schedulers of –potentially different– frameworks that share the cluster (e.g. Spark [106], Hadoop, MPI [15]); and at the bottom level there is a scheduling layer that acts as a multiplexer between the cluster and the top level schedulers. Mesos, the intermediate layer, contains an allocator module that periodically offers free resources to the top level schedulers which can in turn accept or reject this offer. The scheduling policy itself is then two-fold: on one side, schedulers at the top level can implement any scheduling policy (for example use Quincy or Delay scheduling); on the other side the Mesos allocator module makes resource offers to frameworks that run on top of it based on an organizational policy such as fair sharing [51]. The main benefit of Mesos is the ability to share the cluster resources among different frameworks. However, Mesos suffers from an important flaw: top level schedulers do not have full view on the cluster resources utilization because their scheduling decisions are based on the offers they get, their

scheduling will therefore not be optimal. On the other hand, the allocator in Mesos is not aware of the policies that the schedulers implement and consequently would not know how to make better offers.

Similar to Mesos, in Omega [88] a set of schedulers share the cluster resources. Unlike Mesos, these do not depend on an intermediate layer and can make independent scheduling decisions. Given the lack of coordination, two or more scheduling decisions could collide. In that case, one scheduler gets the resources while the other(s) re-schedule. Schedulers in Omega need to agree on a common relative importance on the jobs that they run so that higher priority jobs can kill lower priority jobs (like in Borg). The advantage of Omega over Mesos is that schedulers have a complete view of resource usage in the cluster and can therefore make better informed decisions. Additionally, the scheduling latency is enhanced with respect to Mesos because schedulers can take immediate and independent decisions. The disadvantage is that, depending on the conditions (e.g. percentage of resources occupied, number of schedulers, number of resources desired), schedulers can have a large amount of conflicts. Scheduling in this situation requires a number of interactions that will ultimately hurt both the quality and the latency of scheduling. Another drawback is that it is difficult to implement a cluster-wide policy in such a setting because each scheduler can have a completely different logic.

As the Hadoop original architecture became used for more general computing and limited in scalability for datacenters, Yarn [100] was proposed as the new enhanced framework to run Hadoop-like jobs. The main contribution of Yarn for scheduling is the separation of concerns: the scheduler component takes care of scheduling while other functionalities like tracking and managing the tasks execution is outsourced to other components, like the application manager. The second characteristic of Yarn is that it is a meta-scheduler and other systems can run on top of it (Dryad [57], Giraph, Spark [106], Tez among others). The scheduling latency is enhanced because the scheduler component dedicates solely on scheduling. However, all the scheduling decisions in Yarn still need to pass through a centralized scheduler component. Depending on the number of scheduling requests this component can handle, it might not be enough to target very small jobs.

Meta-schedulers help lowering the scheduling latency by having many schedulers taking decisions at once, but the latency is still hurt because of the need of coordination (in Mesos the allocation module, in Omega the conflicts and in Yarn the centralized scheduler). Unlike centralized schedulers, meta-schedulers make it difficult to get high-quality scheduling decisions because schedulers have limited information on either how the cluster is being utilized (Mesos) or how other schedulers are using the cluster resources (Omega).

5.1.3 Distributed schedulers

At one extreme of the design space there are completely centralized schedulers (Section 5.1.1). At the other, there are decentralized fully independent schedulers, that do not use any central scheduler or component. In this category we find Sparrow [79] and Apollo [16].

In Sparrow [79], the number of distributed schedulers can be as many as needed (in order to provide the best scheduling latency) and are stateless. Since schedulers are stateless, each time a job needs to be scheduled, they send probes to randomly chosen workers. For the number of workers probed, this work leverages the power of two choices [73], probes will be then twice as much the number of tasks in a job. Sparrow is the first scheduler that proposes to have queues in the workers, these queues are unbounded and ordered in a FIFO way. The probe that gets to the head of the queue first executes the next task in the job, this is called late binding, as opposed to early binding where a task is assigned a resource in the cluster from the scheduling. In a way, Sparrow is the anti-thesis of a scheduler, because the scheduling is extremely lightweight (random) and distributed schedulers are completely independent and stateless. As a consequence, the scheduling latency in Sparrow is the best among state-of-the-art schedulers due to having almost no scheduling logic and to being able to spawn as many schedulers as one per job. This is in line with Sparrow's goal, which is to provide fast scheduling for extremely short jobs with tiny tasks [78] (tens of milliseconds of runtime). At the same time, this stateless scheduling is the main shortcoming of Sparrow because datacenters workloads are made up of heterogeneous jobs (not just tiny), and even if the short jobs make up most of the workload, long jobs take most of the resources which will lead to a very bad case of head-of-line blocking (Section5.2). Furthermore, conditions like having a large cluster and a small number of tasks do not favor the sampling method. Another distributed scheduler is Tarcil [32], the main idea is to do better sampling by providing statistical guarantees. It adjusts the sample size based on the load and uses admission control when these guarantees are unlikely to be met.

Distributed schedulers in Apollo [16] make their scheduling decisions independently like in Sparrow. Same as Sparrow, workers in Apollo have FIFO-ordered queues. Unlike Sparrow, the scheduling is done in a stateful manner. Apollo's main contribution is a shared wait-time matrix that schedulers use to place tasks. The information in this matrix is obtained from the workers' estimated waiting times and is refreshed and re-estimated periodically. Apollo is the first scheduler that proposes the notion of future resource availability. The scheduling is done based on *tokens* that express a task resource requirement to execute (similar to Borg). Tasks are matched to servers as a variation of the algorithm in [37]: a task is proposed to the server with the earliest completion time, in the case of more than one task being proposed, the server choses the one that benefits the most from the match (biggest completion time saving). To increase scheduling latency, Apollo schedules tasks with similar priorities in batches. Schedulers in Apollo also take into account other constraints, e.g. task priorities, a capacity management policy, task dependencies, and fault tolerance. Similar to Omega, Apollo suffers from conflicts among its schedulers. Unlike Omega, Apollo defers its correction mechanisms to after a task is dispatched to a server, this is possible because servers have queues. We elaborate on the correction mechanisms in Section 5.5. The distributed nature of Apollo is made to provide a good scheduling latency. Furthermore, because distributed schedulers in Apollo are stateful, their scheduling decisions are more informed than the sampling based scheduling in Sparrow. However, Apollo does not make a distinction in the

nature of jobs, and uses the same mechanisms to schedule all of them.

Distributed schedulers were born as the need for scalability became dominant and they provide very good scheduling latency compared with centralized schedulers. However, in a fully distributed setting it is hard to avoid bad scheduling due to lack of visibility of the cluster and/or to conflicts among schedulers. Another issue with distributed schedulers is that they do not take into account the heterogeneity of the jobs, scheduling all jobs in the same way. This will, as a consequence, incur in problems like the head-of-line blocking (explored in Section 5.2).

5.1.4 Hybrid schedulers

To get the best of both worlds, hybrid scheduling has been proposed and used in Hawk [29], Mercury [60] and Eagle [27]: having one centralized scheduler to do careful scheduling and many distributed schedulers to lower the scheduling latency.

Mercury and Hawk were the first papers to propose a hybrid design to alleviate the trade-off between scheduling scalability and quality placement. Both were published simultaneously in the same conference.

Mercury classifies containers (resource allocation units where tasks run), as being either (a) *guaranteed* run to completion as soon as they arrive to a server; or (b) best-effort *queueable* containers that are queued in a server's queue and can be killed by *guaranteed* containers. A centralized scheduler performs careful placement for *guaranteed* containers, supporting scheduling policies such as capacity [7] and fairness [8]. A number of distributed schedulers place *queueable* task containers trying to minimize queuing delay, to this end they rank nodes according to the estimated queue waiting time and the elapsed time since a *queueable* container executed successfully (not preempted by a *guaranteed*). Queueable containers can enforce application level quotas. A single application can have a mix of guaranteed and queueable containers where to run its tasks. Moreover, a queueable container could be promoted to guaranteed to avoid losing computation when killed. The biggest advantage of Mercury is that, in principle, the proposed scheduler design provides the advantages of both centralized and distributed schedulers. However, depending on the conditions this does not always hold, for example if short latency-sensitive jobs (that are majority) all need *guaranteed* containers the scheduling latency of a centralized scheduler will not be sufficient. Furthermore, it is not clear whether jobs have the information necessary to make an informed choice with respect to the appropriate container type.

5.1.5 Hawk, Eagle, and Kairos

The first two of the schedulers presented in this dissertation, Hawk and Eagle, are hybrid. The third scheduler, Kairos, has a very lightweight centralized algorithm.

Scheduler	Type	Level	Central policy	Distributed policy	Coordination/Conflicts	Node queue	Node queue policy
Quincy	centralized	scheduler	combinatorial optimization	-	-	-	-
Delay	centralized	scheduler	fairness & locality	-	-	-	-
Mesos	distributed	meta-scheduler	-	pluggable	resource allocator offer-/accept	-	-
Omega	distributed	meta-scheduler	-	pluggable	shared state + Optimistic concurrency	-	-
Yarn	centralized	meta-scheduler	capacity/fairness	-	-	-	-
Sparrow	distributed	scheduler	-	random probes	-	unbounded	FIFO
Apollo	distributed	scheduler	-	LWL + constraints	shared wait-time matrix + deferred correction	unbounded	FIFO
Borg	centralized	scheduler	scoring + constraints	-	-	-	-
Mercury	hybrid	scheduler	capacity/fairness (guaranteed)	LWL (queueable)	shared state + random topk	bounded exp. time	FIFO
Hawk	hybrid	scheduler	LWL (long jobs)	random probes (short jobs)	-	unbounded	FIFO
Yaq-C	centralized	scheduler	pluggable	-	-	bounded nr. tasks	SRPT
Yaq-D	distributed	scheduler	-	pluggable	shared state + queues	unbounded	SRPT
Tetrisched	centralized	scheduler	combinatorial optimization	-	-	-	-
Eagle	hybrid	scheduler	LWL (long jobs)	sticky probes (short jobs)	-	unbounded	SRPT (short jobs)
Firmament	centralized	scheduler	combinatorial optimization		-		
Big-C	centralized	scheduler	capacity + preemption	-	-	-	-
Medea	centralized	scheduler	combinatorial optimization (long jobs) + task-based (short jobs)	-	conflict/resubmit	-	-
Kairos	centralized	scheduler	load balance + LAS	-	-	bounded nr. tasks	LAS

Table 5.2 – Different scheduling designs at a glance (ordered by publication date). -: not aplicable. LWL: Least Work Left and variations. SRPT: Shortest Remaining Processing Time first. LAS: Least-Attained Service first.

