Improving Performance and Fairness for Big Data Job Schedulers in Large-Scale Datacenters

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

in partial fulfillment
of the requirements for the degree

Doctor of Philosophy

by

Wei Zhou

August 2019

APPROVAL SHEET

This Dissertation
is submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Author Signature: Wei Zhou

This Dissertation has been read and approved by the examining committee:

Advisor: K. Preston White

Committee Member: William T. Scherer

Committee Member: Cody Harrison Fleming

Committee Member: Samira Khan

Committee Member: Hongfeng Yu

Committee Member: ____________

Accepted for the School of Engineering and Applied Science:

[illegible]

Craig H. Benson, School of Engineering and Applied Science

August 2019

Improving Performance and Fairness for Big Data Job Schedulers in Large-Scale Datacenters

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Systems Engineering)

by

Wei Zhou

August 2019

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Systems Engineering)

Wei Zhou

Wei Zhou

This dissertation has been read and approved by the Examining Committee:

K. Preston White

Advisor

William T. Scherer

Committee Chair

Cody Harrison Fleming

Committee Member

Samira Khan

Committee Member

Hongfeng Yu

Committee Member

Accepted for the School of Engineering and Applied Science:

Craig H. Benson

Craig H. Benson, Dean, School of Engineering and Applied Science

August 2019

Abstract

It is a critical challenge to design a highly-efficient, high-performance, and fair big data job scheduler, especially in large-scale datacenters consisting of heterogeneous servers under intensive, complex, and diverse workloads. Hybrid job schedulers, which combine a centralized job scheduler and multiple distributed job schedulers together, have been considered as a promising alternative to conventional centralized job schedulers deployed in enterprise datacenters. However, our literature survey and experimental study show that, (1) the state-of-the-art hybrid job schedulers fail to ensure low latency for latency-sensitive short jobs; and (2) the state-of-the-art fair job schedulers for constrained jobs fail to ensure fair sharing in heterogenous-server environments.

To this end, we first address the high-latency performance issue of short jobs due to the head-of-line blocking and straggler tasks for hybrid job schedulers. We propose **Dice**, a new general performance optimization framework for hybrid job schedulers to alleviate the high job latency problem of short jobs. Dice is composed of two simple yet effective techniques: *Elastic Sizing* and *Opportunistic Preemption*. Both Elastic Sizing and Opportunistic Preemption keep track of the task waiting times of short jobs. When the mean task waiting time of short jobs is high, Elastic Sizing dynamically and adaptively increases the short partition size to prioritize short jobs over long jobs. On the other hand, Opportunistic Preemption preempts resources from long tasks running in the general partition on demand, so as to mitigate the head-of-line blocking problem of short jobs. We then propose **Eirene**, another new general performance optimization framework for hybrid job schedulers to improve job latency performance of short jobs via two schemes tightly coupled with the general architecture of hybrid job schedulers. Eirene consists of two schemes. *Coordinated Cold Data Migration* leverages high task waiting time of short jobs under heavily-loaded periods and migrates cold data from disks to local memory for the initial phase of reading input so as to shorten task runtime and queueing time. On the other hand, *Scheduler-Aware Task Cloning* exploits spare computing resources under lightly-loaded periods and performs proactive task cloning for short jobs to mitigate the straggler problem.

We then address the unfair scheduling of jobs with placement constraints in heterogeneous environments. We propose **Eunomia**, a performance-variation-aware fair job scheduler with placement constraints for heterogeneous datacenters. Eunomia introduces *progress share fairness*, which is meant to equalize the progress share of jobs as much as possible. Progress share of a job is defined as the ratio between the accumulated progress of scheduled tasks of a job, and the maximum accumulated progress of tasks that can run in the cluster if placement constraints are removed.

Keywords: big data analytics, resource management, hybrid job scheduler, fair job scheduler.

To my beloved parents

Acknowledgments

This dissertation is an exciting yet challenging milestone of my Ph.D. journey. Throughout my whole Ph.D. study, I have received lots of support and assistance from many people.

First of all, I am extremely grateful to my parents. Although I am their only child, they never restricted me to accompany them and have continued to support me to go abroad to study as well as pursue my dreams. I knew that they are always there and wanted the best for me. Without their warm love, continued encouragement, and endless support and trust, I would not be able to achieve my current status. I feel so lucky to be their daughter and hope I have been and will be always their pride. Thanks for their guiding me that my journey in life was to learn, to be strong and happy, and to know and understand myself. Only after that, I could know and understand others.

My greatest appreciation goes to my honorable advisor Prof. Preston White. He is not only a successful professor with great achievements but also an enthusiastic friend who has shared his extensive knowledge with me both from his academic and industry experiences. Thanks for his confidence in me and strong support throughout my Ph.D. life.

I also would like to express the deepest thanks to my dissertation committee members, Prof. William T Scherer, Prof. Cody Harrison Fleming, and Prof. Samira Khan for their direction, dedication, great support, and insightful advice. In addition, I would like to thank Prof. Hongfeng Yu for his wonderful collaboration in our research publications and being the committee member of my dissertation defense. He supports me greatly and is always willing to help me.

Last but not least, I am heartily grateful to those who have ever helped and supported me during my life so far. I send my best wishes to them and hope all of them will enjoy a happy life as well as fulfill their dreams.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Job Scheduling in Datacenters

Scheduling computing jobs has been an old yet important research topic since the era of time-sharing mainframe computers in the 1960s. Job schedulers continue to evolve in the domain of high-performance computing (HPC) to adapt to large-scale computer clusters consisting of millions of CPU cores [1–4]. Recently with the emergence of Big Data Analytics, the 3V's of Big Data, Volume, Velocity and Variety, demand job schedulers to meet challenging requirements of big data workloads in the environment of geographically-distributed datacenters.

For big data analytics, the main goal of an effective and efficient big data job scheduler for datacenter-level clusters is an ability to schedule a large number of tasks of different jobs on a massive number of worker nodes in a timely fashion. There are the three main challenges to address for job schedulers to achieve the goal as below:

- **Job Complexity**: Different big data frameworks and applications like Hadoop MapReduce [5], Dremel [6], Impala [7], Storm [8], Spark [9], and Flink [10] result in different types of jobs with different requirements. Batch jobs usually take days or weeks to complete, while short ad-hoc queries or interactive jobs need to be completed in minutes even seconds. Big jobs may be composed of thousands of tasks, while small jobs are composed of only a few tasks. Most analytical jobs need to manipulate the exactly whole datasets, while approximate query jobs can operate on a small portion of datasets to produce results. Some jobs need to use Graphics Processing Units (GPUs) for fast computations while some jobs do not need. Short latency-sensitive jobs usually demand job schedulers to make scheduling decisions in milliseconds, while long batch jobs require job schedulers to effectively utilize resources. Generally, there are two categories of jobs in enterprise and cloud datacenters: production jobs and best-effort jobs. Production jobs are usually business-critical jobs, which need to be completed before the deadline specified in the job submissions. The failure to meet the deadline could result in serious losses of business operations. Best-effort jobs usually have no deadline requirements, and their examples include the test run of a production job with a small dataset, and ad-hoc and exploratory jobs submitted by research and development teams. The mixture of different types of jobs with different requirements together makes job scheduling extremely hard.

- **Server Complexity**: Server complexity stems from hardware complexity and software complexity. On one hand, compute nodes could be heterogeneous, which means that different nodes may have different amounts of CPU cores, memory, number of disks, etc. Heterogeneity has been recognized in modern-day data centers [11,12]. It poses a challenge for job schedulers to track and utilize the resources of heterogeneous nodes. Moreover, hardware is not reliable and may fail unpredictably. Once node failures occur, job schedulers must re-schedule and re-execute the aborted tasks on other healthy nodes. It requires job schedulers to have the ability to reschedule tasks if needed. On the other hand, with multiple generations of nodes deployed in datacenters at different points of time, it is common to see different versions of operating systems and software packages installed across different nodes. The mixture of various nodes with different hardware and software configurations together makes job scheduling extremely hard.

- **Multiple Objectives**: In order to maximize return on investment, the primary objective of job schedulers is to obtain high cluster utilization. Due to this reason, the latest job schedulers tend to support multiplexing various big data analytics frameworks on computer clusters. In addition to resource utilization, scalability performance, and fairness are all common objectives. A scalable job scheduler is able to make effective and efficient scheduling decisions with the increasing scale of nodes and jobs. Keeping scheduling latency short is also an important metric to job schedulers, especially for latency-sensitive short jobs. In the datacenters with production and best-effort mixed jobs, a job scheduler must be able to achieve stringent SLOs (e.g., predefined deadlines) of production jobs whiling minimizing turnaround time for best-effort jobs. Finally, guaranteeing the fairness of job scheduling is critical for the organizations where the computer clusters are shared by different departments. It is a conflicting goal to satisfy multiple objectives simultaneously, and this demands job schedulers to have an ability to strike a good balance among multiple objectives, without violating strict requirements.

1.2 Research Problems

There is no one-size-fits-all job scheduler that can address all the above challenges. Targeting different use scenarios and favoring different key objectives, the current research from industry and academia on job schedulers for big data analytics frameworks is diverse and falls into different many categories in solving the job scheduling problems from different perspectives. For example, in terms of job scheduler architecture, centralized job schedulers usually achieve high cluster utilization [13–15], while distributed job schedulers excel at ensuring low latency of short jobs [16]. Very recently a new architecture of job schedulers, hybrid job scheduler, is proposed [17–20]. Hybrid job schedulers recognize the ubiquitous fact of mixed workloads of long batch jobs and latency-sensitive short jobs, and thus aim to deliver low latency of short jobs while maintaining high cluster utilization. In general, a hybrid job scheduler consists of a centralized scheduler for long jobs and multiple distributed job schedulers for scheduling short jobs in parallel (See Chapter 2 for more detail about big data job schedulers). Although hybrid job schedulers are shown a very promising alternative of centralized job schedulers and distributed job schedulers, latest research and our motivation experiments reveal that the state-of-the-art hybrid job schedulers are still far away from guaranteeing **low latency of short jobs** and **fair shares of resources** due to the below three problems:

- **Head-of-Line Blocking**. Head-of-Line blocking denotes that tasks staying in the task queue on a worker node have to spend much time waiting for execution until the task at the head of the task queue finishes. The impact is amplified in case latency-aware short jobs are behind long jobs since long task waiting time maybe 10 times longer than the task runtime of short jobs. To mitigate this problem, hybrid job schedulers are proposed to reserve a small portion of worker nodes (called "short partition") dedicated for running short tasks and let short jobs and long jobs share the remaining worker nodes (called "general partition") [17–19]. However, head-of-line blocking is still unavoidable and severely affect job latency performance of short jobs due to many reasons. First, workload fluctuation results in sometimes cluster overloading and even the worker nodes in the short partition build up a long queue of short tasks, and task waiting time lengthens task runtime and in turn job latency. Second, in this situation, more small tasks will be executed in general partition and thus likely suffer from head-of-line blocking by long tasks.

- **Straggler Tasks**. As aforementioned, frequent hardware and software failures are unavoidable and unpredictable in production clusters, which may cause some tasks of a job to fall behind others or never complete. Such tasks are usually called "stragglers" or "outliers". If stragglers are not handled in time, the entire job will risk violating its SLO (Service-Level Objective, i.e., predefined deadline). For example, more than 15% straggler tasks for 25% phases are observed in a large cluster for the Bing search engine [21]. As straggler problem is seen widespread, straggler tasks are also considered one major cause to lengthening job completion delay. For one example, job latency was lengthened by stragglers by 29% in Bing clusters [21]. For another example, straggler tasks could take up to 8× longer than the mean task runtime in Hadoop clusters, causing the jobs to be slowed down by 47% on average [22].

- **Unfair Resource Sharing**. Fair job scheduling has been extended to many different use scenarios. For example, DRF (Dominant Resource Fairness) [23] is proposed for allocating multiple resource types. For another example, Choosy [24] is proposed for allocating resources for jobs with various placement constraints. However, the existing fair job schedulers do not take performance variation due to server heterogeneity into considerations and fail to achieve fair shares while server heterogeneity is one salient characteristic of enterprise datacenters.

In this dissertation, we focus on addressing the above problems in the context of hybrid job schedulers. The motivation is that, as hybrid job scheduler is a new architecture of job schedulers and already shows a big potential to replace centralized job schedulers in production environments. The possibly high job latencies of latency-sensitive short jobs and unfair job scheduling will definitely hinder its wide deployment in enterprise datacenters. The goal of our research is to have a way to ensure much better job latency performance of short jobs and better fair shares for hybrid job schedulers.

1.3 Research Contributions

This dissertation makes the five contributions as in the below:

- A novel *latency-aware dynamic and adaptive cluster re-partitioning* scheme, called "Elastic Sizing", is proposed to adaptively adjust the short partition size according to the extent of mean task waiting time of short jobs during heavy and overloaded periods. This scheme is integrated into the centralized scheduler side of hybrid job schedulers;

- A novel *latency-aware on-demand task preemption* scheme, called "Opportunistic Preemption", is proposed to preempt randomly-chosen running long tasks for waiting short tasks on the same worker nodes to mitigate long task waiting time of short tasks during heavy and overloaded periods. This scheme is integrated into the centralized scheduler side of hybrid job schedulers;

- A novel "Coordinated Cold Data Migration" scheme is proposed to migrate singly-read input data from disk to memory on worker nodes, so as to shorten the time of reading input data of map tasks of short jobs by *overlapping data migration time with long task waiting time* during heavy and overloaded periods. This scheme is integrated into both distributed scheduler and worker node sides of hybrid job schedulers;

- A novel "Scheduler-Aware Task Cloning" scheme is proposed to clone every task of short jobs by *exploring and exploiting free compute resources* during idle or lightly-loaded periods, so as to alleviate the straggler problem. This scheme is integrated into the distributed scheduler side of hybrid job schedulers to leverage its batch sampling mechanism;

- A novel *performance-variation-aware* fair share job scheduler, called "Eunomia", is proposed to take performance variation of worker nodes into considerations of fair job schedulers. Eunomia introduces a key metric, called "progress share" and aims to equalize progress share of jobs as much as possible, so as to achieve the same slowdown of jobs from different users due to resource sharing and placement constraints, regardless of performance variation.

1.4 Dissertation Outline

In **Chapter 2**, we first introduce the necessary background of job schedulers in the context of High-Performance Computing (HPC) and Big Data Analytics in large-scale datacenters.

Then we present a literature survey of existing and state-of-the-art job schedulers, categorize and compare them in terms of various aspects.

Chapter 3 presents Dice, a new general performance optimization framework for hybrid job schedulers to alleviate the high job latency problem of short jobs. In Dice, we conduct trace-driven experiments to study the job latency performance behaviors of two representative hybrid job schedulers (Hawk and Eagle), and find that short jobs still encounter long latency issues due to intermittent and bursty nature of workloads. To this end, we propose Dice to address the job latency performance issue at the centralized scheduler side. Dice is composed of two simple yet effective techniques: Elastic Sizing and Opportunistic Preemption. Both Elastic Sizing and Opportunistic Preemption keep track of the task waiting times of short jobs. When the mean task waiting time of short jobs is high, Elastic Sizing dynamically and adaptively increases the short partition size to prioritize short jobs over long jobs. On the other hand, Opportunistic Preemption preempts resources from long tasks running in the general partition on demand, so as to mitigate the "head-of-line" blocking problem of short jobs. We enhance the two schedulers with Dice and evaluate Dice performance improvement in our prototype implementation. Experiment results show that Dice achieves 50.9%, 54.5%, and 43.5% improvement on 50th-percentile (P50), 75th-percentile (P75), and 90th-percentile (P90) job completion delays of short jobs in Hawk respectively, as well as 33.2%, 74.1%, and 85.3% improvement on those in Eagle respectively under the Google trace, at low performance costs to long jobs.

Chapter 4 presents Eirene, another new general performance optimization framework for hybrid job schedulers to improve job latency performance of short jobs via two schemes tightly coupled with the general architecture of hybrid job schedulers. Eirene is integrated into both the distributed scheduler and worker node sides, and consists of two schemes. *Coordinated Cold Data Migration* leverages high task waiting time of short jobs under heavily-loaded periods and migrates cold data from disks to local memory for the initial phase of reading input so as to shorten task runtime and queueing time. On the other hand, *Scheduler-Aware*

Task Cloning exploits spare computing resources under lightly-loaded periods and performs proactive task cloning for short jobs to mitigate the straggler problem. We implement a prototype of Eirene based on Eagle, a state-of-the-art hybrid job scheduler. Experimental results show that, under heavy loads, Eirene is able to improve P50, P75, P90 latency performance of short jobs by up to 39.2%, 79.1%, 81.3% respectively compared with Eagle under the Facebook trace with a cluster of 50000 nodes. Under moderate loads, Eirene can also improve Eagle's P50, P75, P90 latency performance of short jobs by 9.1%, 11.6%, 15.8% respectively under the Google trace with a cluster of 15000 nodes.

Chapter 5 presents Eunomia, a performance-variation-aware fair job scheduler, to address the unfairness issue due to performance variation in heterogeneous clusters. Eunomia introduces a key metric, called "progress share", which is defined as the ratio between the accumulated task progress given the current allocation and the accumulated task progress if the user can monopolize the cluster. Eunomia aims to equalize progress share of jobs as much as possible, so as to achieve the same slowdown of jobs from different users due to resource sharing and placement constraints, regardless of performance variation. Simulation-based evaluation results show that Eunomia is able to deliver better share fairness compared with state-of-the-art schedulers without performance loss.

In **Chapter 6** we conclude the dissertation with a summary of contributions and discuss future research directions.

Chapter 2

Related Work

2.1 Background

Scheduling computing jobs is an old yet important research topic. Researchers in the domain of high-performance computing (HPC) have studied job schedulers for a very long time, and proposed a number of job schedulers including HTCondor [1], Slurm [2], MAUI [3], and OpenLava [4] for scheduling HPC jobs. HPC jobs are usually long batch jobs with intensive communication among nodes. As a result, HPC compute nodes are usually interconnected with 10Gbps Ethernet or Infiniband adapters for low latency and high throughput communication. HPC compute nodes may have disks for scratch space, but datasets of jobs are stored in a dedicated, distributed file system (e.g., Lustre [25], Global Parallel File System (GPFS) [26], etc.) atop a storage cluster. To this end, HPC job schedulers are all tailored and optimized for executing communication-intensive long batch jobs on the clusters consisting of compute nodes with high-speed network interconnections, attached with a dedicated storage cluster.

Research on job schedulers recently has been reignited by the increasingly prevalent adoption of emerging big-data analytics frameworks motivated by the Google File System (GFS) [27] and MapReduce computing model [28]. Different from HPC clusters, the GFS is built on computer clusters consisting of commodity and cheap compute nodes to achieve cost effectiveness, and datasets are replicated and distributed to local storage of compute nodes for high data availability. In Google, the MapReduce computing model was developed to execute data analytics jobs like generating web indexes from crawled web pages. Because the GFS has no dedicated clusters for computing, MapReduce assigns computation tasks of a job to the nodes where datasets that the tasks need to analyze reside. One main assumption in this MapReduce model is that there is little data dependency among tasks, therefore MapReduce can achieve an "embarrassingly" high parallelism [29]. In contrast, for HPC, tasks usually have strong data dependency and much execution time is spent on communication and synchronization among nodes. As a result, job schedulers used for high-performance computing are not suitable for big-data analytics frameworks, and new job scheduling algorithms (including the job scheduler module in MapReduce) have been developed to execute a variety of data analytics jobs accordingly.

Although originally designed for batch jobs only, the success of GFS and MapReduce in Google, inspired Yahoo to develop an open-source clone of GFS and MapReduce, called Hadoop. Hadoop has since been widely accepted and deployed as the de-facto big-data analytics infrastructure in industry to execute many kinds of data analytic jobs where datasets are extremely large, for example, transactions analysis for banks, server log scanning for Internet companies, claims fraud detection for insurance companies, customer call record analysis for retailers, etc. With ever-increasing volume, variety, and velocity of big data, Hadoop-based big data analytics have been extended from the batch processing model like MapReduce to interactive computing like Dremel [6] and Impala [7], streaming computing like Storm [8], iterative computing like Spark [9] and Flink [10], and approximate computing like BlinkDB [30], and so on. As a result, many MapReduce-based or MapReduce-like job scheduler variants have been developed and adapted to different preferences of many kinds of big-data analytics frameworks. For example, job schedulers for streaming computing may favor low latency, while job schedulers for batch processing may seek for high utilization of clusters.

With the further adoption and proliferation of Hadoop, people have recognized issues with Hadoop's job schedulers that severely hinder the wider deployment of Hadoop. The root cause comes from the two component of the original design of job scheduler in Hadoop (Hadoop 1.0): JobTracker and TaskTracker [5]. JobTracker is responsible not only for bookkeeping and allocating and reclaiming resources, but also for scheduling and monitoring all pending/running jobs. Such a holistic model causes JobTracker to become a system bottleneck to high scalability. Considering scheduling hundreds of thousands of jobs in a cluster consisting of tens of thousands of nodes, the centralized job scheduler cannot make scheduling decisions in a timely fashion, causing prohibitively long job queueing time and job completion time.

To resolve these issues, Mesos [14] and YARN [13], which both separate the job scheduling functionality from resource management, were proposed for high scalability. Actually, YARN has since become the key component of Hadoop 2.0. Furthermore, both Mesos and YARN aim to be unified job schedulers that are able to execute a mix of different types of jobs (e.g., batch processing jobs and streaming computing jobs). These two open-source job schedulers have been the most commonly-used job schedulers in datacenters worldwide. Based on these two and other similar job schedulers, researchers from industry and academia propose a large number of job schedulers for better scalability, fairness, and resource utilization.

2.2 Big Data Job Schedulers

Usually, big data analytics jobs are submitted and executed on a number of computer clusters in datacenters. Figure 2.1 illustrates an example of a typical computer cluster in a datacenter. Dozens of servers (or nodes) and one network switch are put together in one rack, and a number of racks are in turn connected to form a computer cluster. Geographically-distributed computer clusters are then connected via WAN (Wide Area Network).

A node usually has one or more multi-core CPUs, a certain amount of memory, and

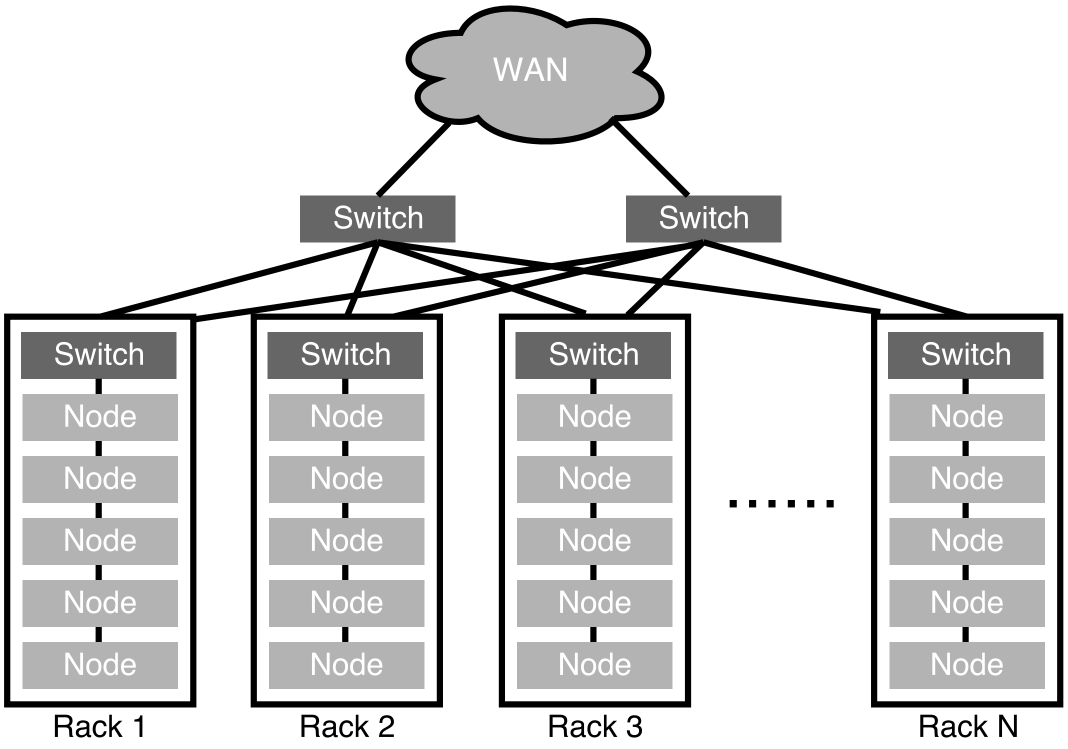


Figure 2.1: An illustrative example of a typical computer cluster in data centers

local storage like hard disk drives (HDDs) or solid-state drives (SSDs). Some nodes may be equipped with GPUs (Graphics Processing Units) as well. For analytics jobs, their datasets are usually distributed (and replicated) into local storage on nodes. Therefore, big data analytics frameworks tend to distribute and execute computing tasks on the nodes where their corresponding datasets reside for leveraging data locality. This is a commonly-used design philosophy for big-data analytics frameworks: "move compute close to data" or "moving computation is cheaper than moving data" [31]. If all the nodes storing the replicas of the needed data are busy, a task will be assigned to a node in the same rack because it only needs to transfer data via the local switch in the rack. This is the so-called "rack locality". Otherwise, a task will be assigned to a node in the other rack, but it takes much more costly network transfer to copy the data before doing computation. Data locality is one important impacting factor every job scheduler should take into considerations.

On top of the computer clusters, as shown in Figure 2.1, job scheduler is meant to

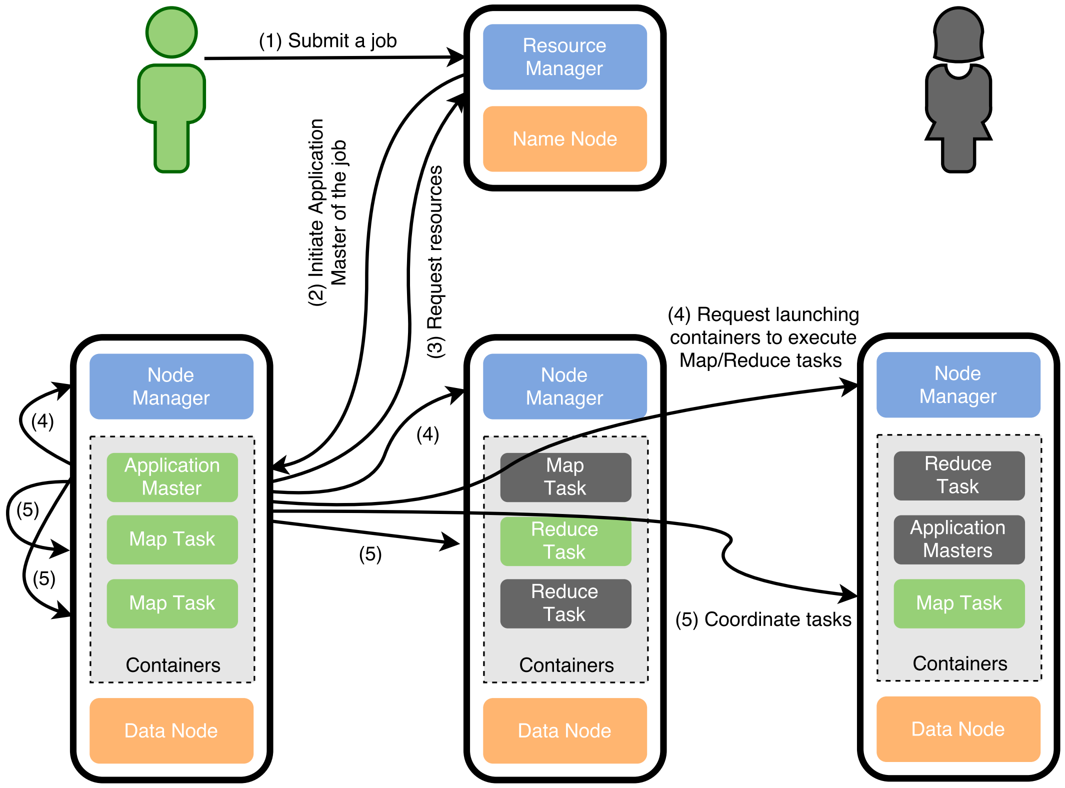


Figure 2.2: An illustrative example of YARN job scheduler

be responsible for accepting or rejecting job submissions from clients, assigning tasks of a job to nodes and monitoring their progress, returning job results to the clients when all the tasks of a job are complete. Let's use the YARN job scheduler running on HDFS as an example shown in Figure 2.2. In HDFS (Hadoop Distributed File System) [31], Name Node manages the metadata of dataset files while data nodes stores replicas of pieces of datasets and service read/write requests from users. For YARN (Yet Another Resource Negotiator) [13], Resource Manager periodically receives the resource utilization information from Node Manager running on each node in the cluster. A job scheduling workflow starts when a client of big data analytics frameworks submits a job to YARN (step 1). Usually, a job defines a number of tasks, where each task includes information about the binary program, dataset location, CPU and memory requirement, etc. Upon receiving the job, YARN then launches an Application Master for this job on one node (step 2). After that, Application Master negotiates the needed resources with Resource Manager (step 3). Then Resource

Manager checks the resource utilization information, and looks for a sufficient amount of idle slots for executing tasks of the job. If the requirement of resources can be met, Resource Manager returns the resource allocation information to Application Master of the job. In step 4, Application Master in turns talks with Node Manager of the nodes where resources are allocated, and copies the binaries to the designated nodes. Node Manager launches Map/Reduce tasks in the containers, and the tasks start to read the data from local storage of nodes or remote storage in other nodes as well as perform actual analytical computations. In the meantime, Application Master monitors the progress of tasks until all the tasks are completed, and finally return the job result to the client (step 5).

There are no "one-size-fits-all" perfect job schedulers. Clusters and nodes could be heterogeneous. Jobs in production environments could be diverse. Users may have different priority preferences and objectives of job scheduling. Each job scheduler must be tailored and optimized for target user scenarios and meeting user requirements. In the following, we will elaborate on the four topics about job schedulers that are the most relevant to our work: (a) job scheduler architectures; (b) head-of-line blocking alleviation; (c) straggler mitigation; and (d) fair job schedulers.

2.2.1 Job Scheduler Architectures

In general, existing big data job schedulers can be categorized into three architectures: centralized job schedulers, distributed job schedulers, and hybrid job schedulers. Table 2.1 compiles a list of influential big data job schedulers categorized by scheduler architecture.

Centralized Job Schedulers

Centralized job schedulers mean there is only one single instance of job scheduler in the cluster. Centralized job schedulers make scheduling decisions for submitted jobs one by one with the global view of cluster resource state and are able to offer strict enforcement of capacity and fairness. Most of the job schedulers shown in Table 2.1 fall into the category of centralized job schedulers. The most representative centralized job scheduler is the

Table 2.1: A summary of influential job schedulers categorized by scheduler architecture

Job Scheduler	**Venue**	**Lead Affiliation**	**Arch.**	**Key Points**
JobTracker [5]	O'Reilly	Apache	Centralized	Hadoop 1.0, clone of Google MapReduce [28]
YARN [13]	SoCC'13	Apache	Centralized	Hadoop 2.0, resource manager/node manager/application master model
Mesos [14]	NSDI'11	UC Berkeley	Centralized	two-level scheduling, resource-offers-based
Kairos [20]	SoCC'18	EPFL	Centralized	centralized scheduler + per-node scheduler, built on YARN
Jockey [32]	EuroSys'12	Microsoft	Centralized	embeds a simulator to estimate job runtime
Bistro [33]	USENIX'15	Facebook	Centralized	tree-based scheduling
Quincy [34]	SOSP'09	Microsoft	Centralized	flow-based scheduling
Firmament [35]	OSDI'16	U. of Cambridge	Centralized	flow-based scheduling
Rayon [36]	SoCC'14	Microsoft	Centralized	reservation-based scheduling, built on YARN
TetriSched [37]	EuroSys'16	CMU	Centralized	reservation-based scheduling, MILP solver, built on YARN
3Sigma [38]	EuroSys'18	CMU	Centralized	reservation-based scheduling, MILP solver, built on YARN
Omega [39]	EuroSys'13	Google	Distributed	state-sharing parallel scheduling
Sparrow [16]	SOSP'13	UC Berkeley	Distributed	probe-based parallel scheduling, no cluster state sharing
Apollo [40]	OSDI'14	Microsoft	Distributed	parallel scheduling, cluster state sharing with loose coordination
Mercury [41]	USENIX'15	Microsoft	Hybrid	centralized scheduler + distributed schedulers, built on YARN
Hawk [17]	USENIX'15	EPFL	Hybrid	centralized scheduler + distributed schedulers, cluster partitioning, built on Spark [9]
Eagle [18]	SoCC'16	EPFL	Hybrid	extends Hawk with job awareness, SRTF algorithm on nodes, built on Spark [9]
Phoenix [19]	ICDCS'17	PSU	Hybrid	extension to Eagle for constrained jobs

JobTracker/TaskTracker framework in Hadoop 1.0 [5]. JobTracker is responsible for not only managing resources, but also making scheduling decisions and monitoring the progress of tasks until all the tasks are completed. It is clear that this holistic design has heavy loads in processing submitted jobs, and is thus not a scalable solution. To address this problem, YARN (Yet Another Resource Negotiator) [13], still a centralized scheduler, delegates the task of job tracking and monitoring to Application Manager that is instantiated for every job after scheduling, and then is able to dedicate itself on resource management and job scheduling for better scalability, as shown in Figure 2.2. In contrast, Mesos takes a different approach for better scalability. As a two-level scheduler, Mesos [14] distributes "resource offers" to scheduler frameworks to guarantee no conflicts of resource allocations. By doing so, Mesos can concentrate on resource allocation and negotiation, and free itself from heavy loads of job scheduling and monitoring. However, Mesos is still a centralized job scheduler. Kairos [20] is also a two-level scheduler, which is composed of one centralized scheduler and one per-node scheduler. For a worker node, the node scheduler is responsible for scheduling and executing tasks received on the node with an approximate implementation of the Least Attained Service (LAS) policy. The LAS policy ensures that the node executes the task receiving the least amount of runtime first, and may preempt the running task if its time quota expires, similar to the process scheduling algorithm in time-sharing operating systems. The centralized scheduler is responsible for distributing tasks among nodes for load balancing. Jockey [32] is a centralized job scheduler aiming to guarantee job latency service-level objective (SLO). Jockey embeds a simulator that simulates job execution and thus it is able to estimate the remaining runtime of a job with different resource allocations at different phases of the job. By doing so, Jockey dynamically adjusts resource allocations so as to achieve high cluster utilization while meeting job SLOs.

The problem of task assignment and resource allocation be abstracted into different forms. One widely-used form is a queuing model. Many centralized job schedulers are queue-based schedulers since it is a natural choice to enqueue jobs in the job submission queue and enqueue tasks in the task waiting queue for a job scheduler. Due to the existence of job queue and task queues, centralized job schedulers implement pluggable scheduling algorithms. A number of well-known scheduling algorithms that have been used in HPC environments, e.g., FCFS (First Come First Served), RR (Round Robin), SJF (Shortest Job First) and LJF (Longest Job First) [42], can be applied to big data job schedulers. FCFS executes the job based on the order of jobs in the queue of job submissions. FCFS is easy to implement, and meaningful when strict ordering of job execution is required. FCFS is supported in YARN of Hadoop 2.0. RR assigns every job in the job queue an equal time slot (called "quantum") to execute in a cyclical round-robin fashion, and the running job will be preempted when its quantum expires. RR is a simple and preemptive scheduling algorithm with a starvation-free guarantee. SJF sorts the jobs in the queue periodically and schedules the job with shortest completion time to execute. SJF is the best way to minimize the average waiting time of jobs, but at the cost of long waiting time for long jobs. In contrast, LJF gives preference to long jobs, which trades job turnaround time for better system utilization.

The problem of task assignment and resource allocation can be also abstracted into a graph model. For example, Bistro [33], Quincy [34], and Firmament [35] are three schedulers that use the graph to model and solve the resource allocation/job scheduling problem. As a tree-based scheduler, Bistro [33] encodes data hosts and their resources into a hierarchical forest of resource trees, and schedules tasks on leaves that can satisfy the hierarchical resource requirements of the tasks from the paths to roots. Quincy [34] and Firmament [35] map the scheduling problem to min-cost flow in a directed graph and encodes the demands like fairness and data locality into edge weights. Quincy solves the min-cost flow problem and computes global matches, which substantially outperform greedy scheduling algorithms. Firmament further outperforms Quincy in terms of scalability while maintaining high placement quality, using multiple min-cost max-flow (MCMF) algorithms with incremental and problem-specific optimizations.

Rayon [36], TetriSched [37] and 3Sigma [38] are three job schedulers that convert the job scheduling problem into the MILP (Mixed Integer Linear Programming) problem in the Operations Research area and solve the problem with solvers. Rayon [36] is the first to introduce reservation-based scheduling, which proposes a declarative reservation definition language (RDL) to capture time-varying resource needs and completion SLAs, converts the planning of current and future resources into a MILP problem as well as develops greedy but scalable heuristics, and proposes adaptive scheduling that dynamically distributes cluster resources to production jobs and best-effort jobs while adapting to the evolving conditions. In tandem with Rayon, TetriSched [37] utilizes the completion SLAs and resource needs of jobs expressed in reservations, makes and optimizes job placement and ordering decisions. Instead of using single-point estimates of historical job runtimes, 3Sigma makes job scheduling decisions based on full distributions of job runtimes.

Distributed Job Schedulers

Distributed job schedulers mean that two or more job schedulers can allocate resources and make scheduling decisions in parallel. Distributed schedulers are a natural way to divide and distribute job scheduling loads for lower job wait time and better scalability and cluster utilization efficiency.

One main challenge to distributed schedulers is how to resolve conflicts of resource allocations when multiple job schedulers allocate the same piece of resources to different jobs. In order to schedule highly parallel short jobs composed of sub-second-runtime tasks, Sparrow [16] enables scheduling from a number of nodes that operate autonomously without consulting and maintaining global resource view. In particular, Sparrow first proposes "Batch Sampling", which probes a number of randomly-selected nodes and places a batch of probes on the nodes with shorter task queues. In order to avoid race conditions where multiple schedulers sample in parallel and contend for the same nodes with short task queues, Sparrow then uses Late Binding to delay task assignments to the nodes until they are ready to run the tasks. In contrast, Omega [39] maintains a master copy of a cluster resource state, and gives each scheduler a private and frequently-update copy. Each scheduler makes scheduling decisions by checking this local copy, and tries to update the master copy in an atomic commit. If the update is not successful due to conflicts, the scheduler can redo the scheduling for the job. Apollo [40] enables individual schedulers to make independent decisions in an optimistic and coordinated manner with a synchronized and global view of cluster resources. Different from Omega, Apollo employs a unique deferred correction mechanism that optimistically defers corrections based on the observations in that resource allocation conflicts caused by independent scheduling decisions are not always harmful, avoiding unnecessary overheads of eager detection and correction mechanisms.

Hybrid Job Schedulers

Hybrid job schedulers combine centralized schedulers and distributed schedulers together and aim to maximize the high cluster utilization advantage of centralized schedulers and low scheduling latency advantage of distributed schedulers.

Mercury [41] is a hybrid scheduler, which augments the centralized scheduler with an auxiliary set of distributed schedulers for strictly enforced capacity and fairness as well as high scalability and efficiency. Mercury defines two kinds of resource allocations: guaranteed container and queuable container for production jobs and best-effort jobs respectively. Its centralized scheduler is used to schedule latency-sensitive tasks that require no queueing delay with guaranteed containers while its distributed schedulers are used to schedule best-effort tasks with queueable containers.

In contrast, Hawk [17], Eagle [18], and Phoenix [19] are hybrid job schedulers for the workloads mixed of long jobs and short jobs. In essence, the general architecture of Hawk, Eagle, and Phoenix divides a cluster into two exclusive partitions: *general partition* and *short partition*, as shown in Figure 2.3. The short partition is dedicated to executing short jobs only while the general partition is used to execute both long and short jobs. The size of the short partition is determined by the resources consumed by short jobs, that is, the total task-seconds of short jobs (the sum of task runtime of tasks for all short jobs) over the total task-seconds of all jobs. Similar to YARN [13], Mesos [14], Borg [43], Kubernetes [15], and other centralized

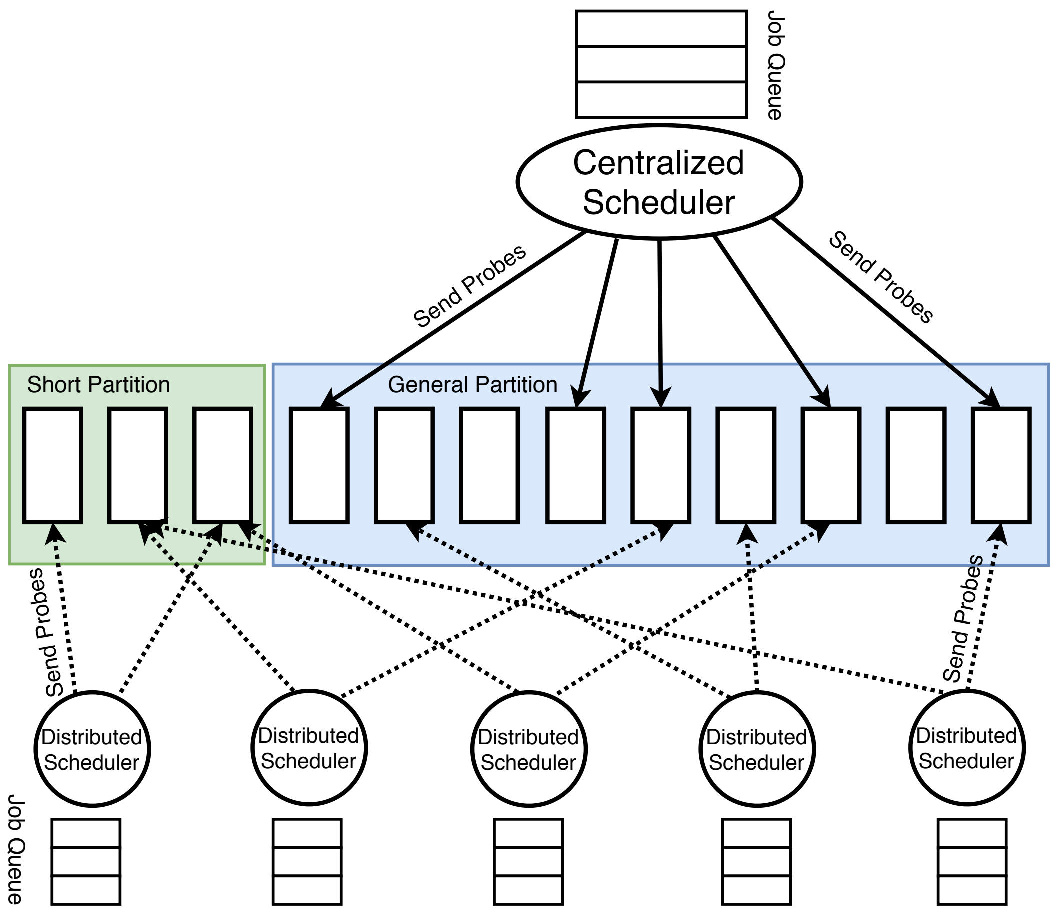


Figure 2.3: A general architecture of hybrid job schedulers

job schedulers, the centralized scheduler in hybrid job schedulers is responsible for enqueueing and placing only long jobs onto worker nodes in the general partition. On the other hand, there are multiple distributed schedulers that can independently schedule only short jobs on any worker nodes in both partitions in parallel. Like Sparrow [16], distributed schedulers employ the "Batch Sampling" scheme to assign and enqueue a batch of task probes for short jobs into probe queues of randomly-chosen worker nodes. When a worker node becomes ready, it fetches one probe from its probe queue and then requests the executable package of one task from a distributed scheduler in charge of the corresponding job. When the worker node receives the task, it launches a container and executes the task program. Such immediate probe placement and late task assignment are also called "Late Binding".

On top of the general architecture of hybrid job schedulers, Hawk [17] introduces the

"Randomized Task Stealing" scheme, where idle worker nodes in the general partition steal task probes of short jobs behind running or waiting long tasks from randomly-chosen busy worker nodes, to compensate occasional poor scheduling decisions by distributed schedulers. Eagle [18] treats a probe as a proxy of the entire job instead of a single task and then proposes the "Sticky Batch Probing (SBP)" scheme. When a task is completed on a worker node, SBP continues to request and execute the remaining tasks of the job until all the tasks are executed. Further, Eagle mitigates the "head-of-line" blocking problem with the "Succinct State Sharing (SSS)" scheme, which shares the information about worker nodes where long jobs are either executing or waiting among distributed schedulers. Phoenix [19] is an extension to the Eagle scheduler to take job placement constraints into considerations.

2.2.2 Head-of-Line Blocking Alleviation

Table 2.2: A summary of representative head-of-line blocking alleviation work

Work	**Cluster (re)partitioning**	**Task (re)distribution**	**Task re-ordering**	**Task Speedup**
Short Partition [16–18]	Yes			
Batch Sampling + Late Binding [16–18]		Yes		
Randomized Task Stealing [17]		Yes		
Succinct State Sharing [18]	Yes			
SRTF [18]			Yes	
LAS-MQ [44]			Yes	
LAS in Node Scheduler [20]			Yes	
Immediate Preemption, Graceful Preemption [45]			Yes	
HotTub [46]				Yes
Elastic Sizing [47]	Yes			
Opportunistic Preemption [47]			Yes	
Coordinated Cold Data Migration [48]				Yes

Understanding the root causes of the head-of-line blocking problem in the context of big data job schedulers is the first step to alleviate this problem. Bursty workloads could build up long task queues sometimes on worker nodes. Tasks waiting behind a long task could suffer from prohibitively long task waiting time. Imbalanced task distribution could cause some nodes to serve much more tasks than other nodes. As a result, actions can be taken to either prevent the head-of-line blocking problem from happening or lower the chances of its occurrences, or minimize its adverse performance impact on latency-sensitive jobs if it happens.

We then can categorize the existing head-of-line blocking alleviation schemes into two groups: architectural designs and optimization schemes. For example, the combination of Batch Sampling and Late Binding probing scheme deployed in Hawk [17], Eagle [18], and Phoenix [19] can be considered as an architectural design to prevent happening of the head-of-line blocking problem since the worker nodes where only the probes return earlier are dispatched tasks by schedulers and execute tasks (the probes that return late are canceled by the schedulers). By the same token, a reserved short partition dedicated for small jobs in hybrid job schedulers [17–19] is also an architectural design to minimize the likelihood that small tasks are blocked by long tasks running on worker nodes.

Regarding optimization schemes, the first main direction is to dynamically repartition the cluster. For example, the Succinct State Sharing (SSS) scheme in Eagle [18] aims to avoid placing the probes of small tasks onto the worker nodes running long tasks, which can be considered a form of dynamic partitioning. In our proposed Dice [47], the "Elastic Sizing" scheme dynamically and adaptively adjusts the short partition size according to the task waiting time of short jobs, so as to improve job-completion-delay performance of short jobs. Elastic Sizing enforces sizing adjustment of the short partition by converting a number of the general-partition nodes into the short-partition nodes throughout consecutive time windows.

The second main direction is to distribute and/or redistribute tasks among worker nodes at runtime to alleviate the head-of-line blocking problem. For example, the randomized task stealing scheme in Hawk [17] aims to move probes of small tasks onto idle worker nodes from busy worker nodes to avoid the head-of-line blocking problem.

The third main direction is to prioritize and reorder latency-sensitive tasks suffering from head-of-line blocking problem on worker nodes and give preference to the tasks with high priority. A typical example is the Shortest Processing Time First (SRTF) algorithm, where a worker node tends to prefers to execute the task whose job has the shortest remaining job completion time rather than the others in the task waiting queue. Eagle deploys the SRTF algorithm with starvation prevention on worker nodes to alleviate the possible head-of-line blocking problem of small tasks by long tasks. In addition to prioritization via scheduling, task preemption is also an alternative option. For example, Big-C [45] implements container-aware immediate preemption and graceful preemption strategies to make tasks preemptive with low cost and latency. Based on these two strategies, Big-C develops a preemptive fair share scheduler to preempt resources from long jobs when short jobs arrive. Inspired by Big-C, Kairos [20] approximates LAS and implements quota-based time sharing on all worker nodes through container-based task preemption as aforementioned. Similar prioritization work based on task preemption includes LAS-MQ [44]. In our proposed Dice [47], the "Opportunistic Preemption" scheme judiciously preempts resources of long tasks only when the task waiting time of short jobs is high. Different from Big-C that always preempts resources from long jobs when short jobs arrive to enforce share fairness and Kairos that always preempts resources from running jobs to enforce quota-based time sharing, Opportunistic Preemption aims to mitigate long task waiting time for short jobs with preemption on demand, while avoiding high resumption overheads of the above "always-on" preemption schemes.