Hawk and Eagle Similar to Mercury, Hawk proposes to blend the centralized and distributed scheduling designs to get good scheduling placement and scalability. In contrast to Mercury, Hawk classifies jobs according to their duration. In Hawk, long jobs are scheduled in a centralized way because they: (a) are not big in number and (b) consume most of the resources, therefore careful scheduling is done for jobs that consume most of the resources while not hurting latency-sensitive jobs (short ones).

Hawk does not suffer from scheduling scalability issues as in centralized schedulers (Section 5.1.1) because most jobs are scheduled in a distributed way. Unlike distributed schedulers (Section 5.1.3), Hawk takes into account the heterogeneity of the jobs by providing informed conflict-free scheduling for long jobs.

Eagle adopts the hybrid scheduling from Hawk and augments it with two novel techniques, contrasted with the state-of-the-art in Sections 5.2 and 5.3.

Kairos 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.

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 the centralized component in Kairos could also be implemented by means of many distributed schedulers. Such a distributed scheduler would implement the load balancing technique presented in Kairos. To this end, the state of the worker nodes could be gossiped across the system, e.g., as in Apollo and Yaq, or shared among the distributed schedulers, e.g., as in Omega. Existing techniques like randomly perturbing the state communicated to different schedulers [16] and atomic transactions over the shared view of the cluster [88] could be used to limit or avoid concurrent conflicting scheduling decisions by different schedulers.

5.2 Avoiding head-of-line blocking

A very common scenario in datacenter scheduling is head-of-line blocking, i.e. short latency-sensitive jobs get stuck behind long-running ones. Given the characteristics of the workloads, avoiding this problem is of paramount significance. Schedulers in the literature deal with this issue in different ways, and in this section we cover different mechanisms that help alleviating head-of-line blocking.

Scheduler	Locality	Fairness	Quality placement	Low sched. latency	No head-of-line	Job-awareness	No estimations
Quincy	✓	✓	✗	✗	✗	✗	✓
Delay	✓	✓	✗	✗	✗	✗	✓
Omega	-	✗	✓	✓	✗	-	-
Yarn	✗	✓	✓	✗	✗	✗	✓
Sparrow	✗	✗	✗	✓	✗	✗	✓
Apollo	(✓)	♢	✓	✓	✗	✗	✗
Borg	(✓)	✗	✓	✗	♢	✗	✗
Mercury	✗	♢	✓	♢	✗	✗	♢
Yaq	✗	✗	✓	✗	♢	✗	✗
Tetrisched	(✓)	✓	✓	✗	✗	✗	✗
Firmament	(✓)	(✓)	✓	✓	✗	✗	✗
Big-C	✗	✗	✗	♢	✗	✗	♢
Hawk	(✓)	(✓)	✓	♢	♢	✗	♢
Eagle	(✓)	(✓)	✓	♢	✓	✓	✗
Kairos	✗	✗	✓	✓	✓	✗	✓

Table 5.3 – Primary challenges addressed by datacenter schedulers over the past 10 years (ordered by publication date). (✓): could be supported. ♢: partially, only for some jobs or among users. -: does not apply.

5.2.1 Priorities

A possible solution to avoid head-of-line blocking is to set different priorities to jobs, for instance by giving more priority to latency-sensitive jobs. Schedulers like Omega, Apollo and Borg support priorities among jobs. Mesos supports priorities among frameworks.

In Mercury, *guaranteed* tasks have priority over *queueable* ones. This priority is enforced by killing (Section 5.2.5). Sparrow's authors suggest that priorities could be supported by adding different queues, one per priority and consuming these queues in order. However, in this setting, it is more practical to reorder a single queue rather than maintaining a set of queues that could potentially degenerate to having one queue per enqueued probe. Priorities can also be incorporated to systems that use MILP for scheduling, as a part of the cost function.

However, these schedulers do not base their job priority on its size. This mechanism is therefore not enough to avoid head-of-line blocking: a long running job can have more priority than a short one hurting its running time greatly. Moreover, relative to its length, a long task can tolerate being delayed by running a short task before it.

5.2.2 Partitioning and reservation

Another way of avoiding head-of-line blocking is to partition the cluster statically for each category of jobs.

In the context of sharing the cluster among different frameworks, Mesos proposes an alternative to partition the cluster statically and allows frameworks to have a guaranteed share of the cluster, this can be enforced by its allocation module.

Tyrex [40] aims to avoid head-of-line blocking by organizing the workload in classes depending on task runtimes, and by assigning different classes to disjoint partitions of worker nodes.

Similar to Hawk's reservation, PerfIso [55] proposes to reserve a set of *buffer* idle cores that only primary (latency-sensitive) applications can use in peak load.

The problem with partitioning is that the portion of the resources used by a given job category is not static over time.

5.2.3 Capacity

A less rigid way of partitioning the cluster statically is to use capacities. The notion of *capacity* is often applied in some systems to depict a configurable share of resources that a given set of applications, user or framework has over the cluster.

In Yarn's CapacityScheduler [7], each group of users (potentially also applications) are assigned to queues. Each queue has two configurable levels of capacity: the original capacity and the maximum capacity, the latter being strictly equal or bigger than the former. When a new job arrives, the scheduler tries to schedule it using the capacity assigned to its queue; if that capacity is fully utilized it uses free capacity from other queues up to the maximum threshold. Apollo and Mercury provide capacity guarantees in a similar way. Apollo uses *opportunistic scheduling* to take advantage of idle resources over a non-peak period. Tasks can execute in regular or opportunistic mode. Opportunistic tasks have less priority than regular tasks. If a task has sufficient tokens to execute it executes in a regular mode, else in opportunistic mode. In contrast, tasks in Mercury are queueable or guaranteed depending on the nature of the job.

Big-C leverages the CapacityScheduler to give priority to short jobs over long ones. It assigns a higher capacity to short jobs (e.g. 95% of resources guaranteed) while long jobs can use up all of the resources if free. To avoid head-of-line blocking Big-C introduces the use of preemption when needed (Section 5.2.5).

Although capacities are not normally used to separate jobs by their running time, this technique could potentially be used to avoid some head-of-line blocking, like in Big-C. However, in the absence of preemption, if long jobs can take up all of the free resources, head-of-line can still occur. Furthermore, capacities are fixed in configuration, while they should instead adapt to the workload dynamics: depend on the load and the percentage of short/long jobs

executing.

There are other configuration parameters in the CapacityScheduler that can help to compensate for this lack of adaptation, but it is not clear if the user has the information to tailor the perfect set of values for configuring the share of the cluster. Also, as mentioned before, capacities are mostly used for another purpose.

5.2.4 Queues and reordering

For highly loaded systems there are queues in the workers, in a centralized scheduler or in both. Yaq [81] proposes to leverage queues to avoid head-of-line blocking by reordering them other than in a FIFO manner. For distributed schedulers the solution proposed is Yaq-D, for centralized ones, Yaq-C. Given that many schedulers [16, 43, 59] make use of runtime estimates, mostly based on recurring jobs [35, 25], good candidates are shortest job first (SJF) or shortest remaining time first (SRPT) [48]. Yaq then combines these queue reordering techniques with admission control (Section 5.5.1). Mercury [60] also mitigates head-of-line blocking by means of load shedding, we discuss this in Section 5.5.1. Techniques in Yaq can be applied to schedulers that use runtime estimates to avoid head-of-line blocking. Nevertheless, queue reordering alone without preemption or a reservation-like mechanism cannot avoid completely the problem, because once a set of long tasks have taken over the cluster, short jobs arriving after will experience head-of-line blocking.

For both cases a worker executes always the Shortest Remaining Processing Time (SRPT) first, which is the best size based policy studied in queueing theory [48] . Eagle combines this same queue reordering with sticky probes in order to execute short jobs first and avoid completely the head-of-line blocking.

A recent system, in [52], aims to prioritize short jobs by organizing them in different 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. Although this system takes into account the runtime of a job, its design also suffers from head-of-line blocking: during a period of high load there could be a job (or combination of jobs) with sufficient tasks to fill the capacity of the queue with the highest priority, jobs in the other queues are by design long leaving thus all upcoming short jobs to starve until tasks in long jobs are done. A system like this would benefit from preemption: thresholds could be used to decide when to preempt a task. Even so, parameters need to be carefully designed: number of queues vs capacities; it is worth noticing that these parameters are also static through time, property that might not necessarily hold for the workload.

5.2.5 Killing/preempting

Some schedulers that support priority and capacities may also provide killing as a way of enforcing their policy [51, 88, 16, 102]. The Mesos allocation module can revoke resources when the cluster gets filled with a long tasks for example. In Omega, Apollo and Borg, jobs can preempt (kill) other jobs with lower priority if the cluster is full. In Omega, is possible that a scheduler kills a task from another scheduler, since there is no synchronization between schedulers it is not clear how this is dealt with in the second scheduler, it will ultimately depend on the policy. In Apollo, all of the schedulers use the same policy so this issue is less of a concern. Additionally, opportunistic tasks can be killed if the server is under resource pressure. In Yarn's CapacityScheduler the killing is optional and configurable. If killing is enabled, when the corresponding *capacity* is not ensured for a queue, jobs from other queues (occupying more share than their *capacity*, can be killed.

Some schedulers try to avoid the number of killings, because it implies discarding whatever progress the task has made and restarting it from scratch later or in another node. In Borg, to avoid cascading killing, jobs have different priority bands and cannot kill other jobs in their band. Optionally, users can be notified before their jobs are killed. Apollo's placement algorithm will also try to avoid the number of tasks to be killed if any.