The fourth main direction is to speed up task execution on worker nodes so as to shorten long task waiting time due to the head-of-line blocking. For example, as we know that compute-intensive workloads like Spark queries could take 21 seconds on average on Java Virtual Machine (JVM) warm-up, while IO-intensive workloads like HDFS reads could also spend 33% execution time on warm-up [46]. Therefore, a new JVM of HotTub [46] is proposed to reuse a pool of already warmed-up JVMs among applications to amortize the warm-up overhead over the lifetime of a worker node. Although HotTub is not an optimization dedicated for big data job schedulers, job schedulers can make use of HotTub to accelerate the task execution of analytics jobs created with many JVM-based analytics frameworks like Spark [9]. Similarly, in our proposed Eirene [48], the "Coordinated Cold Data Migration" scheme aims to shorten task runtime and resulting long task waiting time under heavily-loaded periods by migrating cold data for the initial input read phase of tasks for short jobs from hard disk to memory before the input data is used.

2.2.3 Straggler Mitigation

In large-scale datacenters, frequent hardware and software failures are unavoidable and unpredictable in production environments, which may cause some tasks of a job to fall behind others or never complete. Such tasks are usually called "stragglers" or "outliers". If stragglers are not handled in time, the entire job will risk violating its SLO (i.e., predefined job deadline).

Mantri [21] did a systematic investigation of stragglers in a large MapReduce production cluster, and found that the root causes to stragglers include run-time contentions for CPU, memory and other resources, disk failures, varying bandwidth and congestion along network paths, and load imbalance of tasks. To this end, Mantri uses real-time progress reports to detect and restart straggler tasks early in their lifetime. Mantri will not restart stragglers blindly, and it starts the stragglers that lag due to contention for resources and can be sped up if they are restarted elsewhere. Further, Mantri proposes network-aware task placement to avoid hotspots and replicates task output for avoiding interim data loss and mitigating costly re-computations. In addition to restarting misbehaving tasks, another way to mitigate this straggler issue is speculative execution, that is, duplicating tasks that appear to be stragglers. The challenges are how to determine which tasks are stragglers during task execution and when to duplicate the tasks. LATE [49], Longest Approximate Time to End, was proposed to speculatively execute the tasks that are predicted to finish farthest into the future. In particular, LATE estimates the progress rate of each task, and run a speculative copy of tasks that will finish farthest on fast nodes instead of the straggler nodes.

Different from LATE, Dolly [22] is designed to cope with stragglers of short jobs. Instead of waiting, predicting stragglers, and executing speculative tasks, Dolly makes multiple clones of every task of a job, and only uses the result of the clone that finishes first. The main challenge resulting from extra clones is not even exacerbated resource contentions but contentions of reading intermediate data as input. To solve this issue, Dolly uses a cost-benefit model and thus delays assignment to avoid such contention. GRASS [50] is designed to mitigate the straggler issue of approximation analytics jobs. GRASS includes two scheduling algorithms: GS (Greedy Scheduling) and RAS (Resource Aware Speculative). GS greedily picks the task to schedule next that helps achieve the approximation goal the most at the time of scheduling, while RAS takes the opportunity cost into account and schedules a speculative copy of tasks only if it helps save resources and time. GRASS combines these two together by using RAS at the beginning of job execution and switching to GS when the job becomes close to its approximation bound.

In our proposed Eirene [48], the "Scheduler-Aware Task Cloning" scheme aims to duplicate every task of short jobs and use the result of the clones that are completed first under lightly-loaded periods. It leverages the fact of tiny resource usage of short jobs and the availability of free computing resources under light loads, and proactively launches extra copies of short tasks for straggler mitigation.

2.2.4 Fair Job Schedulers

Basically, a fair job scheduler aims to enforce fair sharing of computing resources in the cluster among the users. Guaranteeing scheduling fairness is important for job schedulers to support multiple tenancies in cloud datacenters. In general, a fair scheduler must adhere to the four properties [23]: *sharing incentive*, *strategy proofness*, *envy freeness*, and *Pareto*

Table 2.3: A summary of representative fair job schedulers

Fair Scheduler	**Share fairness metric**	**Supports multiple resource types**	**Supports placement constraints**	**Supports hetero-geneous servers**	**Performance variation aware**
FairSharePolicy in YARN [5]	resource share	No	No	No	No
DRF [23]	dominant share	Yes	No	No	No
H-DRF [51]	hierarchical share	Yes	No	No	No
DRFH [52]	global dominant share	Yes	No	Yes	No
Choosy [24]	resource share	No	Yes	No	No
TSF [53]	task share	Yes	Yes	Yes	No
Eunomia [54]	task progress share	Yes	Yes	Yes	Yes

efficiency. Sharing incentive means that a user is better off sharing his/her resources in the pool, and it is guaranteed that he/she can then run more tasks on the shared resource pool compared with the number for tasks he/she can run with his/her dedicated resources. Strategy proofness means that a user cannot obtain more resources by lying about his/her demands or constraints. Envy freeness means that a user cannot run more tasks if he/she takes the other's allocation. Pareto efficiency means that no user can run more tasks without decreasing another user's allocation.

As one example, YARN has two built-in job schedulers: Capacity Scheduler and Fair Scheduler [5]. Both schedulers are used to share available resources in the cluster among multiple organizations, with capacity and fairness guarantees respectively. Capacity Scheduler partitions CPU and memory resources based on the capacity assigned to organizations, and maintains a job queue for each partition. Fair Scheduler is very similar to Capacity Scheduler, but it is meant to assign available resources to jobs fairly so that each job has an equal share of resources. Note that YARN's Fair Scheduler is a fair scheduler based on the max-min fairness algorithm. Given each user has enough demand and equal share, it maximizes the lowest share first, then the second lowest, and then the third lowest, and so on. In such a policy, when max-min fairness is reached, increasing the share of a user will result in the decrease of the share of the others with equal or smaller allocations (Pareto efficiency) [23]. The attractive feature of the max-min fairness algorithm is to easily support weighted fairness in resource allocations. By assigning different weights to different users in max-min fairness, it is able to allocate resources to each user according to his/her share (equal share becomes one special case where every user has the same weight), and ensure a user's share regardless of the demand of other users. There has been a large body of literature on improving the existing fair schedulers to enable them to adapt dynamic and various workloads and environments, and most of the existing fair schedulers are based on the max-min fairness algorithm.

The original Fair Scheduler in YARN takes only one resource type, CPU, into considerations and uses the number of cores as the metric to determine the quantity of allocations for each user. Later on, a new fair scheduler, called DRF, is integrated into YARN. In YARN, the original Fair Scheduler is then denoted "FairSharePolicy" while the new DRF scheduler is denoted "DominantResourceFairnessPolicy". DRF (Dominant Resource Fairness) [23] is a generalization of the classical max-min fairness to multiple resource types. DRF defines "dominant share" as the maximum share of any resource type a user is allocated, and aims to maximize the minimum dominant share for all the users. Hierarchical DRF (H-DRF) [51] extends the core idea of DRF to the hierarchical schedulers to support a hierarchy of organizations. H-DRF introduces *hierarchical share guarantee* and ensures each group and each node in the hierarchical organization get their prescribed fair shares. DRFH [52] is then proposed to extend the DRF idea to cloud environments with heterogeneous servers. DRFH proposes *global dominant share*, which is the maximum ratio of any resources allocated to a user to the total amount of the corresponding resources in the entire resource pool. Then DRFH tries to equalize the global dominant share of each user and generalizes DRF to multiple heterogeneous servers.

Recent studies of big data analytics workloads in enterprise datacenters show that placement constraints are common for jobs and job schedulers must support schedule constrained jobs on the nodes that satisfy job constraints [11, 12]. Placement constraints impose a big challenge to fair job schedulers. To this end, Choosy [24] extends the classic max-min fairness algorithm and proposes the CMMF (Constrained Max-Min Fairness) allocation policy to support job placement constraints. CMMF incentives users to pool resources and truthfully report resource requirements. Since CMMF is difficult to implement as an online scheduler, Choosy is implemented to closely approximate CMMF as a simple greedy online scheduler. Then, TSF (Task-Share Fairness) was proposed to extend the idea of Dominant Resource Fairness to support placement constraints in multiple-resource sharing environments [53]. The key of TSF is the proposed "task share", which is defined as the ratio of the total number of scheduled tasks of a job to the maximum number of tasks that can be scheduled if the job placement constraints are removed and the job monopolizes the entire datacenter. Then, TSF aims to equalize the task share of each user and maximize the minimal task share first. In Eunomia [54], we take performance variation due to server heterogeneity into considerations and define the "task progress share" metric, which is defined as the ratio between the accumulated task progress given the current allocation and the accumulated task progress if the user can monopolize the cluster. Eunomia aims to equalize progress share of jobs as much as possible, so as to achieve the same slowdown of jobs from different users due to resource sharing and placement constraints, regardless of performance variation. Table 2.3 summarizes the above-mentioned representative fair job schedulers.

Nowadays, how to enforce fair scheduling in the context of datacenters for Cloud services providers becomes an emerging hot research topic [55–57]. Cloud datacenters have distinct differences from enterprise datacenters. For example, Cloud services usually offer features like multi-tenancies and "pay-as-you-use" economic model to end users. The conventional fair schedulers for enterprise datacenters like DRF [23], Choosy [24] and so on are believed not suitable for Cloud environments. To this end, LTRF [55] treats these conventional fair job schedulers as *Memory Less Resource Fairness* because they allocate resources at an instant time without considering the accumulated effects of resource allocations in the long term. In

LTRF a new resource allocation mechanism called *Long-Term Resource Fairness* is proposed to incentive users to submit non-trivial jobs and pool resources via group-buying. LTRF guarantees that over time, a user should receive the amount of resources in terms of the monetary cost as he/she pays. In order to support multiple-resource fair sharing in Cloud datacenters, Reciprocal Resource Fairness (RRF) [56] proposes the “inter-tenant resource trading (IRT)” and “intra-tenant weight adjustment (IWA)” schemes to achieve the economic fairness of multiple resource types among tenants in a cooperative manner.

In order to enforce strict fairness, sometimes tasks of a job will be preempted by job schedulers when the quota is used up. The progress tasks made will be lost, and the tasks must be re-executed. To make things worse, the tasks of another job that be scheduled next may not be able to be assigned to the nodes containing the input data, which will result in expensive data transmission and in turn a longer completion time of tasks. Therefore, how to deliver the best job latency performance under fair sharing or trade fairness for performance improvement is also an important research topic on fair schedulers. For example, Delay Scheduling [58] takes a counterintuitive approach, which relaxes fairness slightly and lets the next job wait for some time for the running tasks of other jobs to finish for better data locality as the nodes containing the needed data will have available slots during the waiting time. As Delay Scheduling favors data locality over strict fairness, Cluster Fair Queueing (CFQ) [59] also relaxes instantaneous fair sharing in DRF [23] and Choosy [24] and prefers to allocate resources to the jobs that complete the earliest under fair sharing. By doing so, CFQ approximates the Shortest Processing Time First (SRTF) algorithm under fair sharing and thus is able to short job completion time with delay guarantees. A similar work to CFQ is HFSP [60]. HFSP also approximates SRTF for job latency performance improvement under fair sharing, by estimating job completion time at runtime and scheduling jobs according to their spent virtual time modeled with a job aging model. Performance-Aware Fairness PAF [61] discovers that for some data-parallel jobs like Spark MLlib [62], more resource allocations result in marginal performance improvement. Inspired by this observation, PAF leverages the demand elasticity of data-parallel jobs and builds a job latency prediction model as a function of the number of allocated slots. Then PAF seeks to transfer slots from a resource-giver job with demand elasticity to a resource-taker job that can obtain performance gains with more slots. By doing so, PAF is able to improve average job latency performance under approximative fair sharing.

2.3 Summary

In this chapter, we first introduce the background of big data job schedulers for large-scale datacenters. We start with the description of job schedulers deployed in high-performance computing environments, and discuss the evolution of big data job schedulers with the development of big data analytics ecosystem. We then elaborate in detail on the 4 important topics that are the most relevant to our work in this dissertation: (a) job scheduler architectures; (b) head-of-line blocking alleviation; (c) straggler mitigation; and (d) fair job schedulers.

Chapter 3

Dice: Improving Short Job Latency Performance in Hybrid Job Schedulers with Elastic Sizing and Opportunistic Preemption

3.1 Introduction

In this chapter, we aim to address the high-latency performance issue of short jobs due to the head-of-line blocking in the context of hybrid job schedulers. We first recognize a fact that long batched jobs and latency-sensitive short jobs are usually mixed together in enterprise datacenters, and recently hybrid job schedulers emerge as attractive alternatives of conventional centralized job schedulers. Then we explore the interplay between performance improvement on short job latency and prioritization of short jobs under the head-of-line blocking problem. In particular, we present a general performance optimization approach to hybrid job schedulers, called "Dice", to mitigate the long job-completion-delay issue of short jobs with two simple yet effective techniques: Elastic Sizing and Opportunistic

Preemption. Then we describe the performance evaluation methodology and results in detail. We evaluate the job-completion-delay performance improvement on short jobs by Dice with three representative traces of enterprise production workloads. Extensive experiment results show that Dice is able to significantly improve latencies of short jobs, at a relatively low cost by marginally increasing the latencies of long jobs.

3.2 Background and Motivation

3.2.1 Mixture of Long/Short Jobs and Hybrid Job Schedulers

Big data analytics workloads in large-scale enterprise data centers tend to be more and more intensive, complex, and diverse, as evident in latest studies of job traces collected from production environments [11, 12, 63–65]. We observe a mixture of both long jobs and latency-sensitive short jobs in enterprise data centers. Although the total number of short jobs could be 10× greater than that of long jobs, they usually consume disproportionally fewer resources than long jobs. For example, over 90% of jobs in Google clusters are short jobs, but short jobs consume only 17% resources [11]. This is because computer clusters in an enterprise are usually shared by different departments for high utilization efficiency. It is common to see analysts and developers submit short but interactive jobs like ad-hoc queries or personalized search, while long-running services and batch jobs occupy a large portion of computing resources in shared clusters.

The latency of short jobs, that is, job completion delay, matters to users because most (if not all) short jobs are user-facing interactive applications like ad-hoc queries for interactive data analysis or personalized search. This fact demands modern-day big data jobs schedulers to be able to schedule large number of different types of jobs with corresponding resource and latency requirements from a variety of analytics frameworks like Hadoop MapReduce [5], Dremel [6], Impala [7], Storm [8], Spark [9], and Flink [10] in timely fashion.

Centralized job schedulers, like YARN [13], Mesos [14], Kubernetes [15], achieve high resource efficiency since they usually have a global view of cluster resource allocations and demands of batch jobs. However, their job scheduling delay becomes non-trivial when scheduling a great amount of latency-sensitive short jobs. Distributed job schedulers like Apollo [40] and Sparrow [16] successfully minimize job scheduling delay with parallel and independent scheduling decision-making of multiple schedulers but they fail to allocate resources efficiently. Recognizing the mixed nature of long batch jobs and latency-sensitive short jobs, Hawk [17], Eagle [18], Phoenix [19] hybrid job schedulers, which in general consist of one centralized scheduler for long jobs and multiple distributed schedulers for short jobs, have emerged as promising alternatives of existing job schedulers for better latencies of short jobs and cluster utilization.

3.2.2 High Latency of Short Jobs under Hybrid Job Schedulers

As Hawk [17] is shown to improve the P50 and P90 job-completion-delay performance of short jobs by 80% and 90% respectively compared with Sparrow, and Eagle [18] further performs up to 80% better than Hawk, our question is raised: is short job-completion-delay performance good enough under latest hybrid job schedulers?

In order to understand performance behaviors of short jobs under hybrid job schedulers, we conduct a trace-driven experimental study with the open-source Eagle simulator, which is able to simulate both Hawk and Eagle schedulers [66]. In our experiments, we try to mimic the same configuration parameters used in Eagle. In particular, we simulate a cluster of 4000 nodes while 2% of nodes are reserved for the short partition because task-seconds of short jobs account for 2% overall task-seconds in the Yahoo trace [63]. Then we feed the Yahoo trace to the simulator as input workload. The Yahoo trace includes 24262 jobs in total where short jobs account for 90.6% (the jobs with the mean task runtime of smaller than 90.58 seconds are defined as short jobs for the Yahoo trace, that is "cutoff task runtime" to distinguish short and long jobs).

We are especially interested in understanding the impact of the task waiting time on job completion delay of short jobs. Therefore, for every 60-second time window in a simulation run, we first collect and report the ratio of job completion delay to mean task runtime of its corresponding short job. Considering an example case where there are two short jobs with 50-second and 5-second mean task runtime respectively, the same job completion delay of 100 seconds may result in totally different user experience. Hence we believe this ratio, instead of the absolute value of job completion delay, is a better indicator of lags caused by resource contentions. Figure 3.1 illustrates the ratio of job completion delay to mean task runtime of short jobs for Hawk and Eagle schedulers under the Yahoo trace.

Second, we also collect and report the mean task waiting time of short jobs until all the jobs are completed for every 60-second time window. The task waiting time for a given task is defined as the duration from the time when its corresponding job is submitted to the time when the task is executed. In general, the task waiting time consists of task scheduling delay and probe queueing delay on the worker node. With multiple and parallel distributed schedulers dedicated for scheduling short jobs, task scheduling delay is guaranteed to be negligible. So the task waiting time of short jobs is actually determined by probe queueing delay. In case the task waiting time for a job outweighs mean task runtime, the task waiting time thus dominates job completion delay. Figure 3.2 plots the mean task waiting time of short jobs under Hawk and Eagle schedulers under the Yahoo trace.

From Figures 3.1 and 3.2, we have two observations: (1) spikes of the mean task waiting time are correlated and contribute to spikes of job completion delay for short jobs; and (2) spikes of the mean task waiting time can be as high as more than 3000 seconds, which is $(3000/90.58 = 33.1)$ times cutoff task runtime for short jobs.

The above observations clearly dictate that shortening the task waiting time is key and imperative to improve job completion delay of short jobs. As we know for hybrid job schedulers a dedicated short partition is used to ensure low latency of short jobs, our first idea is to rethink the sizing of the short partition for better latency performance of short

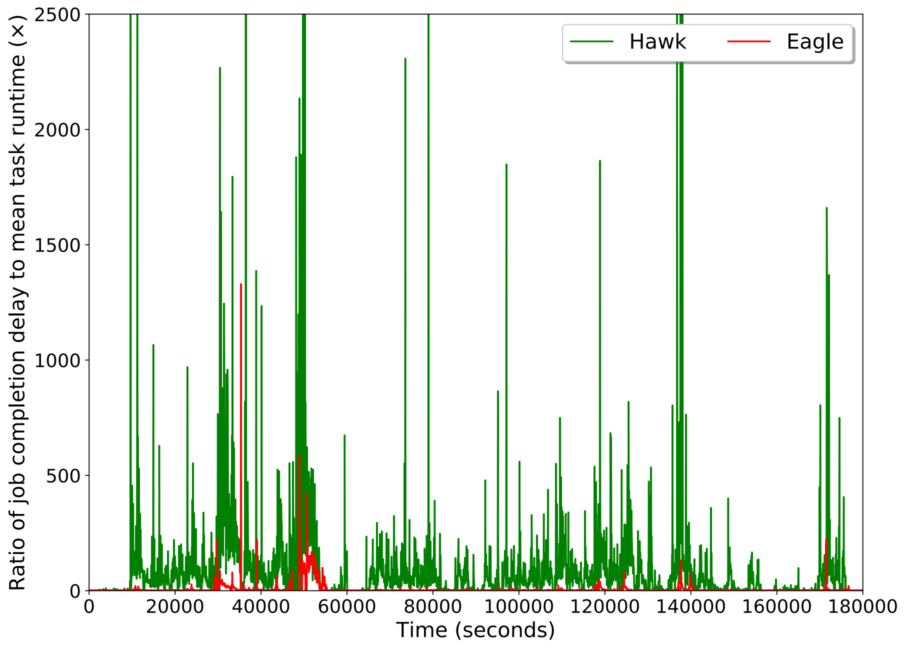


Figure 3.1: Ratio of job completion delay to mean task runtime for short jobs under the Yahoo trace

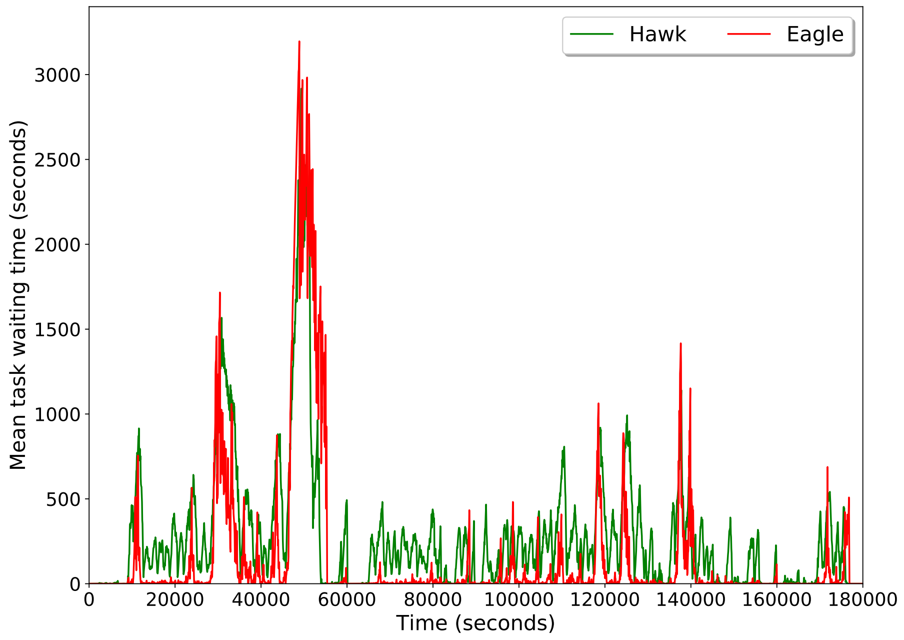


Figure 3.2: Mean task waiting time of short jobs under the Yahoo trace

jobs. On the other hand, since short tasks could be affected by the head-of-line blocking in the general partition, our second idea is to explore the task preemption option.

3.3 Elastic Sizing

In this section, we first quantitatively evaluate the impact of the short partition size on job completion delay as well as cluster utilization. Then we discuss how to strike a good balance between job-completion-delay performance of short jobs and cluster utilization with Elastic Sizing.

3.3.1 Impact of Short Partition Size

Intuitively, a straightforward way to shorten the task waiting time and resulting job completion delay of short jobs is to increase the size of the dedicated short partition. Therefore, we evaluate job-completion-delay performance and cluster utilization as a function of different short partition sizes with the aforementioned simulator. Table 3.1 shows P50, P75, and P90 job completion delays of short and long jobs in both Hawk and Eagle schedulers with the short partition sizes of 2%, 4%, 6%, and 8% under the Yahoo trace. One can see that for Hawk, P50, P75, and P90 job completion delays of short jobs can be improved by 57.8%, 70.3%, and 77.9% respectively if the short partition size is increased from 2% to 8%. On the other hand, affected by fewer worker nodes available for long jobs, P50, P75, and P90 job completion delays of long jobs for the 8% short partition size are 48.0%, 30.0%, and 19.0% higher than those for 2% short partition size respectively. It clearly implies that a bigger size of the dedicated short partition contributes to significant performance improvement on job completion delay of short jobs, with a non-trivial cost to job completion delay of long jobs. We observe a similar pattern from experiment results for Eagle as well.