An alternative to killing is preempting. Although preemption has been studied previously in literature [58, 5, 25], it was not used in practice or studied further because of the complexity of its implementation and, ultimately, the cost it incurs. Additionally, there are other possible complications of fully stopping a task, notably, if an application manager (Yarn) is in charge of dealing with fault tolerance it will try to spawn other copies of the same task. Big-C proposes to implement preemption using Docker containers and, when needed, gradually decreasing the resources assigned to it rather than stopping it completely.

5.2.6 Hawk, Eagle and Kairos

Hawk, Eagle and Kairos try to avoid head-of-line blocking in a fundamentally different way from most of the schedulers presented in this section, because their scheduling is based on job runtimes.

Hawk Unlike papers discussed in this Section, Hawk proposes to schedule jobs differently according to their size: long or short. Additionally, Hawk uses a combination of partitioning and job stealing to avoid head-of-line blocking. Hawk, in a similar way to capacity and partitioning techniques, reserves a small portion of the cluster to run only short jobs while the rest of the cluster runs both long and short jobs. This technique helps specially when at high load, the cluster is taken over by long tasks, because prevents long jobs from clogging all of the resources.

Stealing is also used to avoid the head-of-line blocking, more specifically, a free node will *s*teal

short tasks that are queued behind a long job in a randomly chosen node. While work stealing alleviates the head-of-line blocking, is not sufficient to eliminate it because even if short tasks are relocated, they may have waited behind a long task.

Eagle Eagle, like Hawk, schedules long and short jobs separately. It also uses reservation to avoid occupying the cluster with only long tasks. In addition, Eagle introduces a dynamic partitioning with Succinct State Sharing (SSS) technique in which short tasks will therefore completely avoid the nodes that run long tasks at any moment, this also helps to make use of free resources whenever they are available. Additionally Eagle uses queue reordering (SRPT) among short jobs, similar to Yaq-D, but combines it with Sticky Batch Probing (SBP) to provide job-awareness (Section 5.3).

Kairos Kairos is a scheduler without any *a priori* assumptions (on job runtime or type). It uses preemption to avoid the head-of-line blocking: as soon as a task arrives to a node it preempts the running task that had the most service so far. Moreover, the quanta parameter is set to fit most of short tasks: they will run to completion. On the other hand, tasks whose duration exceeds the quanta will only experience a delay at most n times their duration where n is the number of tasks enqueued in a node divided by number of cores. Kairos guarantees then that here is no head of line blocking for most of short tasks. Interestingly, Kairos also alleviates potential long tasks head-of-line blocking.

Scheduler	Priorities	Capacity	Reservation	Work stealing	Queue reordering	Killing
Mesos	(✓)	-	-	-	-	(✓)
Omega	(✓)	-	-	-	-	(✓)
Sparrow	(✓)	-	-	-	-	-
Apollo	(✓)	(✓)	-	-	-	(✓)
Yarn	-	(✓)	-	-	-	(✓)
Borg	(✓)	-	-	-	-	(✓)
Mercury	(✓)	(✓)	-	-	-	(✓)
Yaq	(✓)	-	-	-	✓	-
Big-C	-	✓	-	-	-	♢
Hawk	(✓)	-	✓	✓	-	-
Eagle	(✓)	-	✓	-	✓	-
Kairos	-	-	-	-	✓	♢

Table 5.4 – Different techniques that help avoiding the head-of-line blocking. ✓: used to avoid head-of-line blocking. (✓): could be used to avoid head-of-line blocking, but is not an explicit part of the scheduling policy. -: not used. ♢: preemption.

5.3 Providing job-awareness

Datacenter schedulers in the literature schedule primarily at the task level. However, given the task-parallel nature of jobs, the running time of a job is determined by the end time of the task that took the longest to finish. A task in a job can finish much later than others because of different reasons, in this dissertation and this section we focus on scheduler induced stragglers.

In distributed stateless schedulers like Sparrow it is straightforward to see how the lack of job-awareness can affect especially short jobs. It suffices for a task from a short job to be behind a long one. This case can happen quite often because schedulers in Sparrow do not have visibility on the cluster resources status. Furthermore, in Sparrow, a probe only executes one task, the scheduler does not take advantage of reaching to the head of the queue to run more than one task.

For distributed stateful schedulers like Apollo, the scheduler induced stragglers can happen when either two schedulers conflict or when the estimated waiting time in a node is far from its real runtime. We motivate in Chapter 4 that estimation based approaches can suffer from misestimations.

Schedulers made for DAG-like applications [57, 74], where a job's workflow is represented by a directed acyclic graph (DAG), take into account dependencies between stages in jobs. These schedulers therefore will try to schedule in a job-aware manner. However, these schedulers by design, are not able to cope with scheduling scalability. Gang schedulers are supported by Mesos and Omega, their schedule is an all-or-nothing approach, therefore, all tasks start at the same time. If a centralized scheduler would try to schedule in a job-aware fashion, the main danger is the misestimations. Some schedulers might experience indirect job-awareness, like Yaq (because queue reordering can bring tasks of the same job to the head of the queue to be executed at the same time).

Some systems in the literature rather find mechanisms to deal with stragglers. For example, Yaq and Mercury migrate tasks that have not started yet to re-balance the load. LATE [108], Mantri [6], Dolly [4], Hopper [84] and DieHard [89] use techniques like restarting or cloning tasks to cope with stragglers due to mis-estimations or due to unexpected worker nodes slowdowns or failures. While these techniques help with runtime stragglers, they are not part of a proactive job-aware scheduling.

Bottom-line, schedulers in the literature either do not expressively schedule at the job level (rather schedule one task at a time and deal with lack of job-awareness by means of correction mechanisms), or they do not provide scheduling scalability.

5.3.1 Hawk, Eagle, and Kairos

Hawk Although Hawk does not provide job-aware scheduling, the algorithm used for the long

jobs could be extended to schedule also at the job level. Nonetheless ,short tasks in Hawk do suffer from lack of job-awareness in their scheduling.

Eagle Eagle is the first scheduler that does an effort to schedule tasks in a job-aware manner. Sticky probes in Eagle will run a job to completion once it started rather than executing tasks of many jobs. It is, thus, job-aware. Ideally, having probes in all of the nodes would make Eagle achieve the highest job-awareness, but in that case SRPT would become too aggressive with respect to long running jobs.

Kairos In Kairos, the distributed LAS policy will ensure that all tasks in a job, execute at the same time for at least a quanta of time which will make, for most of the short jobs, that they finish as fast as possible. In the case that one or more tasks are not done during that time, they will be back in the queue and with very high probability execute together again in a wave. In contrast to Eagle, the job-awareness provided in Kairos exploits more the parallelization than serialization.

5.4 Improving Assumptions

Most datacenter schedulers assume that they have information about incoming jobs (e.g., expected runtime, required resources to run). In Chapter 4 we argue that the dependability on runtime estimates can lead to important job runtimes detriment. In this Section we explain different assumptions needed by datacenter schedulers in the literature.

5.4.1 Knowledge required/assumptions

To provide better scheduling under high-load, most schedulers [16, 60, 81, 19] need to have some assumed knowledge about the incoming job, some need the approximate runtime of tasks, some need the resource requirements and some need both.

Runtime

Table 5.5 shows the range of assumptions schedulers take with respect to the runtime of a task. Most of the literature assumes to know an estimate of the runtimes at the task level. Apollo relies on a per-task runtime estimation to build its wait-time matrix. This information is updated periodically during runtime and to face misestimations, correction mechanisms are applied. Similar to Apollo, Yaq needs per-task estimations for all of the jobs in order to apply queue reordering (SRPT). Mercury needs task runtime estimates of queueable containers to schedule incoming queuable containers in the nodes that have the queue with the least work left. Furthermore, the load shedding technique in Mercury is based on the estimated time left in the queues.

Big-C needs to know whether a job should be treated as a long or a short one and the approximate percentage of resources needed by short and long jobs.

The degree at which misestimation impacts the original intended scheduling is directly related to the amount of estimation needed. In this sense, Apollo and Yaq will have more impact due to misestimations than Big-C.

Resource requirements

Most of schedulers used in production (e.g., Yarn, Borg) provide a way for the user to specify resource requirements for each job. This information is used for making scheduling decisions. In Borg any incoming job has resource requirements in terms of CPU and memory. In Yarn, the size in terms of CPU and memory for each container where a task will run is configurable by the user.

Understandably, to ensure a good job run, the users over commit [31] the resources they need, especially in memory. On top of that, users hardly know what is the best configuration for their job, because ultimately it also depends on other jobs running at the same time in the cluster.

Scheduler	None	Short/long distinction	Long task runtime	All tasks runtime	Resource requirements
Delay	✓	-	-	-	-
Sparrow	✓	-	-	-	-
Apollo	-	-	-	✓	-
Mercury	-	✓*	✓*	-	-
Yaq	-	-	-	✓	✓
Borg	-	-	-	✓	✓
Big-C	-	✓	-	-	-
Hawk	-	✓	✓	-	-
Eagle	-	✓	✓	✓	-
Kairos	✓		-	-	-

Table 5.5 – Knowledge required for schedulers to work. *: in Mercury distinction between queueable and guaranteed and the estimated runtime of a queueable task are needed.

5.4.2 Non a priori assumptions

Datacenter schedulers that do not assume any *a priori* knowledge are very few. Sparrow and Quincy, described in Section 5.1, do not depend on *a priori* information about the incoming jobs.

In Tyrex [40], 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 [52] prioritizes short jobs by organizing jobs in 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.

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.

5.4.3 Hawk, Eagle, and Kairos

Depending on the level of information and the capacity to predict a task runtime, this dissertation proposes three schedulers that assume to know all tasks estimates (Eagle), non *a priori* knowledge (Kairos) or information only about long jobs (Hawk).

Hawk The knowledge that Hawk assumes about jobs is 1. whether an incoming job is long or short and 2. an estimation of the runtime of long tasks.

Eagle To benefit from Eagle's techniques, more notably the queue reordering, an estimation of the runtime of all tasks is needed.

Kairos Similar to other systems presented in the previous subsection 5.4.2, Kairos does not assume any prior knowledge. In contrast to these systems, Kairos uses preemption to allow an incoming task to run as soon as it arrives on a worker node. It offers short tasks the possibility of completing with limited or no waiting time, even in high-utilization scenarios.

5.5 Correction Mechanisms

Scheduling in datacenters is not an easy task, because of the challenges previously mentioned. Some schedulers, like Tetrisched and Firmament, put more effort in trying to find the most optimal placement, for example, given SLO constraints. However, issues like misestimations [80] (both in runtime and in resource requirements) and stragglers turn some -initially good- scheduling decisions into bad scheduling. Therefore, datacenter schedulers employ diverse correction mechanisms to fix unavoidable bad scheduling. This Section explores most commonly used and classify them as being proactive or exploited after scheduling is done.

5.5.1 Proactive

Proactive correction mechanisms act before the scheduler binds a particular task to a worker, as opposed to mechanisms that do it during runtime after a task is assigned to a worker.

Late binding

Distributed schedulers in Sparrow, Hawk and Eagle use probing, a late binding technique, to decrease the amount of mistakes in scheduling. Late binding 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.

In contrast, other distributed and centralized schedulers do early binding. If early binding is based on runtime estimations (e.g. Apollo, Borg), the scheduler will inevitably need to compensate for mistakes due to misestimations and other unforeseen events.

Admission control

Admission control could be seen as a way of doing late binding without sending probes. In Yaq-C and Kairos, the queues in the workers are bounded by a specific maximum number of tasks. In Mercury the queues are bounded by a given expected execution time. Another queue bounding technique in Yaq bounds queues based on the expected delay.

Load balancing

Balancing the load in nodes queues is another proactive mechanism. Mercury and Apollo distributed schedulers use the estimated wait time in the worker queues to balance the load among workers. Additionally, Apollo adds a small random number to each completion time estimation to help avoiding scheduling conflicts. Kairos centralized scheduler enforces that resources do not get wasted based on the number of tasks present in each worker.

5.5.2 After scheduling

Some correction mechanisms are made after the task is scheduled to run in a given node.

Killing

A way of dealing with unexpected bad scheduling is killing.

The allocation module in Mesos can revoke tasks resources by killing them. This module provides a grace period to schedulers for cleaning up. In Borg, when there are not enough free resources for an upcoming job, jobs with higher priority preempt (kill) jobs with lower

ones. A task that is killed could potentially generate a cascade of killings. To prevent this, Borg disallows tasks from *production* priorities to preempt one another. In Apollo, *opportunistic* tasks can be preempted or terminated by non-opportunistic tasks.

Anti-starvation mechanism

Schedulers that use a reordering based on the task runtime estimation, such as SRPT, need to provide an anti-starvation mechanism so that the tasks that last longer dont starve. This is the case for Yaq, Eagle and Kairos. The simplest mechanism consists in limiting the number of times a task has been bypassed or preempted with a threshold. Yaq uses this hard threshold plus the time of the earlier submitted job as a tie-breaker. Another threshold could be relative to the task length, as in Eagle. Mercury has the possibility of a task to be upgraded from queueable to guaranteed if it has been killed too many times.

Conflict management

Distributed schedulers can have conflicts because they make independent scheduling decisions. In Omega, if there is a conflict between two schedulers, one scheduler gets the resources and the other scheduler needs to reschedule according to its conflict management policy (either do incremental commits or all-or-nothing scheduling).

Apollo has a deferred correction mechanism: it allows resolving conflicts between independent schedulers only if they have a significant impact. Apollo also adds a small random number to each completion time estimation. This random factor helps reducing the chances of conflicts by having schedulers choose different, almost equally desirable, servers.

Sparrow does not deal with conflicts because if schedulers probe the same nodes those probes get enqueued in a FIFO order.

5.6 Other desired properties

Additionally to the main challenges explored in previous sections, datacenter schedulers can focus on other aspects. Table 5.6 lists the main aspects that schedulers focus on, apart from the ones studied in this thesis. We argue that some of these functionalities (i.e.. isolation, fault tolerance) should be taken care of by the framework rather than the scheduler. On the other hand, other functionalities can be provided in a very straightforward manner for centralized schedulers. Examples of these are hard constraints, priorities, locality. Similarly, the centralized components in Hawk, Mercury, Eagle and Kairos can provide those functionalities for jobs that are scheduled in a central way. Finally, there are a couple of properties (i.e. SLOs, fairness, soft constraints) that are most of the times subjective to the user and their impact might not be as much as it is perceived.

Scheduler	Constraints	SLO	Priorities	Isolation	Fault tolerance
Delay	✗	✗	✗	✓	✓
Mesos	-	-	✓*	✓*	✓*
Omega	-	-	✓	✗	-
Sparrow	✓	✗	✗	-	✗
Apollo	✓	✗	✓	pluggable	✓
Borg	✓	✓	✓	✓	✓
Yarn	✗	✓	✗	✓	✓
Mercury	✓(guar.)	✗	✓	✓	✗

Table 5.6 – Additional properties offered by datacenter schedulers. We list only schedulers that offer properties not discussed previously. -: does not apply. *: provided at the framework level. guar.: for guaranteed jobs. sched.: schedulers.

5.6.1 Fairness

Fairness is a functionality that aims to give different categories of users their *rightful* share of resources. The definition of how this is done is subjective to the user and configurable according to different needs in a cluster. This functionality was supported by early schedulers like Delay scheduling, Quincy, Mesos, and Omega. More recent schedulers, like Borg and Yarn CapacityScheduler, define it as Capacity. Fairness to be enforced at the user level is out of the scope of the schedulers developed for this thesis.

5.6.2 Constraints

Constraints resource preferences or requirements per job or per task, defined by users. Schedulers in the literature classify them as soft and hard. A job is not able to run if its hard constraints are not met. Soft constraints, instead, are seen as preferred resources.

The definition of constraints in practice is subjective because it is left to the user. Recent work [77] argues that users can't determine which configuration parameters to set and which hardware to use to optimize runtime. Moreover, most of the times users just try to get the most out of the system, typically by asking for more resources than actually needed [31, 82]. In fact, this could hurt scheduling performance rather than helping it, especially given the memory variability over execution time. It has been shown, for instance, that tasks can run with significantly less memory that they would ideally want while only paying a moderate performance penalty [56].

5.6.3 Data locality

Early centralized schedulers like Delay Scheduling considered data locality for their scheduling decisions: trying to place the task in the worker that contains the data that it will process. However nowadays, with commodity servers having more disk and better datacenter networking capacities, locality is less of a worry. This holds especially for tasks that process a small amount of data. There has been also a proposal in the literature [78] to break jobs into tiny tasks that complete in hundreds of milliseconds, which makes the need of data locality even less important.

Nevertheless, some systems can include data locality as a plus, in the case when a task has to process a large amount of data, e.g. for the long jobs. Apollo, Borg, and Mercury can support data locality constraints. In Hawk and Eagle it is straightforward to provide data locality for long jobs in the centralized scheduler.

5.6.4 Isolation

In order to prevent interference when sharing resources in a node, it is important to provide isolation between the tasks that run concurrently in it. We argue that this is a functionality that corresponds more to the framework than to the scheduler. Mesos provides isolation among different frameworks running on top of it. Apollo can accommodate mechanisms employed by other systems. Schedulers that are implemented on top of systems like Spark and Yarn (for example Mercury, Hawk, Eagle, and Kairos) have an implicit isolation mechanism with the containers that the framework allocates. Similarly, a Borglet in Borg is in charge of manipulating the settings of the containers that run Borg tasks and additional mechanisms controlled by the Borg master to avoid interference.

5.6.5 Fault tolerance

Due to machine crashes and other unexpected events, it is important for a datacenter to provide fault tolerance. We can have fault tolerance at different levels: the failure of the master or coordinator, the failure of schedulers and the failure of tasks.

In the case of failure of the master, Mesos uses Zookeper to recover its status. Borg uses Paxos to manage backups of the status kept centrally and also for the election of a new master in case of an outage. To handle the failure of a stateless scheduler, in Sparrow, frameworks need to detect the failure and connect to a backup scheduler. We argue that schedulers that keep a cluster state can easily include the approach of Mesos or Borg for providing fault tolerance.

In frameworks like Yarn, the functionality that manages the failure of tasks ultimately became part of an application manager or equivalent, rather than part of the scheduler.

5.6.6 SLO

The main focus of some schedulers like Tetrisched is to provide a higher SLO attainment. Tetrisched uses a reservation system and plans *ahead* by providing a job the possibility of waiting to get a busy preferred resource or fall back to less preferred options. We argue that in practice few systems count with SLOs specifications, and when these are specified most of the times they are relative and can be relaxed. For example, a user could specify an SLO for a job that is supposed to run in peak hours. In reality this job, given the load in the cluster, would take much longer to finish than if the SLO is relaxed to be just slightly out of peak hours.

5.6.7 Priorities

In Section 5.2.1 we discussed how priorities in datacenters can be used to avoid the head-of-line blocking. However, originally priorities are meant to establish a relative importance among jobs, for example, production jobs should have more priority than other best-effort jobs.

Schedulers proposed in this thesis give priorities to short jobs over long ones, because they are latency sensitive. Additionally, Hawk and Eagle can add priorities among long jobs in the centralized scheduling policy. Finally, Kairos could be extended to incorporate priorities at the node level. For example, priorities could be added to the attained service time to determine the order the queues.

5.7 Benchmarking

Researchers have, to the best of their effort, tried to evaluate fairly their schedulers. However this is not an easy task, because on the one hand datacenter schedulers are linked to different systems architectures that ultimately restrict the scope of the scheduler. On the other hand, the workloads used to evaluate differ and are mostly synthetic representations of real executions. Additionally, the infrastructure to test scalability in practice is limited, even for researchers that work in big data companies.

5.7.1 Software stack

Datacenter schedulers normally operate with a complex software ecosystem composed of many layers of software [87] which ultimately depend on one another to make tasks run.

This dependency can even be recursive: Spark can run on top of other systems like Yarn and Yarn can run on Spark. This can lead to a potentially intricate layering of schedulers. Only in the case of Mesos, there is always other software on top, because it is a low level interface. Nevertheless, that layer that runs on top might be running more than one scheduling layer.