Let's then take a close look at cluster utilization. Figure 3.3 plots cluster utilization trends in Eagle under Yahoo trace with 2%, 4%, 6%, and 8% short partition sizes. It is clear that

Table 3.1: Job completion delays under the Yahoo trace as a function of different short partition sizes

Short Partition	**Job Completion Delay (seconds)**					
	Short Jobs			**Long Jobs**		
Hawk	P50	P75	P90	P50	P75	P90
2%	327.7	687.1	1226.4	3145.2	6837.4	10089.1
4%	191.2	322.3	457.7	3455.5	7467.1	10754.7
6%	157.0	243.8	331.6	3947.6	8168.3	11447.7
8%	138.2	204.3	270.8	4654.6	8891.4	12010.3
Eagle	P50	P75	P90	P50	P75	P90
2%	29.8	59.1	147.3	3160.0	6837.7	10070.3
4%	23.2	42.0	66.0	3455.6	7469.1	10764.3
6%	22.4	40.4	62.9	3943.5	8166.6	11391.0
8%	22.1	39.9	62.1	4650.2	8901.9	12025.4

cluster utilization is inversely proportional to the short partition size within a certain range. In particular, for the 8% short partition size (that is, 92% general partition size), cluster utilization ranges from 92% to approximately 94% during most of the time, with sometimes 100% peaks. This is expected considering task-seconds of short jobs account for 2% overall task-seconds for the Yahoo trace. Therefore, a bigger size of the short partition could result in lower cluster utilization.

3.3.2 Elastic Sizing

Motivated by the above observations and implications, we propose *Elastic Sizing*, which dynamically and adaptively adjusts the short partition size according to the task waiting time of short jobs, so as to improve job-completion-delay performance of short jobs, while minimizing adverse impacts on the performance of long jobs and overall cluster utilization. Elastic Sizing enforces sizing adjustment of the short partition by converting a number of the general-partition nodes into the short-partition nodes throughout consecutive time windows. In particular, the basic workflow of Elastic Sizing is as follows:

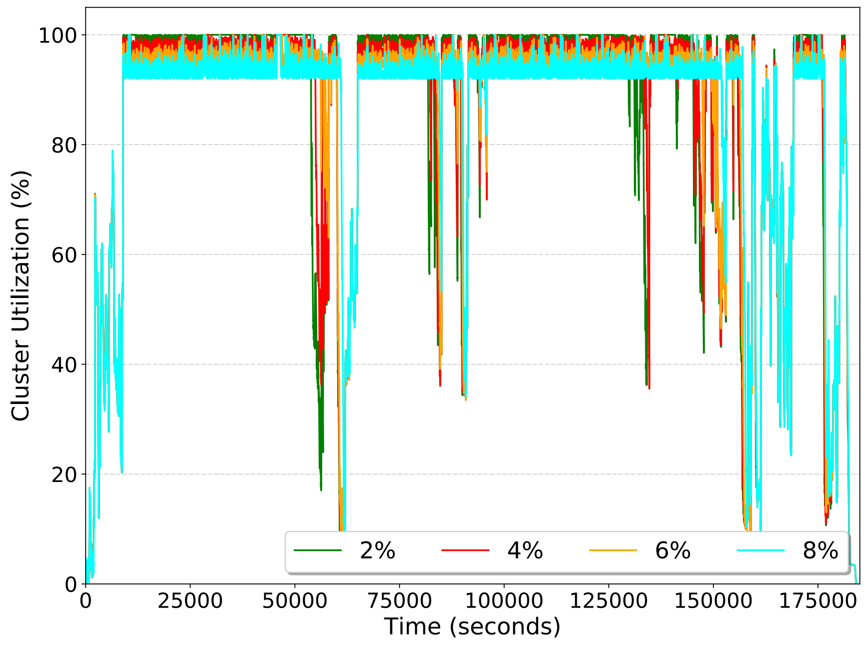


Figure 3.3: Cluster utilization in Eagle under the Yahoo trace

- At the start of a time window, the centralized scheduler of hybrid job schedulers collects the task waiting time of short jobs during last time window from all worker nodes and computes the mean task waiting time;
- The centralized scheduler then decides the number of nodes in the general partition should be temporally converted into the short-partition nodes during the current time window. Implementing node conversion is simple: the centralized scheduler puts the converted nodes onto a blacklist, and avoids scheduling probes of newly-arrived long jobs onto the blacklisted nodes during the current time window. Note that Elastic Sizing requires no changes to distributed schedulers;
- At the end of a time window, the centralized scheduler empties the blacklist.

We then discuss the algorithm to determine the number of nodes to convert. We define the lower bound of the short partition size as *MinShortPartitionSize* number of nodes, the upper bound of the short partition size as *MaxShortPartitionSize* number of nodes. We also define

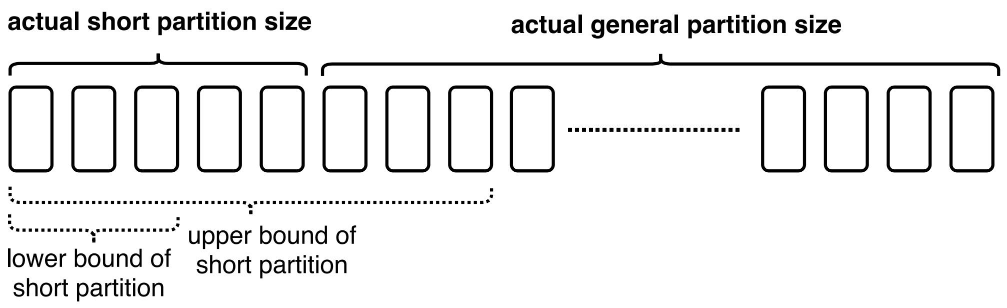

actual short partition size
actual general partition size
lower bound of short partition
upper bound of short partition

Figure 3.4: An illustrative example of Elastic Sizing

the mean task waiting time during last time window as *CurrMeanTaskWaitingTime* and the corresponding maximum mean task waiting time of short jobs as *MaxTaskWaitingTime*. If *CurrMeanTaskWaitingTime* is greater than *MaxTaskWaitingTime*, (*MaxShortPartitionSize* − *MinShortPartitionSize*) general-partition nodes are converted into the short-partition nodes by Elastic Sizing. Otherwise, $p \times$ (*MaxShortPartitionSize* - *MinShortPartitionSize*) number of nodes will be converted, where $p \in [0.0, 1.0]$. Figure 3.4 illustrates an example case of Elastic Sizing.

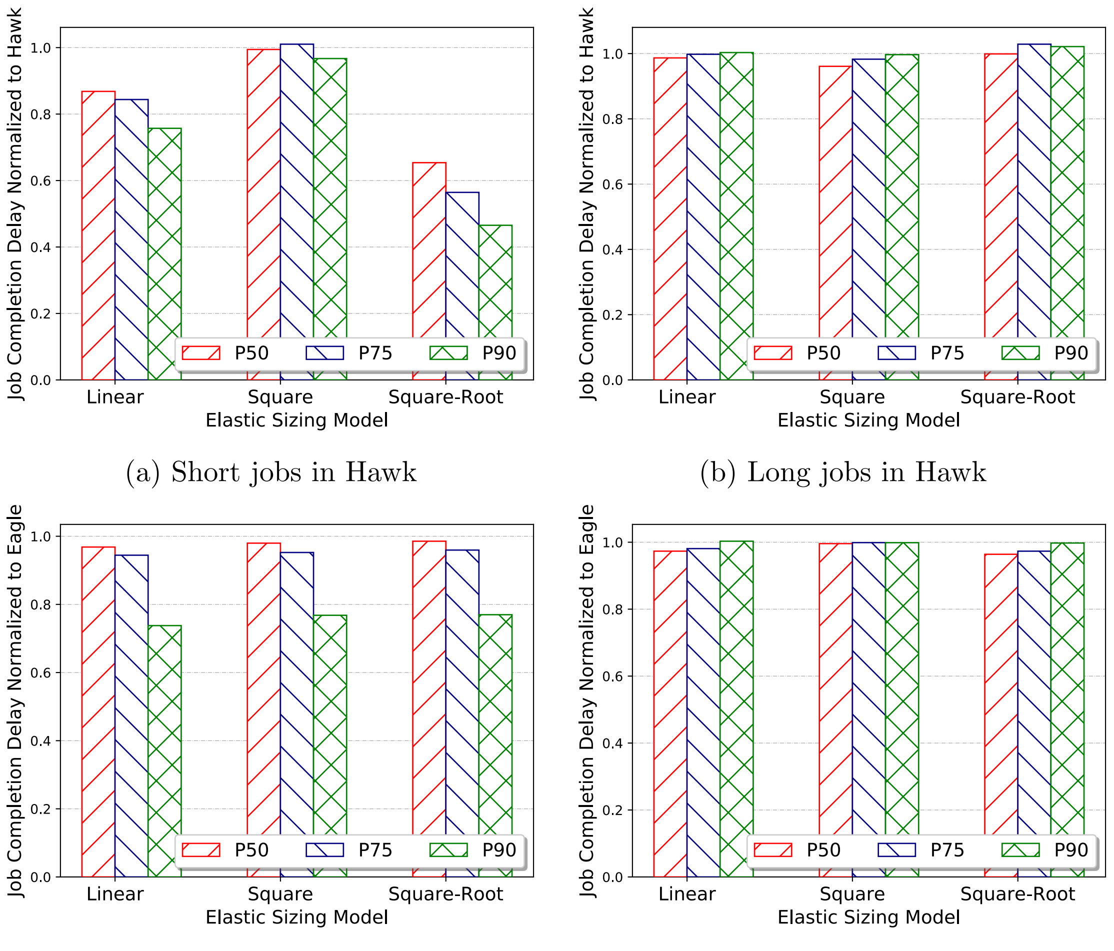
We are interested in the relationship between the rate of node conversions and resulting performance gain. Therefore, we consider and evaluate the below 3 models to compute p because they reflect three different node-conversion strategies: linear conversion, slow-start conversion, and fast-start conversion respectively. This is achieved by leveraging the different responsiveness rate of functions $y = x$, $y = x^2$, and $y = \sqrt{x}$, where $x \in [0.0, 1.0]$.

- Linear model: $p = \frac{CurrMeanTaskWaitingTime}{MaxTaskWaitingTime}$
- Square model: $p = (\frac{CurrMeanTaskWaitingTime}{MaxTaskWaitingTime})^2$
- Square-Root model: $p = \sqrt{\frac{CurrMeanTaskWaitingTime}{MaxTaskWaitingTime}}$

In summary, Elastic Sizing aims to prioritize short jobs over long jobs when short jobs face high task waiting time, by proactively constraining the number of nodes available for scheduling long jobs and thus allocating more resources to short jobs.

3.3.3 Searching Key Parameter Space

In this subsection, we first conduct experiments to understand the relationship between the performance impact and node conversion models. Secondly, we study the performance improvement by Elastic Sizing with different upper bounds of the short partition size.



(a) Short jobs in Hawk (b) Long jobs in Hawk

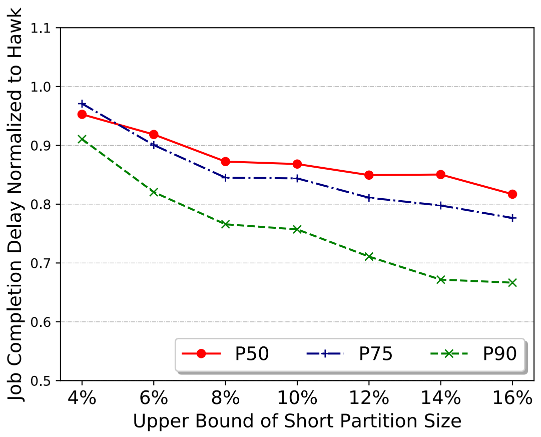
(c) Short jobs in Eagle (d) Long jobs in Eagle

Figure 3.5: Job completion delays for Elastic Sizing with different models normalized to Hawk and Eagle

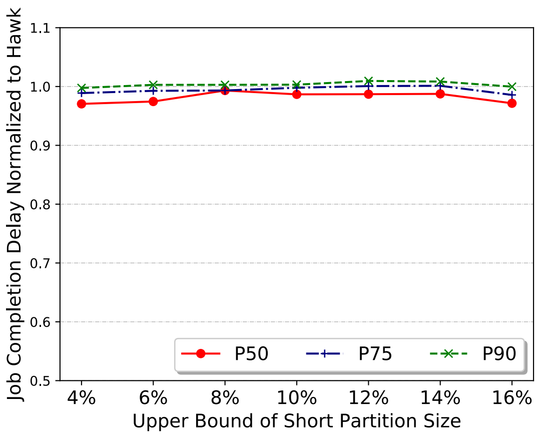
In the first experiment, we configure the upper bound of the short partition size to 10% and the maximum mean task waiting time of short jobs to 1000 seconds. Then we run the simulations of Hawk and Eagle schedulers with different models in Elastic Sizing under the Yahoo trace. Figure 3.5 depicts the job-completion-delay performance in Hawk and Eagle schedulers enhanced with the three node-conversion models of Elastic Sizing. We can see from

Figures 3.5a and 3.5b, Hawk with Elastic Sizing's Square-Root model is able to shorten P50, P75, and P90 job completion delays of short jobs by 34.6%, 43.6%, and 53.5% respectively compared with the original Hawk scheduler. However, this is achieved at the cost of 2.9% and 2.1% longer P75 and P90 job completion delays of long jobs respectively. It is expected because Square-Root model opts to aggressively convert the general-partition nodes into the short-partition nodes. As a counterpart, Square model responds to the increase of the task waiting time of short jobs so slowly that insufficient nodes are converted in time, which is evident in that trivial improvement on job completion delay performance of short jobs is observed for Square model. In contrast, Elastic Sizing's Linear model achieves 13.2%, 15.6%, and 24.3% improvement on P50, P75, and P90 job completion delays for short jobs respectively, with a negligible impact on job completion delay for long jobs. In the meantime, we observe from Figures 3.5c and 3.5d, Eagle is insensitive to different node-conversion models of Elastic Sizing and all the models are able to deliver more than 4% and 23% improvement on P75 and P90 job completion delays for short jobs.

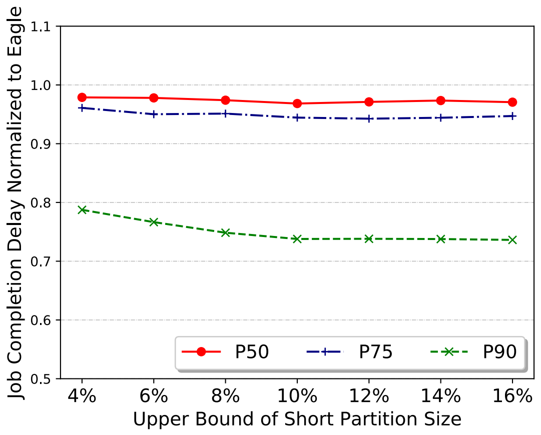
In the second experiment, we vary the upper bounds of the short partition size from 4%, 6%, 8%, 10%, 12%, 14% to 16% with a step of 2% (Note that the lower bound of the short partition size is 2% under Yahoo trace by default), and use the Linear model. Figure 3.6 plots the job-completion-delay performance of Hawk and Eagle enhanced with Elastic Sizing as a function of different upper bounds of the short partition size. One can observe from Figures 3.6a and 3.6c: (1) Elastic Sizing with 16% upper bound of the short partition size improves P50, P75, and P90 job completion delay performance of short jobs by 18.3%, 22.3%, and 33.3% respectively for Hawk, and improves them by 2.9%, 5.3%, and 26.4% respectively for Eagle; (2) with the increase of the upper bound of the short partition size, Elastic Sizing is able to translate higher node conversion into lower job completion delays of short jobs; and (3) Hawk with Elastic Sizing is more sensitive to the upper bound of the short partition size than Eagle with Elastic Sizing.



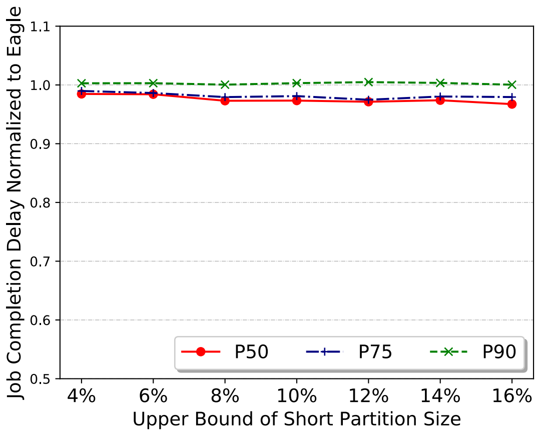
(a) Short jobs in Hawk



(b) Long jobs in Hawk



(c) Short jobs in Eagle



(d) Long jobs in Eagle

Figure 3.6: Job completion delays for Elastic Sizing with different upper bounds of the short partition size normalized to Hawk and Eagle

3.4 Opportunistic Preemption

In this section, we first introduce the background of task preemption in the context of big data job scheduling. Then we present the basic idea of Opportunistic Preemption and explore its key parameter space.

3.4.1 Task Preemption

Process preemption is a commonly-used mechanism to enforce the time-slice quota of running processes and/or the prioritization of higher-priority processes over lower-priority processes in modern operating systems. Recently big data job schedulers have employed task preemption for fair resource sharing and job prioritization enforcement as well. Killing tasks is a simple but costly way to implement preemption because made progress of the tasks is lost and the killed tasks need to be restarted from scratch. On the other hand, job schedulers like Amoeba [67], Natjam [68], etc. checkpoint tasks' progress periodically to save intermediate results to persistent storage. This allows the tasks to be suspended and resumed when needed, which is usually called "checkpointing-based preemption" [69]. Thus whether to enable preemption in job scheduling is mainly determined by the efficiency and overhead of task suspension and resumption with checkpointing. Via lightweight container-based virtualization, Big-C [45] implements immediate preemption and graceful preemption strategies to make tasks preemptive with low cost and latency. Based on these two strategies, Big-C develops a preemptive fair share scheduler to preempt resources from long jobs when short jobs arrive. Inspired by Big-C, Kairos [20] implements time sharing on all worker nodes through container-based task preemption.

Although low latency of suspending and saving task context is achieved with container-based preemption under general workloads, Big-C's experiment results show that Spark tasks with iterative computation are susceptible to high resumption overhead. More importantly, latest studies [46, 61] show that Java Virtual Machine (JVM) warm-up overheads, e.g. class loading and byte-code interpretation, play an important role in short job execution while many popular data analytic frameworks including Hadoop [5] and Spark [9] are built upon JVM. For example, compute-intensive workloads like Spark queries could take 21 seconds on average on JVM warm-up, while IO-intensive workloads like HDFS reads could also spend 33% execution time on warm-up [46].

Without careful considerations of resumption and JVM warm-up overheads, blind preemption, even with low-cost container-based preemption scheme like Big-C, may result in both lengthened job completion delay of short jobs and low cluster utilization.

3.4.2 Opportunistic Preemption

Keeping benefits and possible overheads of task preemption in mind, we propose *Opportunistic Preemption*, which judiciously preempts resources of long tasks only when the task waiting time of short jobs is high. Different from Big-C that always preempts resources from long jobs when short jobs arrive to enforce share fairness and Kairos that always preempts resources from running jobs to enforce quota-based time sharing, Opportunistic Preemption aims to mitigate long task waiting time for short jobs with preemption on demand, while avoiding high resumption overheads of the above "always-on" preemption schemes.

Similar to Elastic Sizing, the centralized scheduler with Opportunistic Preemption enabled periodically collects and aggregates the task waiting time of short jobs during last time window from all worker nodes. When the computed mean task waiting time of short jobs becomes high, Opportunistic Preemption is activated. The question is: how many and what long tasks should be preempted? Considering a data center consisting of tens of thousands of worker nodes and the centralized scheduler does not have detailed information about the running task and the probe queue for every worker node, Opportunistic Preemption computes the total number of needed preemption candidates according to the extent of the mean task waiting time of short jobs and then opts to randomly select worker nodes in the general partition. In particular, the basic workflow of Opportunistic Preemption is as follows:

- At the start of a time window, the centralized scheduler for hybrid job schedulers collects the task waiting time of short jobs during last time window from all worker nodes and computes the mean task waiting time;
- The centralized scheduler then computes the number of nodes in the general partition for preemption candidacy (say n), and sends preemption requests to randomly-chosen n nodes;
- For a node receiving a preemption request, it will save intermediate results of and then suspend the running task if the following four conditions are all met: (1) a long task

is running on the node; (2) no other long tasks are suspended for the node; (3) the number of suspensions for the running task is less than a predefined maximum number of suspensions for a task under preemption; and (4) there are probes of short jobs waiting in the probe queue;

- When the task suspension is completed, the total number of suspensions for the task is incremented by one. A timer that is used to stop suspension with a predefined timeout starts ticking. In the meantime, the node will fetch the probe of a short task according to local scheduling algorithms like SRTF (Shortest Remaining Time First) in Eagle or FIFO (First In First Out) in Hawk, and execute the chosen short task;
- When the timer expires, the node will resume the suspended task after the currently running task is completed.

Figure 3.7 gives 5 illustrative exemplary cases of Opportunistic Preemption. On the figure, the number in the bracket for long tasks denotes the number of preemptions for the task. The maximum number of allowed preemptions per task is 2 in the example. "E" in the *Suspended* column denotes "empty". A preemption request will be fulfilled in case (a) only because it meets all 4 conditions aforementioned. A preemption request will not be fulfilled due to the existence of yet another suspended long task in case (b), no short tasks in probe queue in case (c), running task being not a long task in case (d), the number of preemptions for the running long task being already equal to the maximal number of allowed preemptions per task in case (e).

We then discuss the key parameter space of Opportunistic Preemption. Similar to Elastic Sizing, we define the current short partition size as *CurrShortPartitionSize* number of nodes, the mean task waiting time of short jobs during last time window as *CurrMeanTaskWaitingTime* and the corresponding maximum mean task waiting time of short jobs as *MaxTaskWaitingTime*. If *CurrMeanTaskWaitingTime* is greater than *MaxTaskWaitingTime*, Opportunistic Preemption sends preemption requests to *CurrShortPartitionSize* $\times$ *Multiplier*

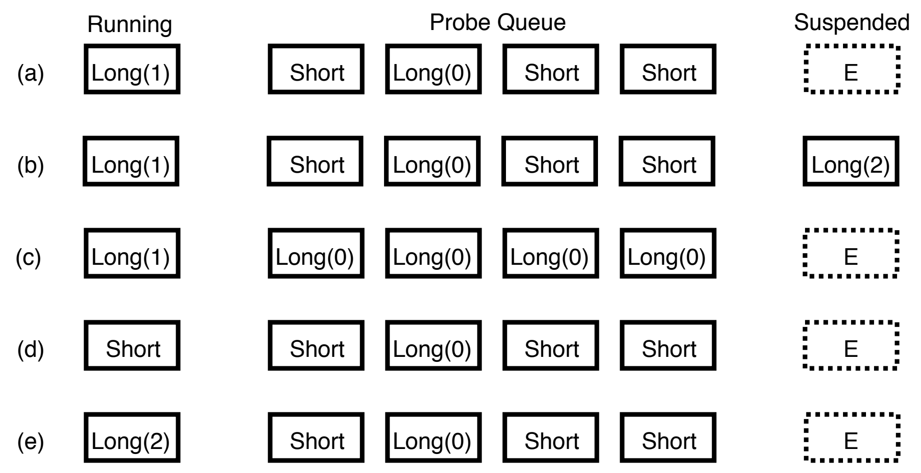


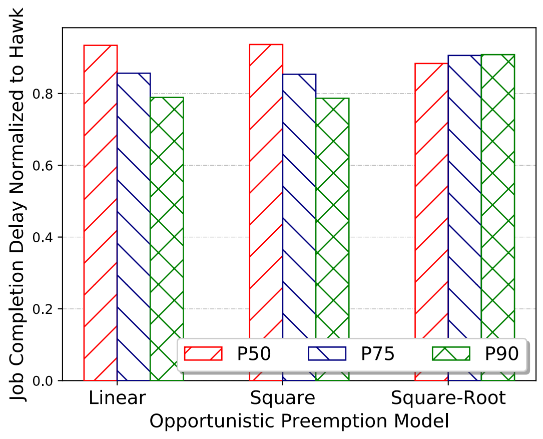
Figure 3.7: Illustrative exemplary cases of Opportunistic Preemption

number of randomly-chosen nodes in the general partition. Otherwise, $p \times (\mathit{CurrShortPartitionSize} \times \mathit{Multiplier})$ preemption requests will be sent, where $p \in [0.0, 1.0]$. *Multiplier* is meant to compensate unfulfilled preemption requests due to the randomization nature of choosing preemption candidates. We consider the same Linear, Square, Square-Root models for p as Elastic Sizing.