Scheduler	Implementation	Simulation	Nodes sim.	Nodes impl.
Delay	Hadoop	-	-	100
Mesos	C++. Run on top: Hadoop, Spark, Torque, MPICH2	-	50000	96
Omega	-	✓	~15000	-
Sparrow	Java & Spark plug-in	✓	10000	110
Apollo	Microsoft-specific	✓	-	20000
Borg	Google-specific	✓	-	-
Mercury	Yarn	-	-	256
Yaq	Yarn	-	-	80
Big-C	Yarn	-	-	23
Hawk	Java & Spark plug-in	✓	up to 150000	100
Eagle	Java & Spark plug-in	✓	up to 23000	100
Kairos	Yarn	✓	up to 23000	30 (120 cores)

Table 5.7 – Evaluation of schedulers. Implementation column show at a glance the language and framework used for implementing the scheduler. Simulation column indicates whether there was a simulator for the scheduler. To get an idea of the scale, number of nodes simulation and implementation show the *largest* amount of nodes used for evaluation. -: not provided or does not apply.

Another difficulty that researchers face in the area is that only some amount of this software is open source.

Ultimately, if schedulers were to be implemented from scratch, without depending on other system components like in [42, 51], they could be highly optimized and have higher *net* performance. However in this case, they would have to trade on portability, flexibility and user adoption. This trade-off is even harder to make for companies that depend hugely on backwards compatibility.

5.7.2 Scale used to test

Table 5.7 shows the scale (of both the number of workers and the number of jobs) at which some schedulers were tested, along with the language and framework used for their implementation. The table also shows if simulation was used for testing scalability.

The scheduler that has been evaluated at the biggest scale in implementation is Apollo, with over 20000 servers in production. The other schedulers use around one or two hundred workers for testing an implemented version. To evaluate scalability, most of the schedulers use a simulator. In that regard, the usual expected number of workers is tens of thousands of nodes.

5.7.3 Workloads used

Table 5.8 compares the nature of the workloads used by different datacenter schedulers.

Since Mesos focuses on providing a way of sharing the cluster among different frameworks, the authors use a mix of Hadoop, Spark [11], Torque [91] and MPI [15] jobs. The authors test with a Facebook Hadoop Mix of 100 jobs taken from the Hive benchmark [53] (text search, simple selection, aggregation and join) following a distribution (in job size and inter-arrival times) of a Facebook production workload. This is combined with a set of 10 jobs that do text search, each with 2400 tasks that run one after the other to simulate batch jobs. Finally, a number of Spark iterative machine learning (collaborative filtering) is added. For MPI the authors run a Tachyon [92] raytracing job with 24 parallel tasks.

Omega and Borg used Google traces from production, both systems tested scalability with simulators. Omega has two versions of simulation: a high-fidelity one and a lightweight one.

In Sparrow, evaluation includes simulations with a workload that has a Poisson process arrival, is composed of 100-task jobs, and the durations of the tasks are modeled as exponential distribution with mean of 100ms. The implementation of Sparrow runs TPC-H queries on top of Spark. The workloads used in Sparrow do not follow the heavy-tailed distribution that is the norm in datacenter workloads [20, 21], because it is focused only on tiny tasks.

Mercury's workload is synthetically generated by GridMix [9] based on workload traces from Microsoft. Yaq uses a Hive-MS workload that consists of 182 queries and a synthetic GridMix workload with 100 tasks per second executed for 30 minutes. The distribution of the workload varies from homogeneous, heavy-tailed (80% 5-seconds jobs and 20% 50-seconds jobs), and exponential.

Hawk and Eagle use the Google workload to test at scale with the simulator, together with synthetic traces based in the description of Cloudera, Facebook, and Yahoo workloads from [21, 20]. For the implementation, a scale down sample from the Google trace was used as sleep jobs.

Kairos evaluates with the Google and Yahoo workloads used in Hawk and Eagle for simulation. Additionally, for implementation, it evaluates with a synthetic workload made up of a modified version of WordCount that has different job running times with different input data sizes, following a heavy-tailed distribution.

The right workload As seen in Table 5.8, most of the schedulers (Mesos, Omega, Borg, Mercury, Hawk, Yaq, and Eagle) use workloads that are synthetic based on some production workload. This approach seems a good reasonable way of proving the solution in a more general way: following an in-production distribution and scalability. Only two schedulers run real-life workloads. Among these schedulers we have: Apollo and Sparrow. Apollo is the only scheduler that evaluates the scheduling directly in production, the down side is that its baseline might not be the best to compare with. Sparrow uses TPC-H queries from the TPC benchmark. The

TPC suite is a set of benchmarks designed to evaluate databases. In particular, TPC-H is a decision support benchmark that consists of low-latency queries. It is not representative of datacenter workloads because it lacks long running jobs and the characteristic heavy-tailed distribution.

Authors evaluate their schedulers and frameworks the best they can, however there could be important differences among the workloads that each of the schedulers use. It would be therefore desirable for the community to count with representative datacenter benchmarks.

Datacenter Workload Studies

Because of privacy reasons, companies do not share details of the workloads they run. For some companies like Microsoft, even obfuscating data in order to publish it is difficult. The only publicly available trace for datacenters is the Google trace [103]. Despite this, there have been studies published on the aggregation of data from different companies such as Facebook, Yahoo, Cloudera. For example, in [21] and in [20] a K-means clusters classification is applied to the workloads in an effort to better classify jobs and describe their characteristics like map/reduce runtimes and input sizes.

The Google trace has been widely used for evaluating datacenter frameworks. For instance, it has been cited in over 400 articles [3] and there are several papers characterizing Google workloads from public traces [72, 90, 1]. However, the community would benefit from having newer versions of the workload and with more data included in the traces. An example is the duration of the tasks: the reported duration of a given task depends on how the scheduling is done. This is a chicken and egg problem because at the same time researchers make an analysis of the duration of the tasks in a workload to use it for other studies.

5.7.4 Comparison against other systems

Due to the fact that a datacenter scheduler is generally in the middle of a complex software ecosystem, it is often hard to change it and to compare it fairly with other schedulers. Another difficulty is that not all of the schedulers are open source. Figure 5.1 shows a diagram of schedulers that were compared with other schedulers. Some schedulers compare indirectly, for example Eagle to Sparrow. As Spark and Yarn are the most commonly used open source frameworks, they served as a testbed for many schedulers, leaving Apollo and Borg isolated.

Table 5.9 contains a detailed information of these comparisons. As observed, some comparisons include both simulation and implementation while some just implementation. In addition, many do not use the same code base and/or the original workloads used by the other schedulers.

The right baseline A first generation of papers of their kind compare to baselines that are, understandably, not too competitive. Among these we have Mesos, which compares to

Scheduler	Targeted jobs	Workload/nr. jobs	Max nr. tasks	Workload distribution
Delay	Hadoop jobs	Synthetic Facebook distribution based (100)	-	Heavy-tailed
Mesos	Hadoop, Spark, MPI jobs	Synthetic Facebook distribution based (100)	Facebook 400, IO-intensive 2400	Heavy-tailed implementation, Normal simulation
Omega	Google MapReduce jobs	Google production-based	-	heavy-tailed
Sparrow	MapReduce jobs	TPC-H 20k impl	100 sim, 33 impl	Homogeneous (short jobs only <100ms)
Apollo	Scope (DAG-like) jobs in HDFS-like system	GridMix Microsoft trace-driven	tens of thousands (one job)	Heavy-tailed
Borg	Google jobs (binaries)	Google prod. traces	25	heavy-tailed
Mercury	MapReduce jobs	GridMix 30's run. Microsoft production derived	not stated	heavy-tailed
Hawk	MapReduce jobs	Google trace (~0.5million) & synthetic Yahoo, Cloudera, Facebook	-	Heavy-tailed
Yaq	MapReduce jobs	GridMix 30's run.	100	Homogeneous, heavy-tailed and exponential
Eagle	MapReduce jobs	Google trace (~0.5million) & synthetic Yahoo, Cloudera, Facebook	100	Heavy-tailed
Big-C	MapReduce jobs	Google trace (2202)	- (scaled-down Google)	heavy-tailed
Kairos	MapReduce jobs	WordCount synthetic trace (100)	224	Heavy-tailed

Table 5.8 – Evaluation of schedulers. Targeted workloads and their nature. Number of jobs shows the largest number of jobs used when evaluating (to get an idea of the scale). Max nr. of tasks per job. -: does not apply.

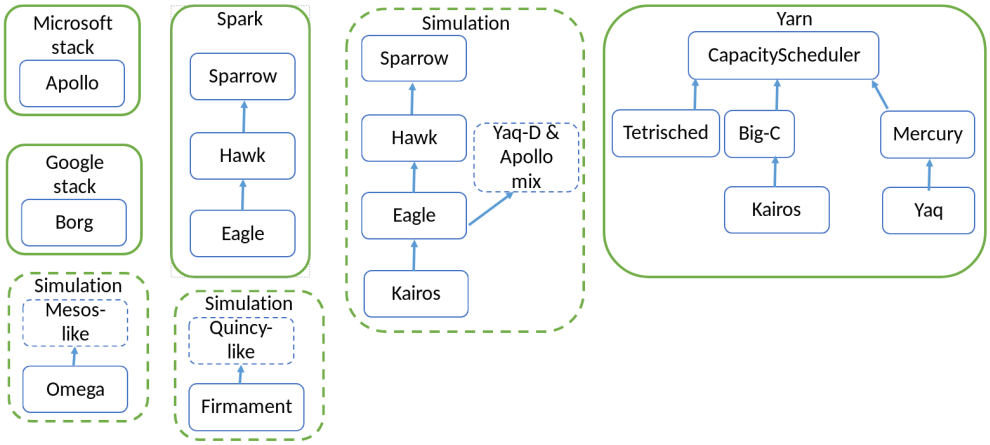


Figure 5.1 – Schedulers that evaluated against other schedulers. Arrows mean compared with. Dotted lines mean comparison without the same code base.

statically partitioning the cluster among different frameworks. Sparrow compares to a FIFO centralized scheduler. Apollo compares to what the system had before as a scheduler: a *blind* scheduler. Borg, being more an experience paper, does not have a comparison against another scheduler. Another set of schedulers (Tetrisched, Yaq-C, Big-C, Kairos) compare to Yarn CapacityScheduler and FIFOScheduler. Others (i.e., Firmament and Omega) port techniques from other schedulers, with similar goals, to their system/simulator and compare to them.

5.7.5 Benchmarking take away

From what we have described in this section, we can conclude that the way datacenter schedulers are evaluated can be improved. In particular, the research community would greatly benefit from a representative workload. Ideally, a benchmark suite that includes data and computation would help to thoroughly test schedulers.

Another aspect that helps to make a more apple to apple comparisons is the availability of the code and infrastructure. In that regard, companies have restrictions, however researchers in academia should be encouraged to open source their work to benefit the community.

5.8 Scheduling in other contexts

Scheduling is and has been a very active topic of research in other contexts like queueing theory, HPC computing and ede computing. This Section gives an overview of related work in the HPC and in the operating systems context.

Scheduler	Compared to	Impl./sim.	Same code base	Orig. workloads
Mesos	static partitioning	✓	-	-
Omega	Mesos	- / ✓	✗	✗
Sparrow	FIFO centralized	- / ✓	-	-
Sparrow	Spark	✓ / ✗	✓	-
Apollo	blind scheduler	- / ✓	-	-
Hawk	Sparrow	✓ / ✓	✓	✗
Yaq-C	Yarn	- / ✓	✓	✗
Yaq-D	Mercury	- / ✓	✓	✗
Eagle	Hawk	✓ / ✓	✓	✓
Eagle	Yaq-D-like	✗ / ✓	✗*	✗
Big-C	Yarn Capacity Sched.	✓ / ✗	✓	✗
Big-C	Yarn FIFO Scheduler	✓ / ✗	✓	✗
Kairos	Eagle	✗ / ✓	✓	✓
Kairos	Big-C	✓ / ✗	✓	✗
Big-C	Yarn FIFO Scheduler	✓ / ✗	✓	✗
Tetrisched	Yarn	✓	✗	✓
Firmament	Quincy	✓	✗	✗

Table 5.9 – Datacenter schedulers that compare to other schedulers. Some use the same code base, some implement other system's techniques on top of a simulator. -: does not apply. *: no admission control.

5.8.1 HPC and Grid computing

High Performance Computing (HPC) schedulers (e.g., SLURM [104], TORQUE [91]) use centralized scheduling and do not have the same latency requirements. The jobs they schedule are usually compute-intensive and often long running. These jobs come with several constraints as they are tightly coupled in nature, requiring periodic message passing and barriers. These schedulers run at most a hundred of concurrent jobs. Recent work [3] compares the Google trace to one private and two public HPC traces, concluding that jobs are in general bigger, longer and fewer. Grid computing focused more on sharing geographically distributed resources (while managed separately) among virtual organizations [36].

5.8.2 Operating systems scheduling

Although this thesis focuses on scheduling for datacenters, where resources are in different machines, there is an ongoing research effort from the operating systems community to improve scheduling as well because scheduling threads on multicore machines is hard. Most of the recent publications focuses on evaluating the resource wastage. For instance [70] showed that the Linux scheduler violates a work-conserving invariant because it allows runnable threads to be stuck in runqueues for seconds while there are idle cores in the system. Other research performs comparisons among different operating systems schedulers. Bouron et. al. [14] compare the scheduler from FreeBSD with the Linux CFS scheduler. The latency of the

Linux scheduler versus a custom implementation of a multilevel feedback queue is compared in [98]. Antother example is the studies of comparing the overhead of CFS against BFS in [46].

Doing good scheduling at the node level is complementary to datacenter schedulers, but as nodes get more and more resources like CPU and memory, node level schedulers become more relevant to datacenters as well.

6 Conclusions and Future Work

"Every new beginning comes from other beginnings end"
— Seneca the Younger

In this dissertation we provide techniques to tackle the four main scheduling problems explained in Chapter 1, namely: high-quality placement vs scalable scheduling, head-of-line blocking, lack of job-awareness, and dependency on estimates. The remainder of this Chapter explains how Chapters 2, 3, and 4 answer the hypotheses made in Chapter 1 in Section 6.1. We present lessons learned in Section 6.2 and finally, we discuss about future directions for datacenter schedulers in Section 6.3.

6.1 Contributions

This thesis provides with the following contributions:

First, we designed, developed and evaluated a novel hybrid scheduler that reconciles the trade-off between high-quality placement and low scheduling latency.

In **Chapter 2**, we showed that datacenter schedulers can get the best of both worlds with a hybrid design (Hawk [29]) by scheduling long jobs in a centralized scheduler and short jobs distributedly. To avoid head-of-line blocking, Hawk [29] reserves a small part of the cluster resources to execute only short jobs (the rest of the cluster can execute both short and long). Additionally, Hawk employs randomized work stealing: a free node `steals` short tasks that are behind long tasks.

Second, we devised, implemented and evaluated two novel techniques that provide job-awareness while avoiding head-of-line blocking for a hybrid scheduler. In **Chapter 3**, we presented Eagle [27]: a new hybrid datacenter scheduler for data-parallel programs. Eagle

avoids completely the head-of-line blocking by making short jobs avoid nodes with long jobs altogether. Similarly to Hawk, Eagle maintains a reserved portion of the cluster to avoid long jobs clogging all of the resources. Sticky probes ensure that a job, once started, has the chance to progress to completion. In this way, Eagle provides job-awareness.

Third, we propose a new design and scheduling policy for an estimate-oblivious scheduler that can achieve good scheduling latency and competitive job runtimes. In **Chapter 4**, we present Kairos [28], a novel datacenter 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. Kairos has been implemented in the widely used Yarn [100] framework and evaluated against state-of-the-art preemptive and non-preemtive estimation-based schedulers.

These chapters collectively serve to prove the hypotheses stated in Chapter 1.

In particular, **thesis statement 1 (Section 1.3)** questions if a datacenter scheduler can achieve high-quality placement and good scheduling latency without compromising one or the other. The hybrid design we employed in Hawk advocates for scheduling long jobs centrally so that jobs that occupy most of the resources get high-quality placement. Short jobs, which make for most of the scheduling load, are scheduled in a distributed way which results in good scheduling latency. We implemented this design in Hawk and proved that such a scheduler can indeed achieve better job runtimes for both short and long jobs, especially in the case of a highly loaded cluster.

The second part of **thesis statement 1 (Section 1.3)** poses the hypothesis that a scheduler that gets the best of both worlds can, in addition, provide job-awareness and avoid head-of-line blocking completely. In Eagle, we showed that the two novel techniques presented indeed avoid completely any short task from being scheduled behind a long task and achieve better job runtimes because jobs run to completion. This results in improved job runtimes for short jobs, especially under high load.

Thesis statement 2 (Section 1.3) propounds that a datacenter scheduler can achieve good scheduling latency and job runtimes without depending on job runtime estimation. Kairos [28] proves that, by using a light-weight load balancing scheduling and a preemptive node sharing algorithm, a datacenter scheduler can dispense with using runtime estimates altogether. We

showed that this design achieves better or equal job runtimes compared to state-of-the-art schedulers.

6.2 Lessons learned

Through the process of building this thesis we learned some lessons that influenced our scheduling designs and techniques.

- Scheduling in datacenters should take into account primarily the length of the jobs. This holds even when long jobs have high priority. Take the case that a long job has high priority, the slowdown that it experiences by letting a short job bypass it is negligible. The converse is not true: a short job that might have a lower priority will experience a great slowdown by letting a long job bypass it. An analysis of the workload characteristics in Chapter 2 shows the heavy-tailed distribution of the workloads. Also, Hawk served as a proof-of-concept system for scheduling jobs according to their length. Later, a scheduling model that takes into account different job runtimes for reordering/prioritizing jobs has been adopted by other systems like Yaq [81], Big-C [19], Eagle (Chapter 3), and Medea [38]. The slowdown is a good metric to measure this, like in Chapter 4.

- A datacenter scheduler should take into account the nature of the jobs rather than schedule all of them using the same logic. Jobs in a datacenter are different in nature and consequently have different needs and priorities. This is corroborated by other systems in the literature [88, 51] that advocate for schedulers with different logics sharing the cluster resources. On the other hand, priorities are not enough to account for differences among jobs, as discussed in Chapter 5. We think this need will persist in the future because the diversity of jobs will continue to increase, for instance with streaming and machine learning jobs.

- Although queueing theory models are not directly applicable to datacenter scheduling, they can serve as inspiration to provide mechanisms and techniques that can be applied in practice. As an example, the two techniques presented in Chapter 3 are inspired in the Pollaczek-Khinchine formula [64] and Little's law [68]. The design of the algorithm for node scheduling and load balancing in Chapter 4 is also inspired by the Foreground-Background discipline [75].

- Finding the perfect scheduling solution is not trivial, even for an offline scheduling algorithm. In a highly loaded cluster, giving always priority to certain type of jobs (for example short) will come at the expense of hurting other jobs (long). At that point, it becomes a matter of which job gets the priority to execute first. We see this effect in Eagle's evaluation. Similarly, when Kairos is compared to Eagle, we see that at high load the long jobs lower percentiles get better while the tail gets worse. This is due to the difference on the scheduling policy used: in Kairos the shorter jobs will get priority while in Eagle the Least Work Left policy is used.

- As consequence of the previous point, it is imperative for a datacenter scheduler to report more percentiles than just the mean. Depending on the runtimes distribution, a better visualization of the results could include reporting some percentiles rather than a full CDF. This, together to the slowdown metric, help to understand the benefits of a given scheduling solution.
- In chapter 4 we see that preemption can be used effectively in a datacenter to help take advantage of the progress of started jobs and of giving immediate priority to other jobs. Notably, preemption can be implemented in a low latency manner [19].
- Techniques that kill, preempt or reorder jobs in a queue according to their size need to be coupled with an anti-starvation mechanism. We used anti-starvation mechanisms for both Eagle and Kairos. Related work (Chapter 5) also uses anti-starvation mechanisms. It is important to not make this mechanism too aggressive and instead, find a sweet spot.
- Late binding is a powerful technique that can help to achieve better scheduling quality for stateless schedulers. In Sparrow [79] it was used only to compensate for the lack of visibility on the cluster that distributed schedulers have. But in fact, late binding is a more powerful technique because it does not rely on estimates and delays the *binding* (a task to a resource) decision to the latest moment.
- A datacenter scheduler should provide mechanisms to fix *bad* scheduling due to unforeseen events. We observe diverse correction mechanisms used in the literature in Chapter 5. We argue that it is important to minimize, the effect that these events have in the original intended scheduling.