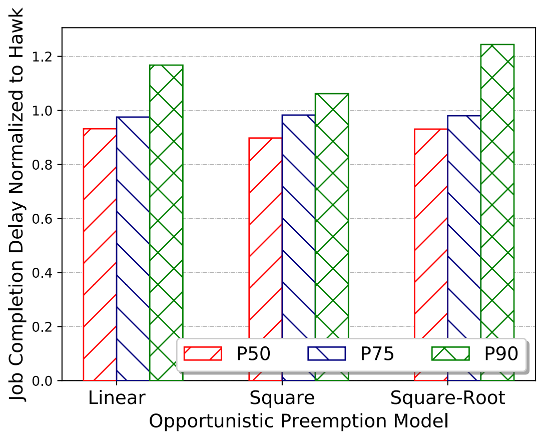
In summary, Opportunistic Preemption aims to mitigate the head-of-line blocking issues caused by long tasks to lower long task waiting time of short jobs, with on-demand task preemption.

3.4.3 Searching Key Parameter Space

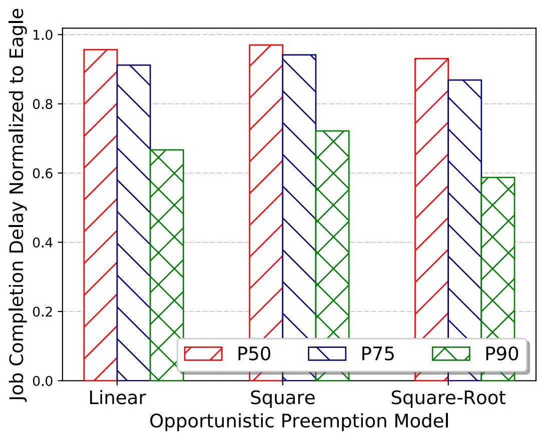
In the first experiment, we assume Opportunistic Preemption is also built on lightweight container-based virtualization, and configure task suspension delay to 3 seconds and task resumption delay to 10 seconds, as on par with experiment results in [61]. Then we run the simulations of Hawk and Eagle schedulers with different Opportunistic Preemption models under the Yahoo trace. We also configure the suspension duration to 100 seconds and the maximum number of allowed preemptions per task to 2. Figure 3.8 depicts the job-completion-delay performance of Hawk and Eagle schedulers with different Opportunistic Preemption models.



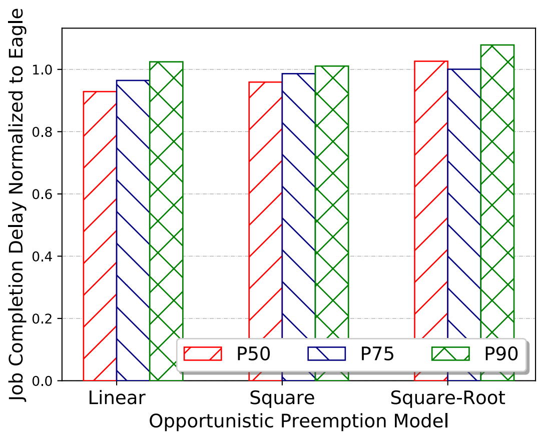
(a) Short jobs in Hawk



(b) Long jobs in Hawk



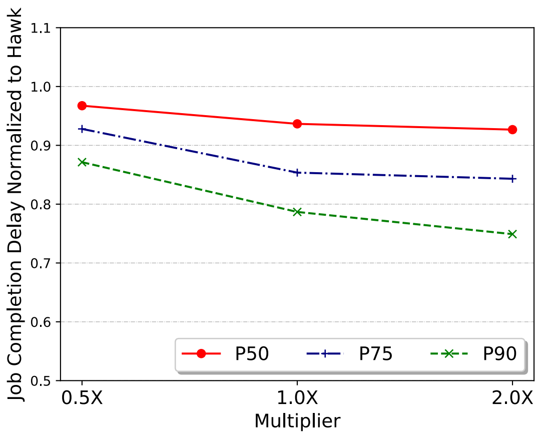
(c) Short jobs in Eagle



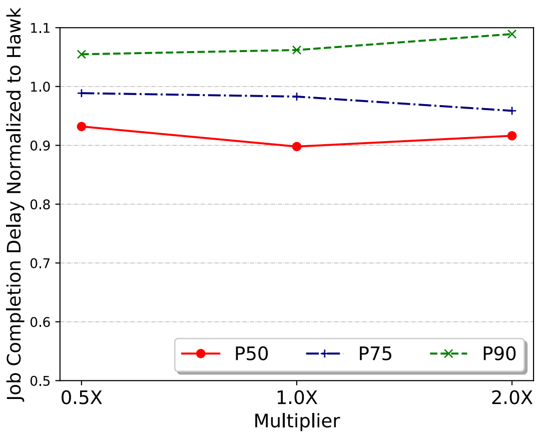
(d) Long jobs in Eagle

Figure 3.8: Job completion delays for Opportunistic Preemption with different models normalized to Hawk and Eagle

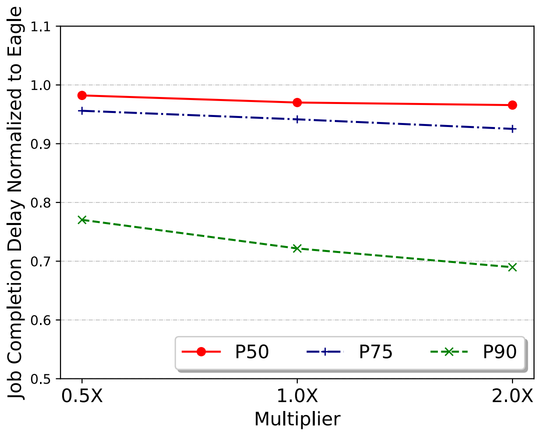
As shown in Figures 3.8a and 3.8c, Opportunistic Preemption is able to consistently improve job completion delays of short jobs under Hawk and Eagle schedulers. For example, Opportunistic Preemption with Square-Root model improves P90 job completion delay of short jobs under Eagle by up to 41.3% while Square model improves P90 job completion delay under Hawk by up to 21.3%. Moreover, experiment results also reveal different characteristics of Opportunistic Preemption compared with Elastic Sizing. First, Opportunistic Preemption is able to shorten job completion delay of short jobs under Eagle more significantly than Elastic Sizing. Second, Square-Root model is not always able to deliver the most performance gains for short jobs with Hawk (see Figure 3.8a) in spite of its aggressive nature, while it



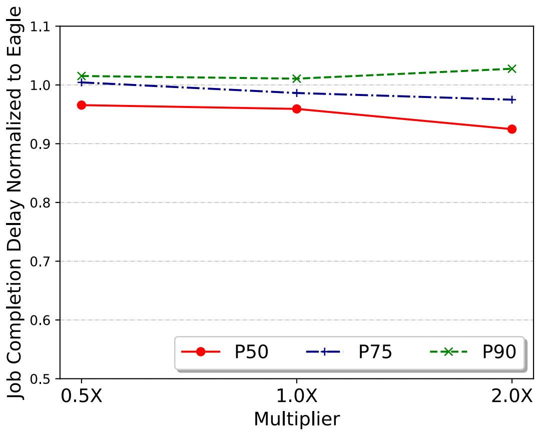
(a) Short jobs in Hawk



(b) Long jobs in Hawk



(c) Short jobs in Eagle



(d) Long jobs in Eagle

Figure 3.9: Job completion delays for Opportunistic Preemption with different multipliers normalized to Hawk and Eagle

could cause the most performance loss for long jobs (20% as in Figure 3.8b). Third, Linear and Square models can achieve similar performance gains but Linear model tends to have more performance loss for long jobs.

The second experiment is to evaluate the impact of different multipliers. We vary the multiplier values from 0.5×, 1.0×, to 2.0×. Figure 3.9 depicts the job-completion-delay performance of Hawk and Eagle schedulers with different Opportunistic Preemption multipliers. As expected, with the increase of multiplier, we can see better improvement on the job-completion-delay performance of short jobs (especially under Eagle) as well as higher cost on job completion delay for long jobs (especially for Hawk). It suggests that Eagle needs an aggressive interference to alleviate long job completion delay of short jobs as its cluster resource is heavily utilized compared with Hawk. This implies that 1.0× multiplier strikes a balance between performance gains on short jobs and performance loss on long jobs.

3.5 Putting It All Together: Dice

A natural idea is to enable both Elastic Sizing and Opportunistic Preemption, which becomes Dice. Dice leverages the commonality of monitoring the mean task waiting time of short jobs in both schemes and makes integration simple. Dice adds the logic into the centralized job scheduler to adjust the short partition size and preempt resources of long jobs when the mean task waiting time of short jobs is high during a time window. On the other hand, Dice recognizes the fact of conservative but nearly cost-free nature of Elastic Sizing and aggressive but potentially costly nature of Opportunistic Preemption, and supplements each other into one unified approach.

In essence, Dice introduces a feedback loop into a hybrid job scheduler for performance optimizations. Latency awareness in Dice enables performance monitoring for short jobs, then actions are taken to activate the two proposed optimizations when performance is below expectation.

3.6 Performance Evaluations

3.6.1 Experimental Setup

We implement a Dice prototype and evaluate its performance in a trace-driven simulator, which is also used to evaluate Sparrow [16], Hawk [17], Eagle [18], Phoenix [19], and Kairos [20]. We feed the simulator with three traces representative of enterprise workloads in large data centers. In particular, we use the Yahoo and Cloudera traces [63, 64], as well as the Google trace [11, 12]. Table 4.4 shows the key job characteristics of the three traces, where the

Table 3.2: Trace characteristics

Trace	**Total jobs**	**Short jobs**	**Task-seconds of short jobs**
Yahoo	24,262	90.6%	2%
Cloudera	21,030	95.0%	9%
Google	506,546	90.0%	17%

percentage of task-seconds of short jobs determines the lower bound of the short partition size.

In our experiments, we simulate clusters of 3000, 4000, and 5000 worker nodes using Yahoo trace and 11000, 12000, and 13000 worker nodes using Cloudera and Google traces to evaluate Dice performance under heavy, medium-heavy, and medium loads respectively. Each worker node has one single core and maintains one probe queue. Since job arrival timestamps are constant during trace replay in the simulations, varying the total number of worker nodes actually varies the workload intensity, and thus directly affects the task waiting time and job completion time.

We enhance Hawk and Eagle job schedulers with Dice, and compare P50, P75, and P90 job completion delay performance with the original Hawk and Eagle schedulers respectively. Every 60 seconds, Dice collects and aggregates the task waiting time of short jobs from worker nodes and computes the mean task waiting time. In Dice's configuration, Elastic Sizing employs Linear model and sets the upper bound of the short partition to 10%, 17%, and 25% for Yahoo, Cloudera, Google traces respectively. This makes room for the short partition size adjustment be a fixed 8% of the cluster size. Opportunistic Preemption employs Square model and sets multiplier to 1.0×.

3.6.2 Results

Figure 3.10 plots normalized job-completion-delay performance for Hawk and Eagle schedulers enhanced with Dice under Yahoo trace as a function of varying numbers of nodes. From

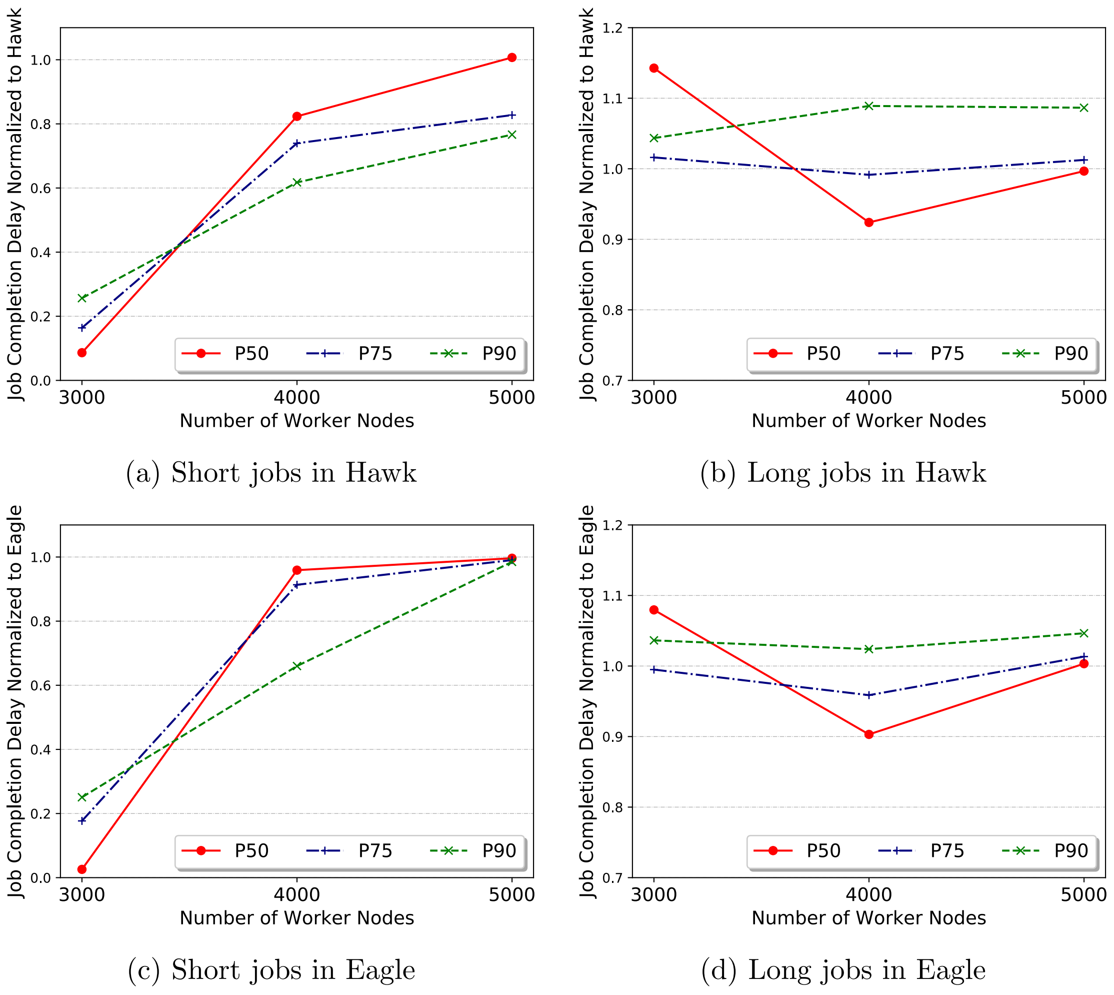


Figure 3.10: Job completion delays for Dice normalized to Hawk and Eagle under Yahoo trace

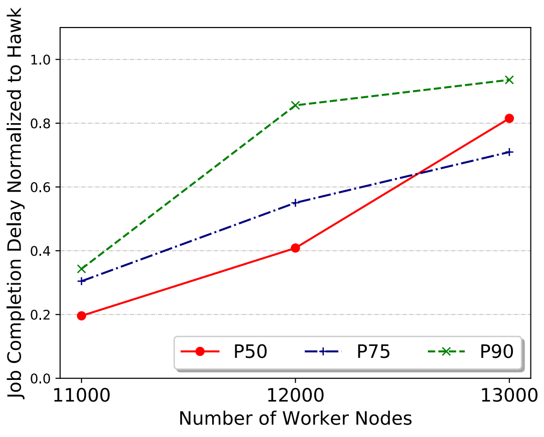
Figures 3.10a and 3.10c, we can see that Dice consistently and significantly improves the latencies of short jobs. First, for the 3,000-node cluster configuration, Dice achieves 91.4%, 83.6%, and 74.4% improvement on P50, P75, and P90 job completion delays of short jobs in Hawk respectively, as well as 97.4%, 82.3%, and 74.9% improvement on those in Eagle respectively. The reason for such a surprisingly significant improvement is that, the task waiting time of short jobs is kept extremely high under saturated load, and thus Dice has to activate both Elastic Sizing and Opportunistic Preemption during the simulation. In other words, high latency of short jobs dictates Dice to maximize the short partition size and proactively preempts resources from long tasks under heavy load to shorten job completion delay of short jobs. On the other hand, we can also see that P50 job completion delay of long jobs is lengthened by 14.3% in Hawk and 8.0% in Eagle. We believe, prioritizing short jobs over long jobs in Dice is imperative to guarantee responsiveness of short jobs under heavy or even saturated loads.

Second, under medium-heavy load (4000 nodes), Dice achieves 17.7%, 26.0%, and 38.3% improvement on P50, P75, and P90 job completion delays of short jobs in Hawk respectively, as well as 4.1%, 8.7%, and 34.0% improvement on those in Eagle respectively. We can also see that 8.9%/2.4% longer P90 job completion delay of long jobs in Hawk/Eagle respectively is traded for such performance improvement for short jobs.

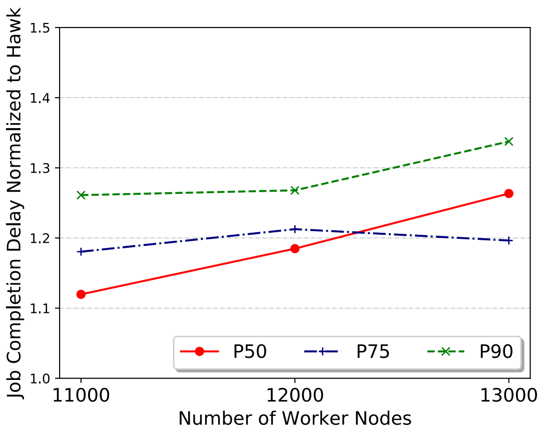
Third, it is clear that performance gains for short jobs decrease with the increase of cluster size. This is expected that lower workload intensity as a result of the bigger scale of a cluster reduces the chances of high task waiting time of short jobs, and Dice has fewer chances to activate Elastic Sizing and Opportunistic Preemption.

We can observe the same patterns from experiment results under the Cloudera and Google traces, as shown in Figures 3.11 and 3.12. More specifically, for a cluster of 12000 nodes under the Cloudera trace, Dice achieves 59.1%, 45.0%, and 14.4% improvement on P50, P75, and P90 job completion delays of short jobs in Hawk respectively, as well as 27.6%, 57.3%, and 11.5% improvement on those in Eagle respectively, while lengthening P90 job completion delay of long jobs by 26.8% in Hawk and 24.7% in Eagle respectively. For a cluster of 12000 nodes under the Google trace, Dice achieves 50.9%, 54.5%, and 43.5% improvement on P50, P75, and P90 job completion delays of short jobs in Hawk respectively, as well as 33.2%, 74.1%, and 85.3% improvement on those in Eagle respectively, while lengthening P50 job completion delay of long jobs by 4.9% in Hawk and P75 job completion delay of long jobs by 14.6% in Eagle respectively. Overall, we confirm that in Dice performance gains obtained for short jobs outweigh performance costs on long jobs.

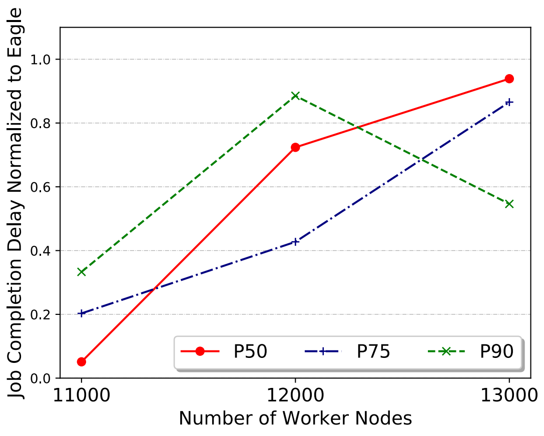
In order to quantitatively evaluate how Elastic Sizing and Opportunistic Preemption individually and combinatorially contribute to performance gains, we conduct experiments of



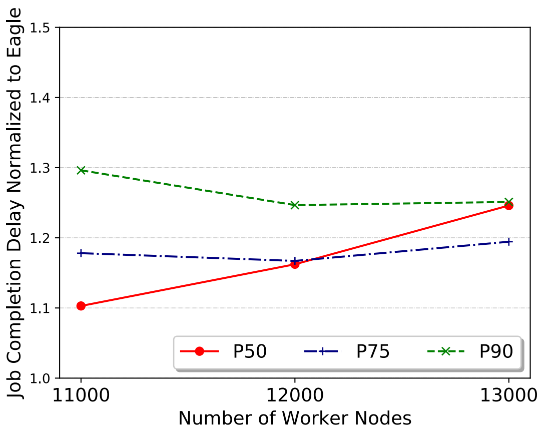
(a) Short jobs in Hawk



(b) Long jobs in Hawk



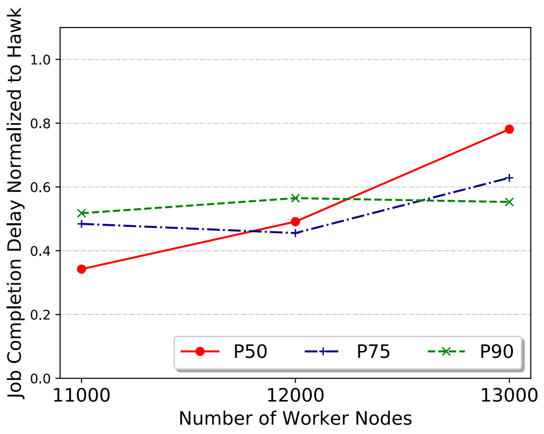
(c) Short jobs in Eagle



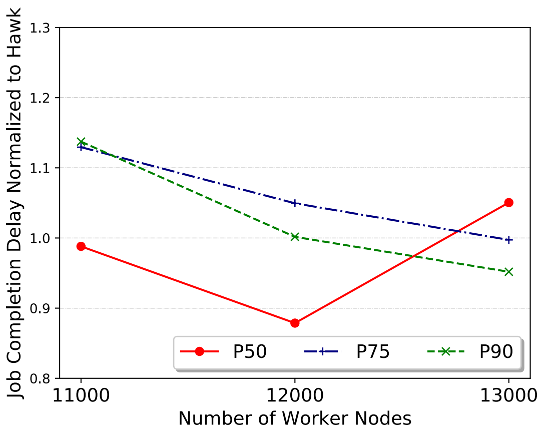
(d) Long jobs in Eagle

Figure 3.11: Job completion delays for Dice normalized to Hawk and Eagle under Cloudera trace

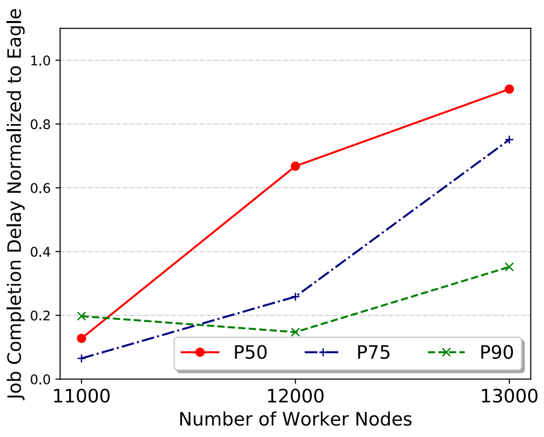
Hawk and Eagle schedulers enhanced with Elastic Sizing, Opportunistic Preemption, Dice under the three traces. Figure 3.13 compares their performance in terms of job completion delay of short jobs. One can see from Figure 3.13a, in the case of Hawk under the Yahoo trace, Dice achieves 5.2%, 12.4%, and 18.5% improvement on P50, P75, and P90 job completion delay of short jobs respectively compared with Elastic Sizing, as well as 12.1%, 13.4%, and 21.5% improvement respectively compared with Opportunistic Preemption. Similarly, we can also see from Figure 3.13b, in the case of Hawk under the Cloudera trace, Dice achieves 3.5%, 0.7%, and -1.4% improvement on P50, P75, and P90 job completion delays of short jobs respectively compared with Elastic Sizing, as well as 55.0%, 38.5%, and 15.3% improvement



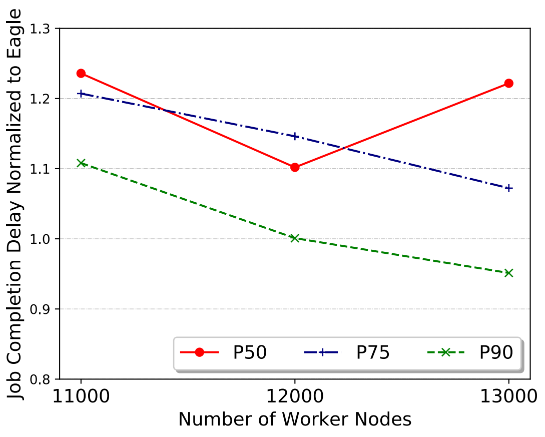
(a) Short jobs in Hawk



(b) Long jobs in Hawk



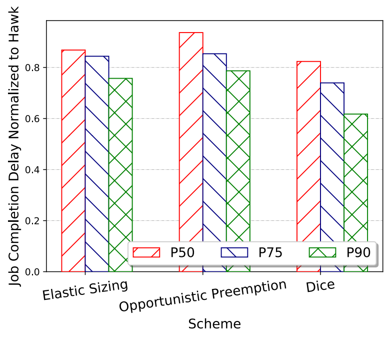
(c) Short jobs in Eagle

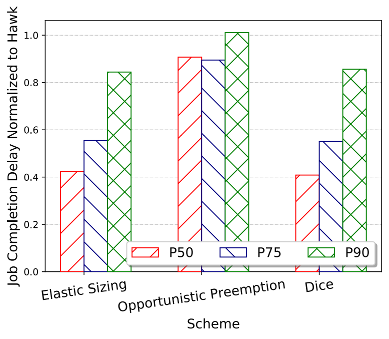


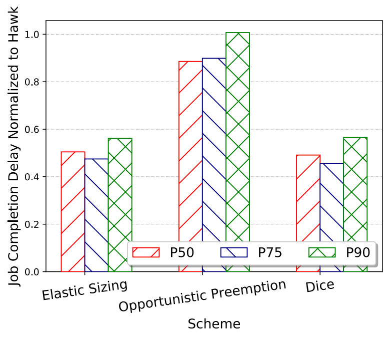
(d) Long jobs in Eagle

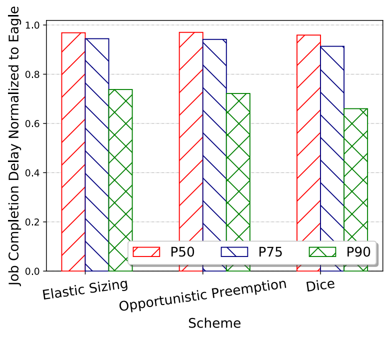
Figure 3.12: Job completion delays for Dice normalized to Hawk and Eagle under Google trace

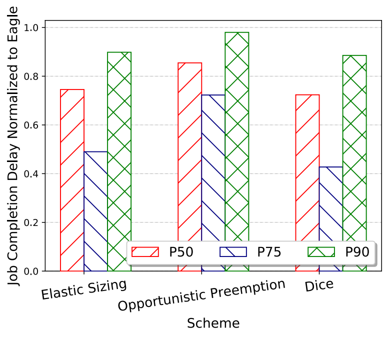
respectively compared with Opportunistic Preemption. It clearly indicates that using Elastic Sizing and Opportunistic Preemption combinatorially is able to deliver more performance gains than using them individually in most cases. We can observe the same pattern from experiment results for the Eagle scheduler and under the Google trace. Even in some cases, the combination of Elastic Sizing and Opportunistic Preemption causes performance loss, the performance loss is negligible (the maximum performance loss we observed from experiment results is -1.4%). This implies that it is empirically beneficial to deploy both Elastic Sizing and Opportunistic Preemption in hybrid job schedulers.


(a) Short jobs in Hawk under Yahoo


(b) Short jobs in Hawk under Cloudera


(c) Short jobs in Hawk under Google


(d) Short jobs in Eagle under Yahoo


(e) Short jobs in Eagle under Cloudera

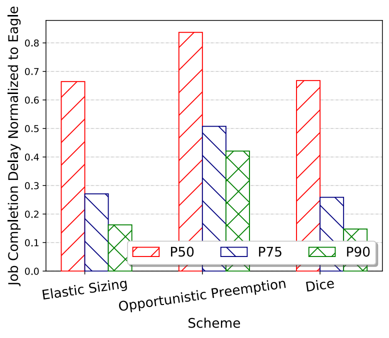

(f) Short jobs in Eagle under Google

Figure 3.13: Job completion delays for Elastic Sizing, Opportunistic Preemption, and Dice normalized to Hawk and Eagle

3.7 Novelty of Dice

Elastic Sizing. In the context of resource management and job scheduling, the most similar work to cluster partitioning in hybrid job schedulers is the node label scheme [70] in YARN [13] and the floating partition scheme in Slurm [2, 71]. Each node in a cluster managed by YARN can be tagged with a label representing ownership or capacity (e.g., GPU support, memory size) and a group of nodes with the same label form a sub-cluster. A label can be set as *exclusive* or *non-exclusive*. A sub-cluster labeled as exclusive can run jobs with the same label only. In a sub-cluster labeled as non-exclusive, resources can be shared with any jobs in the cluster when idle resources are available. Similarly, a cluster in Slurm can be partitioned into disjoint sub-clusters as well, and Slurm's floating partition scheme can be used to share idle resources across partitions. Node label and floating partition are a static partitioning approach, and they are meant to share and leverage idle resources. In contrast, Elastic Sizing is a dynamic partitioning and node conversion approach, which dynamically and adaptively adjusts the short and general partition sizes according to the task waiting time of short jobs, even there are no idle resources. In addition, the basic idea of Elastic Sizing can be easily extended to YARN and Slurm.

Opportunistic Preemption. Task preemption has been widely used in major job schedulers and cluster managers like YARN [13], Mesos [14], Kubernetes [15], Slurm [2] for enforcement of job prioritization. For node label in YARN, a labeled job requesting labeled resources can preempt non-labeled jobs on labeled nodes. Big-C [45] implements a low-overhead task preemption mechanism via the container technique and then develops a preemptive job scheduler to prioritize short jobs over long jobs. Built on top of the container-based task preemption in Big-C, Kairos [20] proposes a two-layer scheduling framework to address head-of-line blocking problem of short jobs: one centralized scheduler for coarse-grained load balancing and local scheduler on every node for achieving Least Attained Service (LAS). In particular, local scheduler preempts the running task that has the highest LAS time when a new task arrives. When the time quota of the running task expires, local scheduler suspends the task and resumes a task with the least LAS time waiting in the task queue. Different from the always-on task preemption in Big-C and Kairos, Opportunistic Preemption is activated on demand, when the mean task waiting time of short jobs is high, to avoid the non-trivial task suspension and resumption overheads and JVM warmup overhead due to preemption.

3.8 Summary

In this chapter, we propose Dice, a general performance optimization approach for state-of-the-art hybrid job schedulers to address the high-latency issue of short jobs due to the bursty nature of workloads in large-scale enterprise data centers. Dice keeps track of the task waiting time of short jobs, and deploys Elastic Sizing and Opportunistic Preemption schemes to optimize job completion delays of short jobs. In particular, Elastic Sizing dynamically and adaptively increases the short partition size to accommodate long task waiting time of short jobs due to the shortage of resource reservation for short jobs, while Opportunistic Preemption preempts resources from long tasks running in the general partition on demand, so as to mitigate the head-of-line blocking problem. Trace-driven experiments show that Dice is able to achieve significant performance gains for short jobs with acceptable performance costs to long jobs.

Chapter 4

Eirene: Improving Short Job Latency Performance in Hybrid Job Schedulers with Coordinated Cold Data Migration and Scheduler-Aware Task Cloning

4.1 Introduction

In this chapter, we still tackle with the same high-latency performance issue of short jobs due to the head-of-line blocking in the context of hybrid job schedulers, but from a different angle. In Chapter 3, we explore the centralized-scheduler-side approaches to prioritize short tasks over long tasks by short-task-performance-aware cluster repartitioning and task preemption. Here we strive to address the performance issue from both the distributed-scheduler and worker node sides. Particularly, we first conduct a motivation study and obtain a deep understanding of workload fluctuation. We observe that short jobs are still facing long job latencies under hybrid job schedulers due to workload fluctuation and straggler task problem. Then we propose Eirene to address the above performance issues of short jobs with two schemes. On one hand, *Coordinated Cold Data Migration* aims to migrate cold data for the initial input read phase of tasks for short jobs from hard disk to memory as so to shorten task runtime and resulting long task waiting time under heavily-loaded periods. Eirene overlaps cold data migration for short tasks waiting in the queue on the worker nodes with the task waiting time, which is achieved by the coordination between distributed schedulers and worker nodes. On the other hand, *Scheduler-Aware Task Cloning* aims to duplicate every task of short jobs and use the result of the clones that are completed first under lightly-loaded periods. Eirene leverages the fact of tiny resource usage of short jobs and the availability of free computing resources under light loads, and proactively launches extra copies of short tasks for straggler mitigation. Then we discuss a prototype implementation of Eirene on top of Eagle [18], and experimental results demonstrate the effectiveness and efficiency of Eirene.

4.2 Motivations

In this section, we first present the findings of workload fluctuation from our experiment study for motivation. Then, we discuss the challenges and opportunities for addressing performance issues of short jobs in the context of workload fluctuation.

4.2.1 Experiment Study of Workload Fluctuation

To investigate performance behaviors of short jobs under hybrid job schedulers, we conduct a trace-driven experiment study with the open-sourced Eagle simulator [66]. In the experiment, we simulate a cluster of 4000, 5000, 6000 worker nodes and then feed the Yahoo trace [63] as input workload to the simulator. 90.6% of the jobs are short jobs in the Yahoo trace, and the total task-seconds of short jobs account for 2% of the overall task-seconds. As a result, 2% of the cluster is reserved for the short partition and the rest of the cluster is allocated for the general partition.

Figure 4.1 plots the mean task waiting time of short jobs under the Yahoo trace in Eagle. Task waiting time is defined as the duration between job submission time and the moment a task of the job begins execution. The reason we are interested in the metric of task waiting time of short jobs is we believe it is a good indicator of system loads and resource contentions. One can see from the figure that during most of the time the mean task waiting time of short jobs is close to 0, which means short tasks are almost immediately executed after their probes are put on worker nodes. This implies that the cluster is under light or moderate loads. On the other hand, we can see there are durations of extremely high mean task waiting time. For example, for the cluster of 4000 nodes, the mean task waiting time reaches a peak of over 3000 seconds. Such a long task waiting time clearly indicates the existence of heavy loads. Considering the mean task runtime of short jobs is less than 90.58 seconds, job latencies of short jobs are thus dominantly lagged by task waiting time. Even with the increase of worker nodes and resulting less intensity of workload, we can still see the lasting peaks of mean task waiting time. It is critically important to find a workable way to significantly reduce task waiting time and task runtime so as to lower latencies of short jobs. As workload fluctuation is evident from our study, we expect a desirable performance optimization approach for short jobs to be able to address performance issues under both lightly- and heavily-loaded periods.

4.2.2 Input Data Read Stage of Tasks and Cold Data Migration

For big data analytics jobs, the stage of reading input data usually accounts for a non-negligible portion of job latency. For example, reading map inputs of SQL queries on Hive takes up to 15% of query duration [72]. For another example, reading inputs from disk causes the first iteration of logistic regression jobs to run 15× slower than late iterations [9]. Such a noticeable duration spent on accessing to singly-read data, that is, cold data, results from the fact that the input stage reads much more data than late stages after filtering and aggregation,

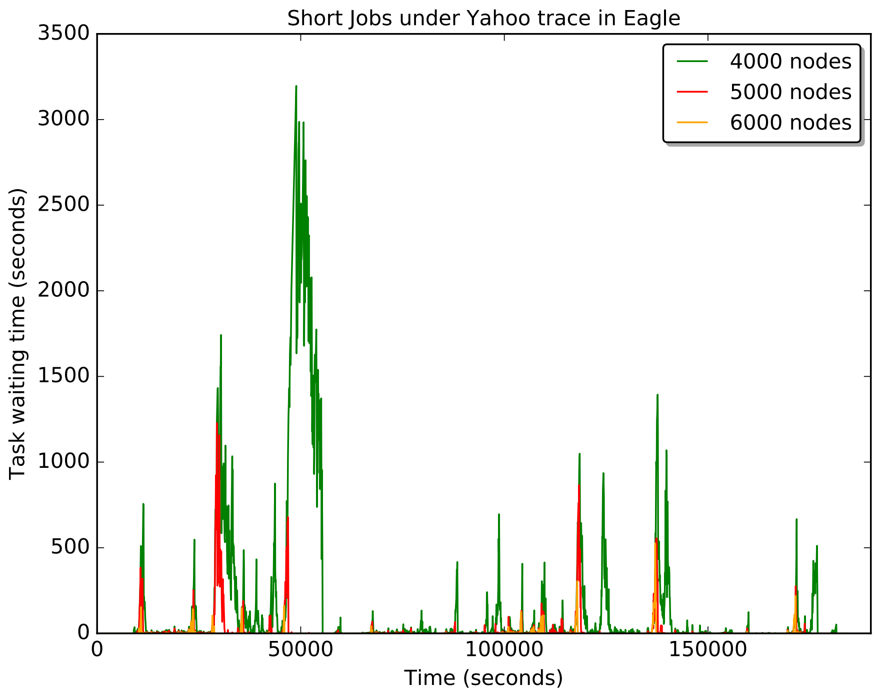


Figure 4.1: Mean task waiting time of short jobs under the Yahoo trace in Eagle

while existing performance optimization approaches based on caching repeatedly/frequently-accessed data, like Spark [9], PACman [73], Triple-H [74], do not benefit the input data read stage of tasks.

To this end, Ignem [75] and DYRS [76] are two systems to migrate cold data from disk to memory before using them at the input read stage of task execution. Experiment results in Ignem show that reading input data from memory is 160× faster than reading from hard disk, and Ignem improves hive queries by up to 34% [75]. The key to effective cold data migration is whether there is sufficient lead time, which is defined as the duration between job submission time and the moment the input data is accessed for a task. As we observe high task waiting time of short jobs under heavily-loaded periods, this becomes the best opportunity to overlap cold data migration with task waiting time of short jobs, which should in turn help reduce task runtime and improves latency performance of short jobs. However, it is very challenging to support cold data migration in the context of hybrid job schedulers because distributed schedulers themselves do not know which worker nodes will execute which tasks when they place probes of jobs onto randomly-chosen worker nodes due to batch sampling and late binding.

4.2.3 Straggler Problem and Mitigation

Stragglers, where one or more tasks of a job take much longer time to complete than other tasks, are commonly seen in enterprise production workloads. For example, more than 15% straggler tasks for 25% phases are observed in a large cluster for the Bing search engine [21]. Moreover, we also analyze 4 representative traces derived from production workloads in enterprise data centers [11, 12, 63, 64] and use the same definition of straggler tasks in Mantri research [21]: the tasks that take 1.5× the median task runtime for a job. We find that in 3 out of these 4 traces, over 30% of short jobs have at least 1 straggler task as shown in Table 4.1. There are many sources contributing to straggler tasks, like transient hardware errors or resource contentions, oversubscribed and congested networks, data skew in workloads (e.g., some tasks may take more input data than others due to imbalanced data distribution), Java just-in-time compilation overhead for the "first task" [21, 72, 77, 78].

As straggler problem is seen widespread, straggler tasks are also considered one major cause to lengthening job completion delay. For one example, job latency was lengthened by stragglers by 29% in Bing clusters [21]. For another example, straggler tasks could take up to 8× longer than the mean task runtime in Hadoop clusters, causing the jobs to be slowed down by 47% on average [22]. As a result, a number of straggler mitigation approaches are proposed to address this issue [21, 22, 49, 50, 72, 77–81], and the common strategy by most of them is *speculative execution*. Speculative execution spawns duplicate copies of straggler tasks when they are detected slow. The fundamental limitation of speculative execution is its hysteresis in response to straggler tasks. This is because speculative execution needs to wait to monitor task progress and collect statistically sufficient samples to detect straggler tasks. More importantly, spawning redundant copies of straggler tasks at this point of time may be too late to be functional. To this end, Dolly [22] was proposed to completely avoid

Table 4.1: Stragglers in 4 traces of production workloads

Trace	**# of short jobs**	**portion of straggler jobs**	**# of short tasks**	**portion of straggler tasks**
Yahoo	$21,981$	58.8%	$514,583$	10.7%
Cloudera	$19,975$	52.9%	$3,897,480$	4.4%
Google	$455,891$	4.4%	$12,867,052$	7.0%
Facebook	$1,145,663$	34.3%	$11,724,548$	8.2%

waiting and speculation by cloning every task of jobs and using the result of the clones that are completed first. The key to the effectiveness of task cloning like Dolly is whether there are sufficient spare computing resources in the cluster so that the duplicate copies of task can be launched nearly at the same time when the primary copies of tasks are started. Blindly applying Dolly's idea to hybrid job schedulers is impractical and could exacerbate job latency performance because it is resource-intensive to duplicate every task of both long and short jobs. This inspires us to consider cloning every task of short jobs by leveraging free resources under lightly-loaded periods, but it remains a challenging question of how to judiciously integrate task cloning into hybrid job schedulers.

4.3 Design and Implementation

In this section, we first describe the design principals and the basic idea of Eirene. Then we discuss the design and implementation of Coordinated Cold Data Migration and Scheduler-Aware Task Cloning schemes in detail respectively.

4.3.1 Design Principals and Basic Idea

We take the below design principals into considerations in Eirene research:

- **Focus on performance improvement of short jobs**. Due to the user-facing and interactive nature of short jobs like ad-hoc and exploratory queries, short jobs are usually more sensitive to job latencies than long jobs. In particular, we aim to improve tail-latency performance of short jobs, which matters the most to user experience;

- **Best-effort approach**. Our strategy is to develop schemes to explore possible opportunities like workload fluctuation and spare resources to improve latency performance of short jobs. It is acceptable to make imperfect decisions and miss opportunities for performance improvement;

- **Keep design and implementation simple**. The goal of Eirene is not to develop a new hybrid job scheduler. Instead, we examine common and inherent performance issues in data centers managed by hybrid job schedulers and propose performance optimization techniques accordingly. We intend to keep design and implementation simple and thus applicable to the latest-of-the-art hybrid job schedulers.

Keeping the above design principals in mind, we propose Eirene on top of the state-of-the-art hybrid job scheduler Eagle [18] to improve latency performance of short jobs while minimizing adverse performance impact on long jobs in the context of hybrid job schedulers. The basic idea behind Eirene is simple: *when possible*, Eirene performs cold data migration to shorten initial input read phase of tasks, and clones every task of short jobs to mitigate straggler tasks. Eirene is not simply applying Ignem [75] and Dolly [22] to hybrid job schedulers. Instead, Eirene leverages the ubiquitous workload fluctuation in enterprise data centers, and judiciously activates cold data migration for short tasks by leveraging long task waiting time of those short tasks during heavily-loaded periods, which is called *Coordinated Cold Data Migration*. On the other hand, Eirene duplicates every task for short jobs by exploiting free resources during lightly-loaded periods, which is called *Scheduler-Aware Task Cloning*. Eirene tightly couples these two functional modules into distributed schedulers and worker nodes of hybrid job schedulers, and significantly improve tail-latency performance of short jobs under fluctuating workloads.

4.3.2 Coordinated Cold Data Migration

Architecture

Note that due to batch sampling and late binding, a distributed scheduler itself does not know which worker nodes will execute which tasks when it sends a batch of probes of a short job onto randomly-chosen worker nodes. This renders distributed schedulers incapable of dictating cold data migration solely. In Eirene, cold data migration is realized under the coordination between distributed schedulers and worker nodes. In the proposed Coordinated Cold Data Migration, worker nodes have delegated autonomy of performing cold data migration in a distributed and parallel manner, while distributed schedulers are responsible for coordinating the data migration efforts of worker nodes and leveraging migrated data for accelerating initial input read phase of tasks.

Figure 4.2 depicts an architectural diagram of a worker node with Coordinated Cold Data Migration support. On the left side of the figure, there are Probe Queue (PQ), Task I/O Engine, and Task Execution Engine from the original worker node design in the general architecture of hybrid job schedulers. PQ is used to enqueue received probes from distributed schedulers. When a worker node becomes ready to execute a task, Task Execution Engine fetches one probe from PQ and requests a task of the corresponding job from the distributed scheduler. When it receives the task, Task Execution Engine launches a container and executes the task program within the container. If the task is involved with data reads or writes, Task I/O Engine is responsible for reading or writing data on the underlying distributed file system like HDFS (Hadoop File System). Coordinated Cold Data Migration augments a *Migration Manager* module, as shown on the right side of Figure 4.2. Migration Manager is meant to migrate cold data of short tasks from disks on local or remote nodes to local memory on the background. It is composed of 4 sub-modules: *Per-Job Task Migration Status Table*, *Migration Queue*, *Migration Fast Lane*, and *RamDisk Virtual File System*. In particular, Per-Job Task Migration Status Table (MST) is used to keep track of data migration progress

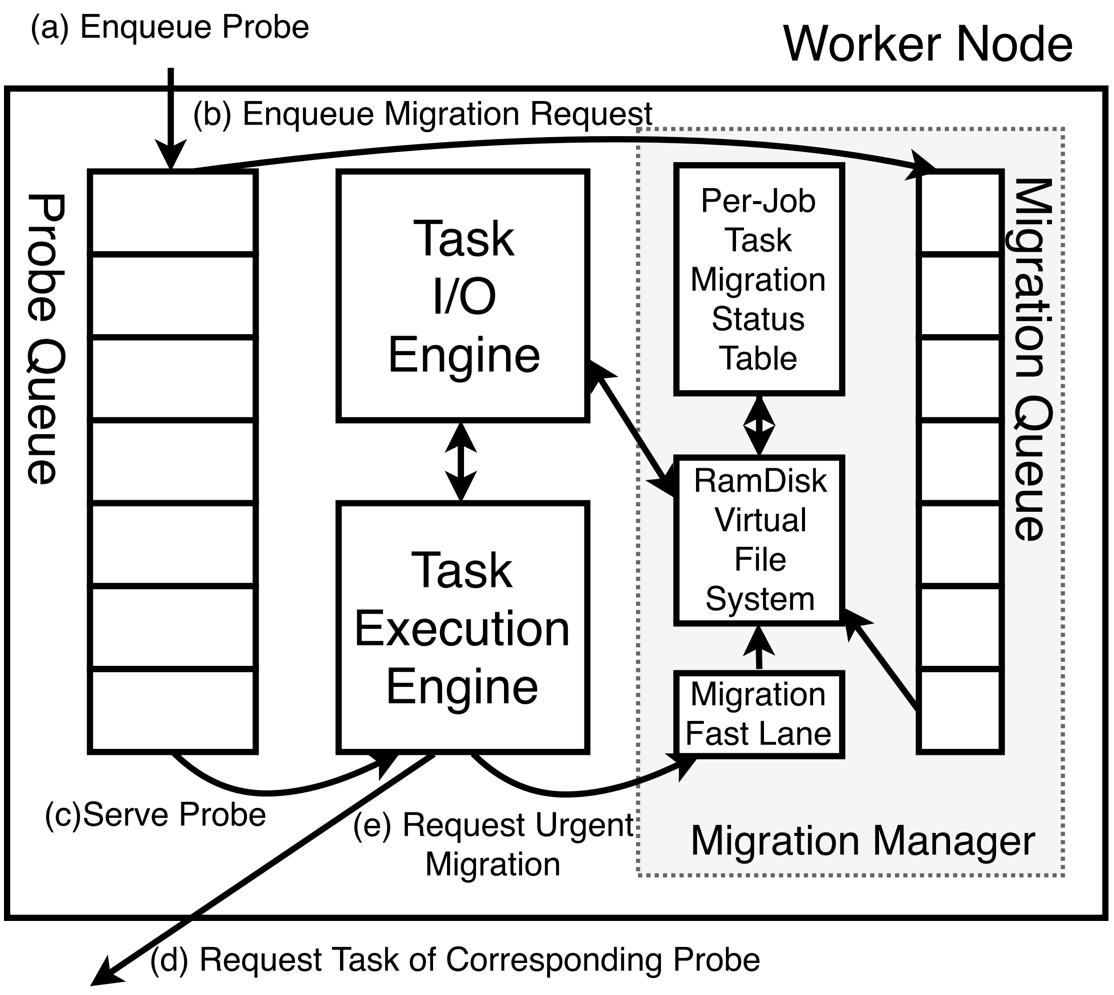

(a) Enqueue Probe
Worker Node
(b) Enqueue Migration Request
Probe Queue
Task I/O Engine
Per-Job Task Migration Status Table
Migration Queue
RamDisk Virtual File System
Task Execution Engine
Migration Fast Lane
(c)Serve Probe
(e) Request Urgent Migration
Migration Manager
(d) Request Task of Corresponding Probe

Figure 4.2: An architectural diagram of a worker node with Coordinated Cold Data Migration support

for all waiting or running jobs on a worker node, including the information about job id, task number, migration status, whether the migrated data is read, and the location of migrated task data on RVFS, as one example MST table shown in Table 4.2. Migration Queue (MQ) enqueues migration requests associated with probes staying in Probe Queue, and its function is to enforce I/O bandwidth management of data migration to avoid contentions on foreground task execution. Migration Fast Lane (MFL) is used to accommodate urgent data migration requests without delays, which is important to allow Coordinated Cold Data Migration to collaborate with the Sticky Batch Probing (SBP) feature in Eagle [18]. RamDisk Virtual File System (RVFS) is responsible for storing migrated data in RamDisk and servicing read

Table 4.2: Per-Job Task Migration Status Table (MST)

job id	**task no.**	**migration status**	**read by task?**	**location of migrated task data on RVFS**
1	0	Not Migrated	N/A	N/A
1	1	Migrated	Yes	/ramdisk/job1_task1_data
1	2	Not Migrated	N/A	N/A
1	3	Migrating	No	/ramdisk/job1_task3_data

Table 4.3: Per-Job Task Status Table (TST)

job id	**task no.**	**task status**	**location of task data on distributed file systems like hdfs**
1	0	Not Started	hdfs://namenode:port/data/fileA_block0
1	1	Completed	hdfs://namenode:port/data/fileA_block1
1	2	Running	hdfs://namenode:port/data/fileA_block2
1	3	Not Started	hdfs://namenode:port/data/fileA_block3

requests from the initial phase of tasks. Moreover, RVFS leverages the MST information to evict migrated data and reclaim space for new task data being migrated.