6.3 Future Work

We discuss in this Section the limitations of our schedulers and interesting future research to extend our work.

6.3.1 Extending Hawk

The centralized scheduler component in Hawk schedules long jobs in a Least Work Left (LWL) fashion, but it could, in fact, be plugged with a more complicated scheduling policy. For instance: including constraints, taking into account hardware heterogeneity, and other relevant placement requirements that long jobs need.

The stealing mechanism in Hawk could be enhanced by means of hinting which task became the scheduler induced straggler, for example, through a gossiping protocol among nodes. It is also worth exploring probing systems further. For example, an interesting question is: how would a small number of distributed schedulers, that schedule more than just one job, take advantage of sharing the jobs probes? What would be the right number of schedulers to have?

At one extreme we have Sparrow's technique that envisions having as many as one scheduler per job. The other extreme is having one centralized scheduler sharing the probes from all of the jobs. The number of probes sent also makes a difference: should all the schedulers share all of the cluster? how to load balance such a setting?

Accommodating long and short jobs in the same node and sharing resources among them while providing isolation is also an interesting path to explore, because Hawk only schedules using slots.

6.3.2 Porting Eagle

Although we implemented Eagle's techniques in a hybrid scheduler, it would be interesting to see how these techniques fare in centralized or distributed designs. For a purely distributed design, the bitmap sent to schedulers to schedule short tasks (so as to avoid scheduling them behind a long one) could be gathered similar to how the wait-time matrix is for Apollo [16] (however, it would be much more lightweight and less prone to conflicts). For a centralized design, it is straightforward to implement and to make short jobs avoid long jobs.

The second technique based on probing is designed for distributed schedulers. It would be interesting to test it in centralized schedulers. The intuition is that late binding and running a job to completion will not only provide job-awareness, but also alleviate the centralized scheduler load by using a simple yet powerful scheduling technique.

6.3.3 Extending Kairos

Scheduling in Kairos has the disadvantage that it does not provide any guarantees for jobs. Experimenting with different techniques to combine this and other important constraints jobs might have would be beneficial and important. Given that prioritizing always shorter jobs causes a detriment in longer jobs runtime, better starvation mechanisms could be studied as well. This would benefit any system based on reordering or scheduling short jobs before long ones. Another interesting path to explore is to extend the algorithms in Kairos to include other resources like memory.

6.4 Discussion

This thesis presents three systems that further improve the state-of-the-art in datacenter scheduling. Hawk (Chapter 2) introduces a new hybrid scheduling design by combining a centralized scheduler and many distributed schedulers for short jobs. Eagle (Chapter 3) improves short job runtimes by avoiding the head-of-line blocking and being job-aware. Kairos (Chapte 4) dispenses with runtime estimates while still achieving good job runtimes. The solutions propose three flavors of runtime estimation dependency: (i) Eagle depends on runtime estimates at the task-level. (ii) Hawk needs to estimate if a job is under or over a

threshold to classify it as short or long. (iii) Kairos does not depend on any runtime estimate.

Bibliography

[1] O. A. Abdul-Rahman and K. Aida. Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In *Proc. of the IEEE 6th International Conference on Cloud Computing Technology and Science*, CloudCom'14, pages 272–277. IEEE, 2014.

[2] 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 *Proc. of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 469–482. USENIX Association, 2017.

[3] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben. Bigger, longer, fewer: what do cluster jobs look like outside Google? Technical report, Technical Report CMU-PDL-17-104, Carnegie Mellon Univedrsity Parallel Data Laboratory, 2017.

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

[5] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20. USENIX Association, 2012.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 265–278. USENIX Association, 2010.

[7] Apache Foundation. Apache Hadoop capacity scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[8] Apache foundation. Apache Hadoop fair scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[9] Apache foundation. Apache Hadoop gridmix. http://hadoop.apache.org/docs/r1.2.1/gridmix.html.

[10] Apache Foundation. Apache Hadoop project. http://hadooop.apache.org.

[11] Apache Foundation. Apache Spark project. http://spark.apache.org, 2016.

[12] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proc. of the 4th annual Symposium on Cloud Computing*, SOCC'13, page 20. ACM, 2013.

[13] N. Avrahami and Y. Azar. Minimizing total flow time and total completion time with immediate dispatching. In *Proc. of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pages 11–18. ACM, 2003.

[14] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena. The battle of the schedulers: FreeBSD ULE vs. Linux CFS. Technical report, 2018.

[15] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, SC'03, page 25. ACM, 2003.

[16] 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 *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 285–300. USENIX Association, 2014.

[17] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[18] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a SQL implementation on the mapreduce framework. *Proc. of the VLDB Endowment*, 4(12):1318–1327, 2011.

[19] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proc. of the 2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 251–263. USENIX Association, 2017.

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

[21] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. of the IEEE 19th Annual International Symposium*

on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '11, pages 390–399. IEEE Computer Society, 2011.

[22] E. Coppa and I. Finocchi. On data skewness, stragglers, and mapreduce progress indicators. In *Proc. of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 139–152. ACM, 2015.

[23] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Pearson Education, 2005.

[24] D. H. Craft. Resource management in a decentralized system. In *ACM SIGOPS Operating Systems Review*, volume 17, pages 11–19. ACM, 1983.

[25] 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 *Proc. of the ACM Symposium on Cloud Computing*, SoCC '14, pages 2:1–2:14. ACM, 2014.

[26] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[27] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proc. of the 7th ACM Symposium on Cloud Computing*, SoCC'16, pages 497–509. ACM, 2016.

[28] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. Technical report, Ecole Polytechnique Federale de Lausanne, Switzerland, 2018.

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

[30] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.

[31] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144. ACM, 2014.

[32] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proc. of the 6th ACM Symposium on Cloud Computing*, SoCC, pages 97–110. ACM, 2015.

[33] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 431–442. ACM, 2014.

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

[35] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proc. of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112. ACM, 2012.

[36] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[37] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

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

[39] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 29. WH Freeman New York, 2002.

[40] B. Ghit and D. H. J. Epema. Tyrex: Size-based resource allocation in mapreduce frameworks. In *Proc. of the IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, series=CCGrid'16, pages = 11–20, year = 2016.*

[41] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336. USENIX Association, 2011.

[42] I. Gog, M. Schwarzkopf, A. Gleave, R. M. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16. USENIX Association, 2016.

[43] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, 2014.

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

[45] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 81–97. USENIX Association, 2016.

[46] T. Groves, J. Knockel, and E. Schulte. BFS vs. CFS scheduler comparison. https://www.cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf, 2009.

[47] J. Hamilton. Cost of power in large-scale data centers. https://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers/, 2008.

[48] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Cambridge University Press, 1st edition, 2013.

[49] M. Harchol-Balter, C. Li, T. Osogami, A. Scheller-Wolf, and M. S. Squillante. Analysis of task assignment with cycle stealing under central queue. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, ICDCS'03, pages 628–637. IEEE, 2003.

[50] M. Harchol-Balter, A. Scheller-Wolf, and A. R. Young. Surprising results on task assignment in server farms with high-variability workloads. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):287–298, 2009.

[51] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308. USENIX Association, 2011.

[52] Z. Hu, B. Li, Z. Qin, and R. S. M. Goh. Job scheduling without prior information in big data processing systems. In *In Proc. of the IEEE 37th International Conference on Distributed Computing Systems*, ICDCS'17, pages 572–582. IEEE, 2017.

[53] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proc. of the IEEE 26th International Conference on Data Engineering Workshops*, ICDEW'10, pages 41–51. IEEE, 2010.

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

[55] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *Proc. of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 519–532, Boston, MA, 2018. USENIX Association.

[56] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *Proc. of the 2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 97–109. USENIX Association, 2017.

[57] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of the ACM European Conference on Computer Systems*, EuroSys'07. ACM, March 2007.

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

[59] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. *ACM SIGCOMM Computer Communication Review*, 45(4):407–420, 2015.

[60] 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 *Proc. of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, pages 485–497. USENIX Association, 2015.

[61] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[62] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proc. of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103. IEEE Computer Society, 2010.

[63] J. E. Kelley Jr and M. R. Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173. ACM, 1959.

[64] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[65] C. Kozyrakis. Resource efficient computing for warehouse-scale datacenters. In *In Proc. of Design, Automation & Test in Europe Conference & Exhibition*, DATE'13, pages 1351–1356. IEEE, 2013.

[66] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[67] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. *Journal of Computer and System Sciences*, 73(6):875–891, 2007.

[68] J. D. C. Little. A proof for the queuing formula: L=λw. *Operations Research*, 9(3), 1961.

[69] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[70] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The linux scheduler: a decade of wasted cores. In *Proc. of the 11th European Conference on Computer Systems*, EuroSys'16, page 1. ACM, 2016.

[71] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[72] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[73] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[74] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 439–455. ACM, 2013.

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

[76] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[77] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proc. of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 184–200. ACM, 2017.

[78] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *Proc. of the 14th Workshop on Hot Topics in Operating Systems*, volume 13 of *HotOS'13*, pages 14–14, 2013.

[79] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84. ACM, 2013.

[80] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proc. of the 13th ACM European Conference on Computer Systems*, EuroSys'18, page 2. ACM, 2018.

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

[82] 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 *Proc. of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13. ACM, 2012.

[83] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. In *Proc. VLDB 2013*, 2013.

[84] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proc. of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 379–392. ACM, 2015.

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

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

[87] M. Schwarzkopf. Operating system support for warehouse-scale computing. *PhD. University of Cambridge*, 2015.