On the other hand, distributed schedulers in Coordinated Cold Data Migration maintain Per-Job Task Status Table (TST) to track the task progress of jobs as well as the location of the input data on the underlying distributed file system for all the tasks of every short job, as one example TST table is shown in Table 4.3. More importantly, distributed schedulers augment every probe with the TST information upon placing them onto worker nodes.

Workflow

In Coordinated Cold Data Migration, worker nodes have delegated autonomy of decision making of data migration, that is, a worker node itself decides whether and which tasks for data migration given a probe in its Probe Queue (PQ). When a worker node receives a probe of a job containing the TST table and enqueues the probe in PQ, Eirene generates a predefined number of random task numbers and enqueues migration requests of them into the MQ queue. On the other hand, when a worker node receives the response from the distributed scheduler about the next task of a job to execute accompanied with the TST table, the worker node will examine both TST and MST tables to see if there is any task of the same job, which is not started and whose data is not migrated. If yes, the worker node will put the task number into MFL, and perform data migration immediately. By doing so, the worker node is able to overlap the execution of the current task with data migration for the next task to execute, aligned with the "Sticky Batch Probing (SBP)" scheme in Eagle.

At the side of the distributed scheduler, it reads the MST table embedded in the request of the next task by the worker node and thus knows which tasks have data already migrated to memory. Then it chooses one random task from the tasks that are not started but whose data have been migrated to memory, and responds to the worker node with the chosen task number. If none of such tasks is found, the distributed scheduler just returns any "Not Started" task number to the worker node. By doing so, the coordination between a distributed scheduler and a worker node is able to maximize potential performance gains from cold data migration.

Figure 4.3 illustrates an example of the interaction and coordination between a distributed scheduler and a worker node in Coordinated Cold Data Migration. T0-T3 denote tasks 0 to 3 of job 1 respectively. In TST, "S", "R", "C" denote "Scheduled", "Running", and "Completed" respectively. In MST, "M" and "X" denote "Migrated" and "Migrating" respectively. Figure 4.3(a) shows a point-in-time system snapshot *after* a distributed scheduler receives a new job, say Job 1 (J1), and sends one probe containing the TST table to a worker node. At the time, the worker node is executing one task of Job 0 (J0), as we can see that a probe of J0 is staying at the head of PQ, followed by the probe of J1. The worker node reads the TST table and knows none of J1's tasks is completed or running, so T1 of J1 is randomly chosen and enqueued into MQ for data migration. Because of no ongoing data migration, the worker node starts to migrate data for T1 immediately, as T1 is marked "migrating" in the MST table. Figure 4.3(b) shows a system snapshot *before* the worker node requests the next task to execute from the distributed scheduler, while T1 is shown to be migrated already. Figure 4.3(c) shows that the distributed scheduler returns T1 to the worker node. The worker node then executes T1 and starts to migrate data for T3 immediately. As shown in Figure 4.3(d), the worker node finishes the execution of T1 and data migration for T3,

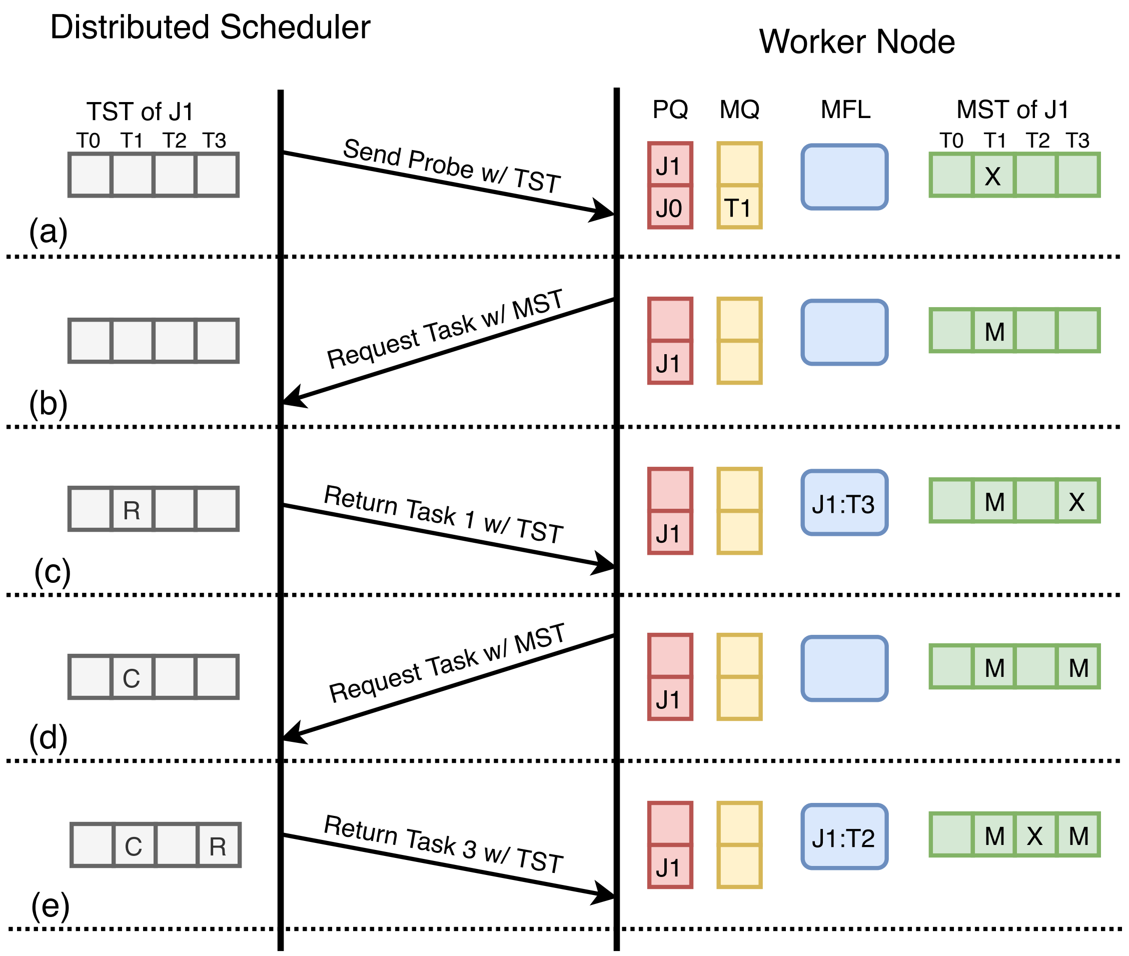


Figure 4.3: An example of the interaction between distributed scheduler and worker node. J0 and J1 denote jobs 0 and 1 respectively.

and requests the next task from the distributed scheduler again. Figure 4.3(e) shows that the distributed scheduler responds to the worker node with T3, and the worker node then starts to execute T3 and performs data migration for T2.

Implementation Issues

We discuss a few key implementation issues in the below:

Listing 4.1: Example code snippet of reading input data in Python

```
import os

# assume HDFS is used as the underlying distributed file system.
# assume hdfsRead() and rvfsRead() are provided APIs to read data
# from HDFS and RVFS respectively.
def readInputData(location_on_hdfs):
    # if environment variable of data location on rvfs exists.
    if os.environ['location_on_rvfs']:
        # read data from rvfs
        dataBuffer = rvfsRead(os.environ['location_on_rvfs'])
    else:
        # read data from HDFS
        dataBuffer = hdfsRead(location_on_hdfs)

    return dataBuffer
```

- *How to enforce I/O bandwidth management with MQ?* Migration Manager specifies the maximum number of allowed concurrent migrations (CM) to limit the maximum disk bandwidth for data migration. Note that experiment results in Ignem [75] show that it takes 6.42 seconds on average to read an HDFS block of 64MB from hard disk to memory. Assuming CM is 10, then disk bandwidth used for data migration is $10 \times \frac{64}{6.42} = 99.6$ MB/s at the peak, which is lower than sustainable sequential read throughputs of hard disks on the market. When the current number of concurrent migrations reaches the CM threshold, the additional migration requests will be enqueued and waiting in MQ.

- *How to speed up reading input with migrated data on RVFS?* Remember that the request of the next task to execute sent by the worker node contains the MST table, which includes the location of migrated data on RVFS. Therefore, the distributed scheduler leverages such information and passes the location of migrated data to the task. Moreover, a small modification to the task program is made to read migrated data from RVFS if available, or otherwise read data from the original location on the underlying file system. A simplified version of example code snippet to read input data is shown in Listing 4.1.

- *How to minimize unnecessary cold data migration?* Note that Eirene generates a predefined number of task numbers and enqueues them in the MQ queue for future data migration when a worker node receives a probe of a job from the distributed scheduler. It is possible for the migration of cold data for a task being performed later than the start time of task execution due to concurrent migration bandwidth control, which renders data migration for the task a waste of time and I/O bandwidth. To this end, Migration Manager will cancel the migration request of a task if the task already starts execution, and fetch the next migration request in the MQ queue when migration bandwidth becomes available.

4.3.3 Scheduler-Aware Task Cloning

In order to effectively mitigate stragglers and improve latencies of short jobs, Scheduler-Aware Task Cloning in Eirene duplicates every task of short jobs and uses the results of the tasks that finish first, the same as Dolly [22]. However, Scheduler-Aware Task Cloning distinguishes itself from the existing approaches like Dolly in two important aspects as below:

- *Short Jobs Oriented.* Scheduler-Aware Task Cloning is applied to short jobs only. This is because Eirene recognizes the fact in production environments short jobs consume a very small portion of resources but they are also latency sensitive. Replicating all tasks of short jobs will not likely result in resource contentions. More importantly, Scheduler-Aware Task Cloning leverages the fluctuating nature of workloads and activates task cloning only under idle or lightly-loaded periods.

- *Distributed Scheduler Aware.* Scheduler-Aware Task Cloning intentionally minimizes changes to hybrid job schedulers for simplicity and feasibility. As a result, it is designed to be tightly coupled with a distributed scheduler to leverage its inherent feature of "batch sampling".

In detail, for a given short job consisting of N tasks, the original batch-sampling scheme sends probes to $2 \times N$ randomly-chosen worker nodes. After the N tasks are assigned to the first N probes whose worker nodes request the distributed scheduler of tasks to execute, the later N probes are canceled when the corresponding worker nodes request tasks to execute from the distributed scheduler. Then the worker nodes fetch the next probes and may execute tasks of other jobs, like the example shown in Figure 4.4(a). Figure 4.4(a) shows the timeline of task execution of a 4-task job ("J0") under the original batch sampling scheme. t_0 denotes the time when J0 is submitted to the job scheduler. t_1 denotes the completion time of T1, which is the straggler task. Job latency of J0 is thus $t_1 - t_0$. In contrast, Scheduler-Aware Task Cloning tries to leverage such probes and repurpose them to represent clones of tasks that have not been completed. In addition to the "task status" column in Per-Job Task Status Table, Scheduler-Aware Task Cloning adds one new column, called "cloned task status", to track the status of cloned tasks. When the distributed scheduler receives a request of a task to execute from a worker node, it checks if all the primary copies of tasks have been launched or completed ("Started" and "Completed" status in the "task status" column). If yes, the distributed scheduler will try to find a running task that has no clone (from the "cloned task status" column) and return its task number to the worker node. The worker node does not differentiate a primary copy or a duplicate copy of a task. It just launches a container and executes the received task program from the distributed scheduler. The distributed scheduler may receive two task completion messages, but it marks the task as "Completed" in the table only when the first one arrives and simply ignores the second one. An example timeline of task scheduling and execution with Scheduler-Aware Task Cloning is shown in Figure 4.4(b). Figure 4.4(b) shows the timeline of the task execution of J0 with Scheduler-Aware Task Cloning. Note that the probes on worker nodes E - G are repurposed to represent and execute cloned tasks. Because of Sticky Batch Probing, the probe on worker node A is also repurposed to represent and execute the last clone task after T0 is completed. T1's clone completes at t_2, so job latency of J0 becomes $t_2 - t_0$, which is shorter than $t_1 - t_0$.

It has to be noted that Scheduler-Aware Task Cloning also cooperates with Coordinated Cold Data Migration for maximizing the performance potential of data migration. In determining which task to clone, Scheduler-Aware Task Cloning gives the preference to the

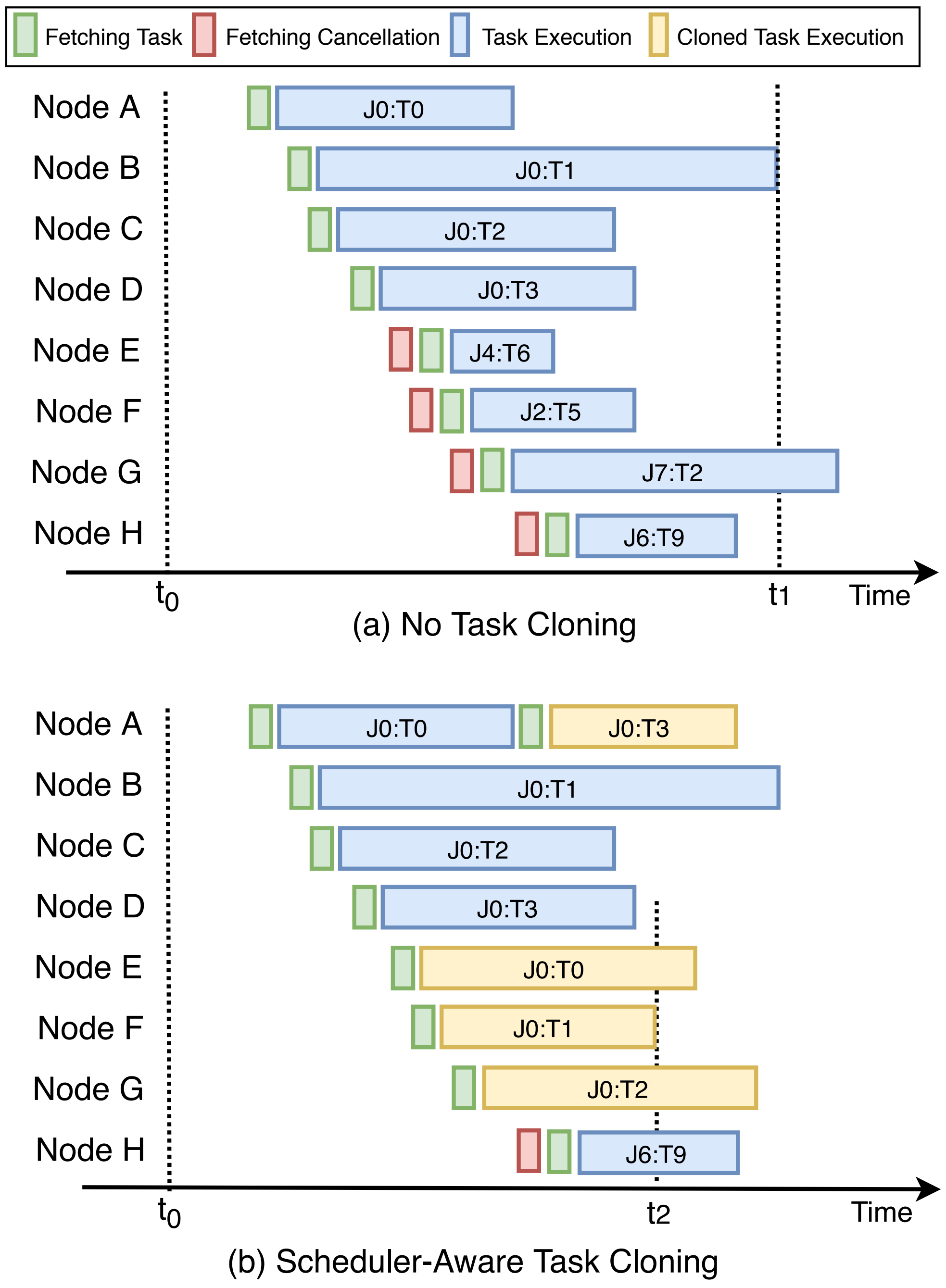


Figure 4.4: An example with and without Scheduler-Aware Task Cloning

worker nodes that have migrated data of tasks that have no clones yet. However, we expect the chances of Coordinated Cold Data Migration and Scheduler-Aware Task Cloning being both activated to be low because they are usually activated under different load intensity conditions.

Table 4.4: Trace characteristics

Trace	Total jobs	Cutoff task runtime	Short jobs	Task-seconds of short jobs
Yahoo	24,262	90.6 seconds	90.6%	2%
Cloudera	21,030	272.8 seconds	95.0%	9%
Google	506,546	1129.5 seconds	90.0%	17%
Facebook	100,000	76.6 seconds	98.0%	2%

4.4 Performance Evaluations

4.4.1 Experimental Setup

In order to evaluate the effectiveness and efficiency of the proposed Coordinated Cold Data Migration and Scheduler-Aware Task Cloning schemes, we implement a prototype of Eirene on top of the state-of-the-art hybrid job scheduler, Eagle [18] in the open-sourced Hawk/Eagle simulator [66], which is widely used in research work of Sparrow [16], Hawk [17], Eagle [18], Phoenix [19], Kairos [20].

We feed the same traces used in Section 4.2.3 as input workload to the simulator, and the detailed characteristics of traces are shown in Table 4.4. The original Facebook trace is long, we just use the first 100,000 records in the experiments to reduce simulation time. The jobs with mean task runtime less than cutoff task runtime are short jobs. The ratio of total task-seconds of short jobs to those of all jobs also dictates the size of the short partition.

Table 4.5 shows configuration parameters of simulations. Note that MPP denotes the maximum number of randomly-generated task numbers to migrate per probe when a probe is placed into the PQ queue by the distributed scheduler. Data migration time of 6.42 seconds and task performance speedup ratio of 15% is cited from the Ignem research [75]. Regarding the Scheduler-Aware Task Cloning scheme, since the traces do not include task runtime information for cloned tasks, we use runtime of randomly-chosen primary copies of tasks of the same job to project runtime of cloned tasks. Table 4.5 also gives the configurations of cluster size used in our experiments. We vary cluster sizes to study the scalability of Eirene.

Table 4.5: Configuration parameters of simulations

Description	**Abbr.**	**Values**
Coordinated Cold Data Migration (CCDM)		
Max. number of concurrent migrations	CM	10
Max. number of migrations per probe	MPP	2
Data migration time (seconds)	DMT	6.42
Task performance speedup ratio (%)	TPS	15
Scheduler-Aware Task Cloning (SATC)		
Projected runtime of cloned tasks	PRT	random
Cluster scale	**Values**	
Cluster sizes for the Yahoo trace	3500, 4000, 4500, 5000	
Cluster sizes for the Cloudera trace	13000, 13500, 14000, 14500	
Cluster sizes for the Google trace	12000, 13000, 14000, 15000	
Cluster sizes for the Facebook trace	50000, 55000, 60000, 65000	

In particular, the cluster sizes are carefully chosen to demonstrate the performance trend from the overloaded case to the moderately-loaded case. We run simulations with bigger cluster sizes but the experiment results are consistent with the trend, so we omit to report most of their results in the chapter.

Regarding the performance metrics, we consider 50-percentile (P50), 75-percentile (P75), 90-percentile (P90) job latencies as key metrics to evaluate the tail-latency performance of short jobs. Moreover, we focus on normalized performance numbers, which are the ratio of P50, P75, P90 latency numbers from Eirene to the ones from the original Eagle respectively.

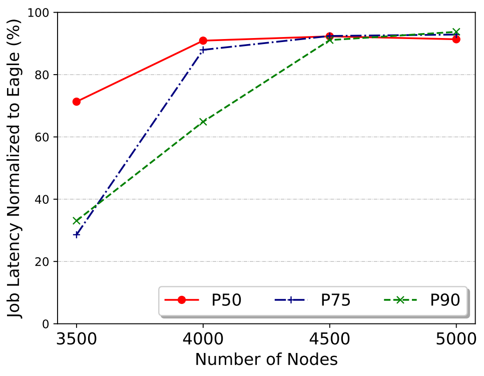
4.4.2 Results

Performance Analysis

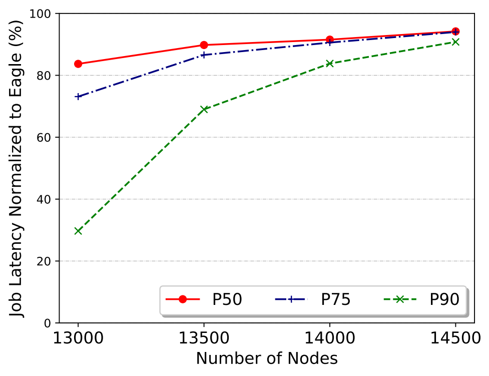
Figure 4.5 depicts the normalized latency performance of Eirene compared with Eagle as a function of different cluster sizes, under the four traces. We obtain a few observations from the figure. First, we clearly see the performance improvement by Eirene across the 4 different workloads. Second, significant performance improvement is observed under the overloaded case. For example, Eirene improves P50, P75, P90 latency performance of short jobs by up to 39.2%, 79.1%, 81.3% respectively compared with Eagle, under the Facebook trace with a cluster of 50000 nodes. Furthermore, we can see that in the overloaded case, Eirene improves Eagle's P90 latency performance of short jobs by 66.9%, 70.3%, 67.4%, 81.3% under the Yahoo, Cloudera, Google, Facebook traces respectively. It is clear that Eirene is able to drastically improve tail-latency performance of short jobs by shortening task waiting time and resulting long latencies of jobs. Third, we can observe that the performance improvement is decreased with the increase of cluster size. In the moderately-loaded case, we can still see that Eirene is able to deliver non-trivial performance improvement. For example, Eirene improves Eagle's P50, P75, P90 latency performance of short jobs by 9.1%, 11.6%, 15.8% respectively under the Google trace with a cluster of 15000 nodes. In summary, Eirene is shown to consistently improve tail-latency performance across different cluster sizes in a scalable manner.

To understand the contributions to performance improvement by Coordinated Cold Data Migration (CCDM thereafter) and Scheduler-Aware Task Cloning (SATC thereafter) individually, we conduct experiments with the two schemes in overloaded and lightly-loaded cases. Figure 4.6 plots the P50, P75, P90 latency performance of short jobs normalized to Eagle in the overloaded case with the Eagle, CCDM, SATC, and Eirene schemes. One can see that in the overloaded case, CCDM is the sole contributor to performance improvement compared with Eagle across the 4 traces, while SATC does not result in any performance improvement. This is expected because SATC is not activated if the cluster is kept overloaded, while the high task waiting time due to overloading is leveraged by CCDM to migrate cold data for task runtime reduction.