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

[89] M. Sedaghat, E. Wadbro, J. Wilkes, S. de Luna, O. Seleznjev, and E. Elmroth. Diehard: Reliable scheduling to survive correlated failures in cloud data centers. In *In Proc. of the IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing*, CCGrid'16, pages 52–59. IEEE, 2016.

[90] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proc. of the 2nd ACM Symposium on Cloud Computing*, SoCC'11, page 3. ACM, 2011.

[91] G. Staples. Torque resource manager. In *Proc. of the 2006 ACM/IEEE conference on Supercomputing*, SC'06, page 8. ACM, 2006.

[92] J. Stone. Tachyon Parallel/MultiprocessorRay tracing system. http://jedi.ks.uiuc.edu/~johns/raytracer/.

[93] V. Sundarapandian. *Probability, statistics and queuing theory*. PHI Learning Pvt. Ltd., 2009.

[94] A. S. Tanenbaum and M. Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[95] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*, volume 2. Prentice-Hall, 1987.

[96] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: a software-defined storage architecture. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196. ACM, 2013.

[97] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. of the VLDB Endowment*, 2(2):1626–1629, 2009.

[98] L. A. Torrey, J. Coleman, and B. P. Miller. A comparison of interactivity in the linux 2.6 scheduler and an MLFQ scheduler. *Software: Practice and Experience*, 37(4):347–364, 2007.

[99] 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 *Proc. of the 11th European Conference on Computer Systems*, EuroSys'16, page 35. ACM, 2016.

[100] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16. ACM, 2013.

[101] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In *Proc. of the 2014 IEEE International Conference on Cluster Computing*, CLUSTER'14, pages 48–56. IEEE, 2014.

[102] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proc. of the 10th ACM European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17. ACM, 2015.

[103] J. Wilkes. More Google cluster data, 2011. http://googleresearch.blogspot.ch/2011/11/more-google-cluster-data.html.

[104] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[105] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical report, Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, 2009.

[106] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278. ACM, 2010.

[107] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[108] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42. USENIX Association, 2008.

[109] 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 *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 755–770. USENIX Association, 2016.

Pamela Delgado

Curriculum Vitae

"The size of your dreams must always exceed your current capacity to achieve them"
— Ellen Johnson Sirleaf

Education

2012 - 2018 **Ph.D. Student and Research Assistant**, *École Polytechnique Fédérale de Lausanne (EPFL)*, Lausanne - Switzerland.
Computer Science. Distributed and scalable scheduling in datacenters

2010 - 2012 **Master of Science. Computer Science**, *École Polytechnique Fédérale de Lausanne (EPFL)*, Lausanne - Switzerland, *GPA – 5.36/6*.
Computer Science. Specialization: Foundations of Software

2002 - 2008 **Bachelor: Systems Engineering**, *Universidad Católica Boliviana San Pablo*, Cochabamba - Bolivia.
Graduated with honours

Mar - Jun 2006 **Exchange student**, *Pontificia Universidad Católica de Chile*, Santiago - Chile.

Special Achievements and Awards

2013 - 2018 Grant from Microsoft Research. Swiss Joint Research Center. Project: "Towards Resource Efficient Datacenters".

2013 Google Anita Borg Memorial Scholarship EMEA. Google Inc.

Jul 2012 Master's Final Project maximum score. École Polytechnique Fédérale de Lausanne

2010 - 2012 Swiss Federal Scholarship for Foreign Students. Swiss Federal Commission for Scholarships for Foreign Students.

Apr 2008 Bachelor's Final Project maximum score. Universidad Católica Boliviana San Pablo.

Mar 2006 Academic Excellence Scholarship for Latin American Students. Pontificia Universidad Católica de Chile.

Feb 2005 Academic Effort Recognition Scholarship. Universidad Católica Boliviana San Pablo.

Publications

Kairos: Preemptive Data Center Scheduler Without Runtime Estimates, Pamela Delgado, Diego Didona, Florin Dinu and Willy Zwaenepoel.
The ACM Symposium on Cloud Computing (SoCC), Carlsbad, CA, USA. 2018.

Job-Aware Scheduling in Eagle: Divide and Stick to Your Probes, Pamela Delgado, Diego Didona, Florin Dinu and Willy Zwaenepoel.
The ACM Symposium on Cloud Computing (SoCC), Santa Clara, CA, USA. 2016.

Eagle: A Better Hybrid Data Center Scheduler, Pamela Delgado, Diego Didona, Florin Dinu and Willy Zwaenepoel.
Poster at 11th European Conference on Computer Systems (EuroSys), London, UK, 2016.

EPFL IC LABOS BC 158 Station 14 – CH-1015 Switzerland
☎ +41 216938133 • ✉ pamela.delgado@epfl.ch

Hawk: Hybrid Datacenter Scheduling, Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec and Willy Zwaenepoel.
The USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, 2015.

Distal: A Framework for Implementing Fault-tolerant Distributed Algorithm, Martin Biely, Pamela Delgado, Zarko Milosevic and André Schiper.
43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, 2013.

Efficiently Scheduling Data-Parallel Computations on Very Large Clusters, Pamela Delgado, Khaled Elmeleegy, Anne-Marie Kermarrec and Willy Zwaenepoel.
Poster at 8th European Conference on Computer Systems (EuroSys), Prague, Czech Republic, 2013.

Projects in Research Laboratories

Summer 2013 ***Research intern in Systems and Networking***, Simulation Research for Fabric Computers, Microsoft Research Cambridge, Antony Rowstron.

Autumn 2012 ***Large-scale task scheduling in Hadoop***, Operating Systems Laboratory (LABOS), EPFL, Prof. Willy Zwaenepoel.

Spring 2012 ***Domain Specific Language for Distributed Algorithms in Scala***, *Master project*, Laboratoire de Systemes Reparties (LSR), EPFL, Prof. André Schiper.

Autumn 2011 ***Invariant Verifier for Parallel Programs***, *Semester project*, Programming Methods Laboratory (LAMP), EPFL, Prof. Martin Odersky.

Spring 2011 ***STAMP in Java: Benchmark for Software Transactional Memory***, *Semester project*, Distributed Programming Laboratory (LPD), EPFL, Prof. Rachid Guerraoui.

Teaching Experience

Autumn 2017 **Analysis I**, *Teaching Assistant*, EPFL, Lausanne, Switzerland.

Spring 2017 **Operating Systems Implementation**, *Teaching Assistant*, EPFL, Lausanne, Switzerland.

2012 - 2017 **Operating Systems**, *Teaching Assistant*, EPFL, Lausanne, Switzerland.

Autumn 2016 **Programming I**, *Teaching Assistant*, EPFL, Lausanne, Switzerland.

Autumn 2015 **Information, Communication, Computation**, *Teaching Assistant*, EPFL, Lausanne, Switzerland.

Spring 2005 **Operating Systems**, *Teaching Assistant*, Universidad Católica Boliviana San Pablo, Cochabamba, Bolivia.

Invited Talks

2018 **Microsoft Swiss JRC workshop 2018**, *Towards Resource Efficient Datacenters - Closing summary*, EPFL Lausanne, Switzerland.

2017 **Ecocloud Annual Event**, *Job-aware Scheduling in Eagle: Divide and Stick to Your Probes*, Lausanne, Switzerland.

2017 **Microsoft Swiss JRC workshop 2017**, *Towards Resource Efficient Datacenters - Eagle*, MSR Cambridge, United Kingdom.

2017 **Eurosys Shadow PC meeting**, *Job-aware Scheduling in Eagle: Divide and Stick to Your Probes*, Google Zurich, Switzerland.

2017 **Annual EPFL INRIA Workshop 2017**, *Hawk: Hybrid Datacenter Scheduling*, Rennes, France.

2016 **ACM Symposium on Cloud Computing (SoCC)**, *Job-aware Scheduling in Eagle: Divide and Stick to Your Probes*, Santa Clara, US.

EPFL IC LABOS BC 158 Station 14 – CH-1015 Switzerland
☎ +41 216938133 • ✉ pamela.delgado@epfl.ch

2016 **Microsoft Systems and Networking Group Meeting**, *Job-aware Scheduling in Eagle: Divide and Stick to Your Probes*, Microsoft Research Lab Redmond, US.

2016 **Nutanix**, *Job-aware Scheduling in Eagle: Divide and Stick to Your Probes*, Santa Clara, US.

2016 **Microsoft Swiss JRC workshop 2016**, *Towards Resource Efficient Datacenters - Hawk*, ETHZ Zurich, Switzerland.

2015 **Usenix Annual Technical Conference (ATC)**, *Hawk: Hybrid Datacenter Scheduling*, Santa Clara, US.

2014 **Presentation to School Students (Présentation aux gymnasiens)**, *Big Data: au-delà des systèmes distribués*, EPFL Lausanne - Switzerland.

Community Service

Reviewed papers, *IEEE Transactions on Parallel and Distributed Systems journal (TPDS) 2018, IEEE Transactions on Automation Science and Engineering (T-ASE) 2016, Eurosys 2018, Eurosys 2017 (shadow PC).*

Professional Experience

Summer 2011 **Summer Intern**, *KeyLemon SA*, Face recognition application for Android, Martigny, Switzerland.

2009 - 2010 **Software engineer**, *Outsourcing for Enable Consultants*, Recess web application for Canadian schools, Toronto, Canada.

2008 - 2009 **Software engineer**, *PirAMide Informatik SRL*, Medical Imaging and digital dictation modules development, Cochabamba, Bolivia.
In partnership with Medspazio, Geneva Switzerland

2006 - 2008 **Software engineer**, *PirAMide Informatik SRL*, EJB Automatic Migration Tool, Information systems development, Cochabamba, Bolivia.

Miscellaneous

July - Nov 2005 **Edition Assistant**, *"Acta Nova" Scientific Magazine Vol. 3 N 1 December 2005.*
Universidad Católica Boliviana San Pablo

Languages

English, Fluent.

French, Advanced.

Spanish, Fluent.

Italian, Intermediate.

German, Elementary.

EPFL IC LABOS BC 158 Station 14 – CH-1015 Switzerland
☎ +41 216938133 • ✉ pamela.delgado@epfl.ch