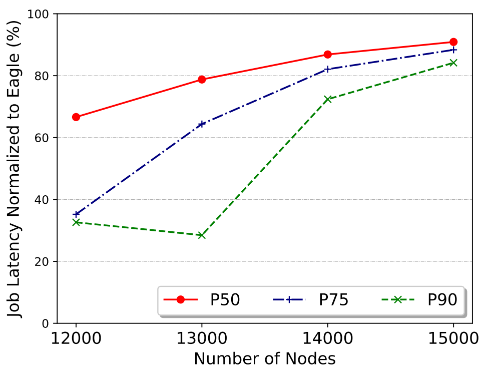
On the other side, Figure 4.7 plots the P50, P75, P90 latency performance of short jobs normalized to Eagle in the lightly-loaded case with different schemes. One can see that in this case, the performance improvement results from the SATC scheme under the Yahoo and Facebook traces. For example, SATC improves the P50, P75, P90 latency performance of short jobs of Eagle by 13.1%, 10.0%, and 8.8% respectively under the Facebook trace. This



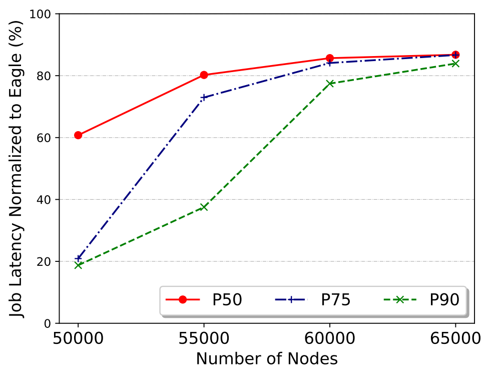
(a) Yahoo trace



(b) Cloudera trace



(c) Google trace

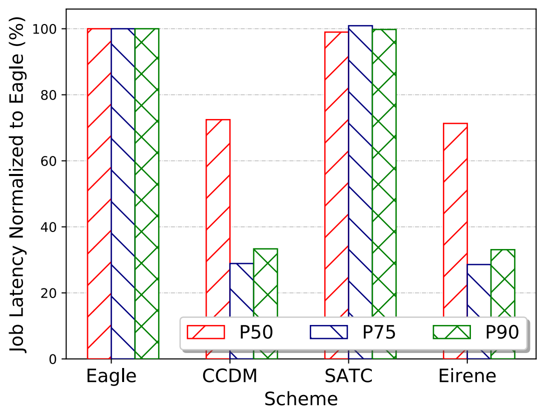


(d) Facebook trace

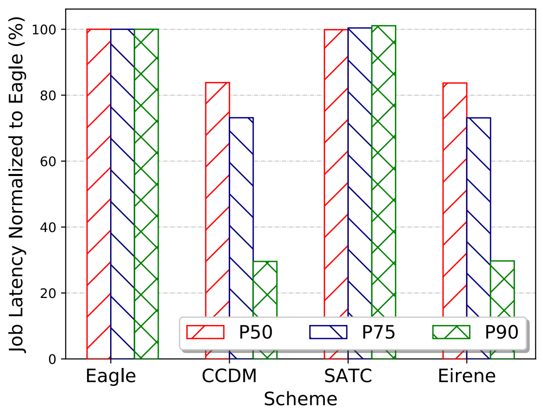
Figure 4.5: P50, P75, P90 latency performance of short jobs normalized to Eagle with different cluster sizes

is expected since abundant computing resources under light loads enables SATC to execute cloned copies of tasks of short jobs nearly at the same time the primary copies of tasks are executed. We have two additional observations. First, we can see trivial performance improvement for the Cloudera trace. It is likely because this trace has only 4.4% straggler tasks over all of the tasks for short jobs. Second, we notice there is a slight performance regression (up to -3.1% on P50 latency) under the Google trace for SATC since this trace has only 4.4% straggler jobs. In addition, short jobs in the Google trace have a relatively large task cutoff runtime (1129.5 seconds), which may cause the task cloning scheme to consume considerable computing resources.

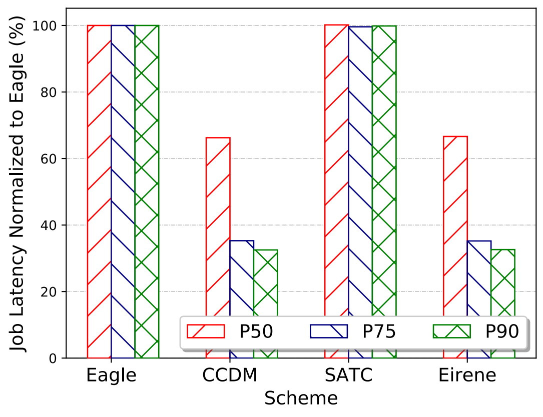
We add two counters to the simulator to keep track of the total number of short tasks benefited from CCDM and SATC respectively. The first counter is incremented for every



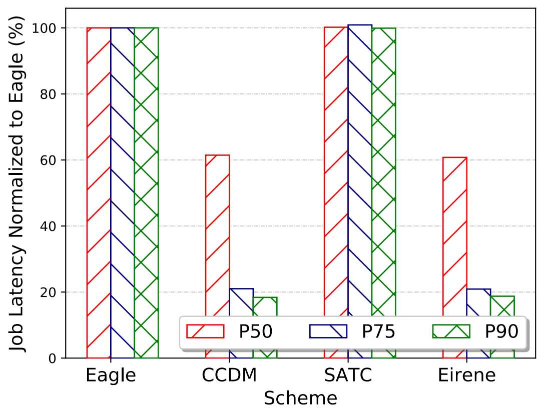
(a) Yahoo trace, 3500 nodes



(b) Cloudera trace, 13000 nodes



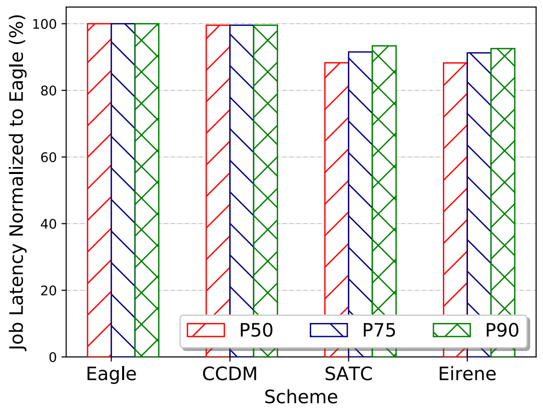
(c) Google trace, 12000 nodes



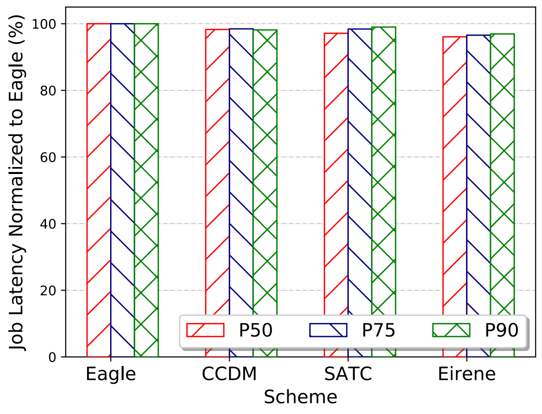
(d) Facebook trace, 50000 nodes

Figure 4.6: P50, P75, P90 latency performance of short jobs normalized to Eagle with different schemes in the overloaded case

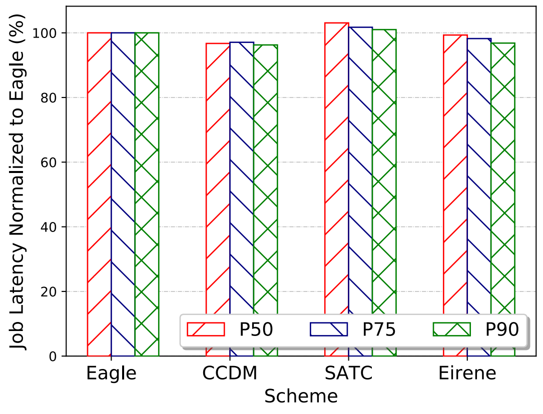
occurrence when a task reads input data from RVFS rather than the underlying distributed file system. The second counter is incremented for every occurrence when the cloned copy of a task completes earlier than the primary copy. To illustrate the trend of contributions to performance improvement by CCDM and SATC as the increase of cluster size, we plot the trend of these two counters under the Yahoo trace on a cluster of from 3000 nodes to 6000 nodes. We can see from Figure 4.8, as cluster size increases, the number of tasks benefited from CCDM decreases linearly with a steep slope and the number of tasks benefited from SATC increases linearly with a slow slope. In addition to the fact that tail latencies of short jobs are mainly affected under heavily-loaded periods rather than lightly-loaded periods, this may be another reason why SATC obtains fewer performance gains than CCDM and we see



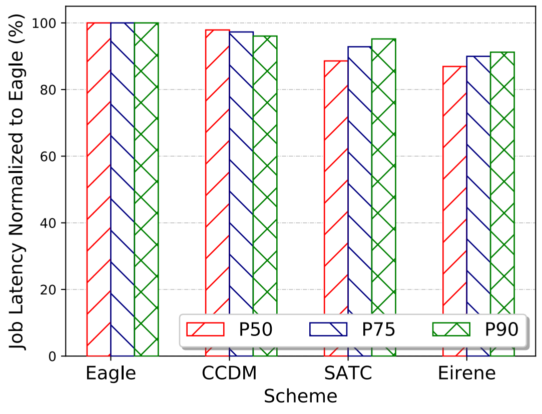
(a) Yahoo trace, 6000 nodes



(b) Cloudera trace, 18000 nodes



(c) Google trace, 20000 nodes



(d) Facebook trace, 90000 nodes

Figure 4.7: P50, P75, P90 latency performance of short jobs normalized to Eagle with different schemes in the lightly-loaded case

decreasing performance improvement as the cluster scale increases in Eirene.

Sensitivity Study

As we witness from the above section that CCDM plays a more important role than SATC in terms of the latency performance of short jobs, it is desirable to quantitatively evaluate the impact of tunable parameters of CCDM shown in Table 4.5. In the following, we vary CCDM configuration parameters and evaluate their performance impacts under the Yahoo trace as a case study.

Data migration time (DMT). Figure 4.9 plots the latency performance and the number of tasks benefited from CCDM with the varying DMTs to simulate different disk/network

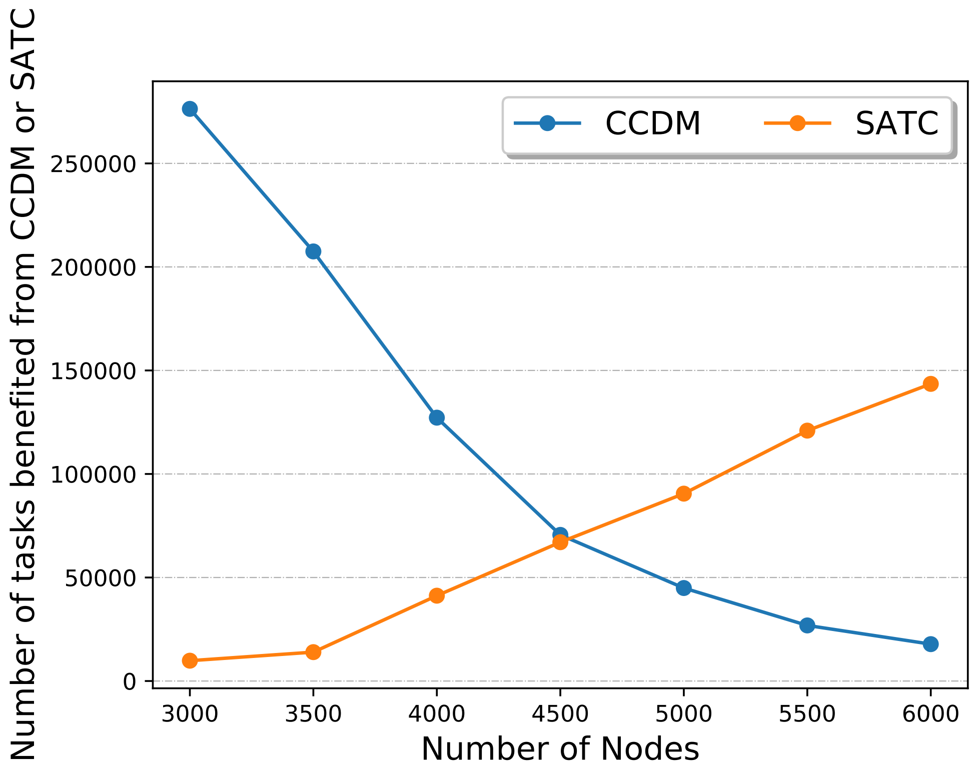
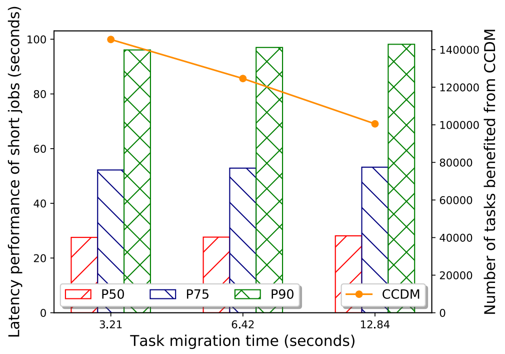


Figure 4.8: Number of short tasks benefited from CCDM or SATC under the Yahoo trace as the cluster scale increases

speeds. It is reasonable to see the number of tasks benefited from CCDM drops as the increase of DMT because a longer DMT has a higher chance to miss more opportunities for performing and benefiting from data migration. However, we find that the latency performance is almost not affected by DMT. One possible reason is that, under heavy loads, most of the short tasks are assigned and executed via Sticky Batch Probing. Since CCDM is capable of collaborating with it and migrates data of the next task when the current task is running, most performance gains may come from the reduction of runtime of such tasks, and thus the performance improvement by CCDM is agnostic to disk/network speed when task runtime of short jobs is typically much greater than DMT.

Task performance speedup ratio (TPS). Figure 4.10 plots the latency performance and the number of tasks benefited from CCDM with the varying TPSs. Although the number of tasks benefited from CCDM is decreased by 7.9% with the increase of TPS from 5% to 15%,



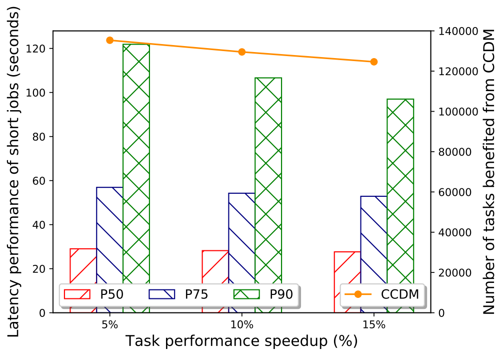


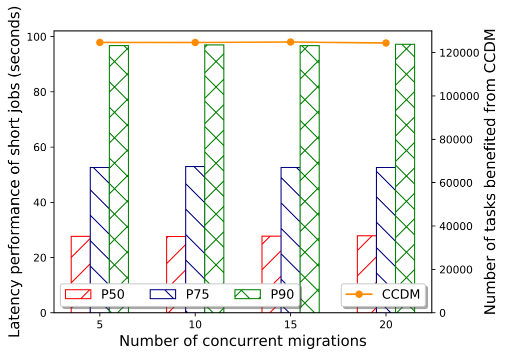
Figure 4.9: Latency performance and number of tasks benefited from CCDM with different DMTs

Figure 4.10: Latency performance and number of tasks benefited from CCDM with different TPSs

P90 latency performance of short jobs is actually improved by 20.4%. This may be because higher task performance speedup shortens task runtime and queueing time on worker nodes and thus reduces the chances of performing cold data migration and benefitting from it. In addition, the result also implies that even with only 5% task performance speedup, CCDM is still able to achieve non-negligible performance gains for short jobs.

Maximum number of concurrent migrations (CM). Figure 4.11 plots the latency performance and the number of tasks benefited from CCDM with the varying CMs. We can see that there are almost no noticeable changes to latency performance and the total number of tasks benefited from CCDM with the increase of CM from 5 to 20. This indicates that migration bandwidth is not a performance bottleneck, and we can choose a conservative value for CM.

Maximum number of migrations per probe (MPP). Figure 4.12 plots the latency performance and the number of tasks benefited from CCDM with the varying MPPs. One can see that, with the increase of MPP from 1 to 4, the number of tasks benefited from CCDM is increased by 33.8% but the P90 latency performance is improved by only 5.2%. This indicates that a conservative value for MPP like 1 or 2 is preferred to obtain sufficient performance gains and minimize disk/network bandwidth usage for data migration.



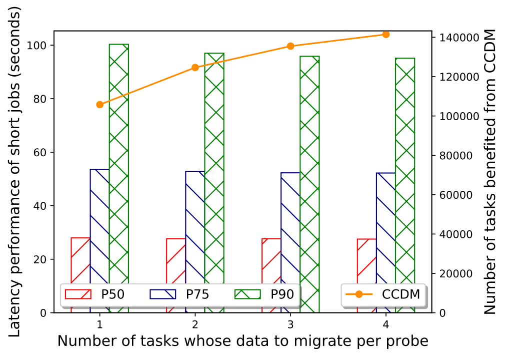


Figure 4.11: Latency performance and number of tasks benefited from CCDM with different CMs

Figure 4.12: Latency performance and number of tasks benefited from CCDM with different MPPs

4.5 Summary

In this chapter, we propose Eirene to improve tail-latency performance of short jobs hybrid job schedulers under fluctuating workloads. Eirene consists of two schemes: Coordinated Cold Data Migration and Scheduler-Aware Task Cloning. Coordinated Cold Data Migration judiciously leverages high task waiting time of short jobs during heavily-loaded periods and performs cold data migration under the coordination between distributed schedulers and worker nodes to shorten the time of reading input data at the initial stage of tasks. On the other hand, Scheduler-Aware Task Cloning proactively clones every task of short jobs during lightly-loaded periods to address the straggler problem. Experiment results from a prototype implementation of Eirene on top of a state-of-the-art hybrid job scheduler demonstrate the effectiveness and efficiency of the proposed schemes.

Chapter 5

Eunomia: A Performance-Variation-Aware Fair Job Scheduler With Placement Constraints For Heterogeneous Servers

5.1 Introduction

In this chapter, we aim to address the unfair scheduling of jobs with placement constraints in heterogeneous environments. We first emphasize a widely-known fact of server heterogeneity and performance variations in enterprise datacenters and point out the unfairness problem of latest fair schedulers for constrained jobs in heterogeneous environments with an illustrative example. Then we present Eunomia, a performance-variation-aware fair job scheduler with placement constraints for heterogeneous datacenters. Eunomia introduces *progress share fairness*, which is meant to equalize the progress share of jobs as much as possible. Progress share of a job is defined as the ratio between the accumulated progress of scheduled tasks of a job, and the maximum accumulated progress of tasks that can run in the cluster if placement constraints are removed. Finally, we evaluate an Eunomia prototype implementation and conduct quantitative evaluations via trace-driven simulations. Simulation results based on micro-benchmarks and the Google trace show that, Eunomia is able to deliver better share fairness compared with two state-of-the-art schedulers: Choosy [24] and TSF [53], without performance loss.

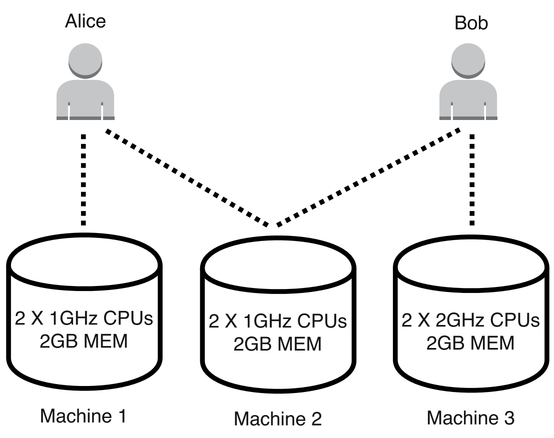
5.2 Motivations

5.2.1 Server Heterogeneity and Performance Variability

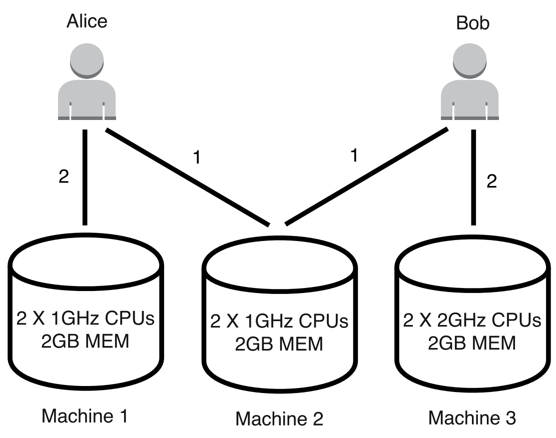
Server heterogeneity has been commonly observed and recognized in production datacenters. Usually, several generations of machines with different hardware configurations may co-exist in the same cluster [11,12]. A subset of machines may even be equipped with GPUs for special tasks like visualization, high-performance computing, or machine learning. For example, processors used in Amazon Web Services (AWS) are a variety of Intel Xeon CPUs including E5-2680v2, E5-2686v4, E5-2670, E7-8880v3, E5-2670v2, E5-2676v3, E5-2686v4 series, and processors' clock speeds range from 2.3GHz to up to 3.3GHz [82]. For another example, SSD storage provided in AWS ranges from 1 $\times$ 4GB SSD to 8 $\times$ 80GB SSDs while its HDD storage ranges from 3 $\times$ 2TB HDDs to 24 $\times$ 2TB HDDs. Various combinations of computation and I/O configurations lead to performance variability in heterogeneous clusters.

5.2.2 An Illustrative Example

Simply applying fair schedulers for constrained jobs to heterogeneous clusters cannot reach actual share fairness without taking performance variation into account. Moreover, it is not uncommon to witness cases like the scheduling systems were manipulated to gain advantages by greedy users in large companies.



(a) User/machine assumption



(b) Naive fair allocation

Figure 5.1: An example of naive fair resource sharing/scheduling

We give an example of naive fair schedulers for constrained jobs that do not consider performance variation in Figure 5.1a. As shown in this figure, we assume there are 3 machines. Machines 1 and 2 have the same hardware configuration: two 1GHz CPUs and 2GB MEM, denoted as <2 × 1GHz CPUs, 2GB MEM>. Machine 3's configuration is <2 × 2GHz CPUs, 2GB MEM>. Due to software constraints, Alice's job can only run on Machines 1 and 2, while Bob's job can only run on Machines 2 and 3, as shown in Figure 5.1a where dotted lines denote placement constraints. The tasks of the two jobs have the same demand: <1 CPU, 1GB MEM>.

As shown in Figure 5.1b, because Alice and Bob have the equal share of the cluster, Alice can run 3 tasks (2 on Machine 1 and 1 on Machine 2), while Bob also can run 3 tasks (2 on Machine 3 and 1 on Machine 2). This seems a fair resource allocation since Alice and Bob run the same number of tasks on the cluster. However, it is clear that 2 of Bob's 3 tasks are running on the faster CPU, and none of Alice's tasks are running on the faster CPU. If both jobs have the same number of tasks and it is assumed that task execution time is inversely proportional to CPU clock speed (note it is not always a realistic assumption, we just want to simplify the assumption in this example), Bob benefits more from the allocation because his job can be completed much earlier than Alice.

Server Type	Alice's Task Progress	Bob's Task Progress
<1GHz CPU, 1GB MEM>	1	1
<2GHz CPU, 2GB MEM>	2	2

Table 5.1: CPU task progress matrix as a function of different server types

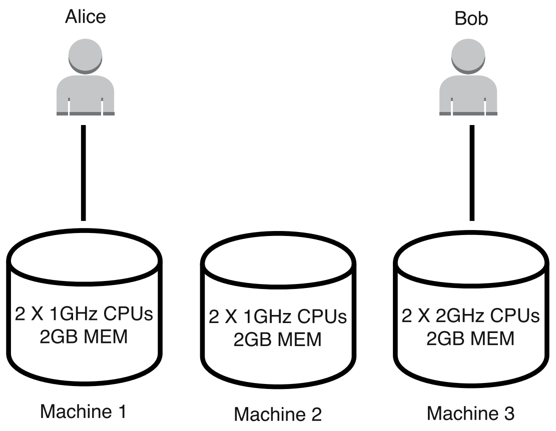
5.3 Eunomia

5.3.1 Basic Idea

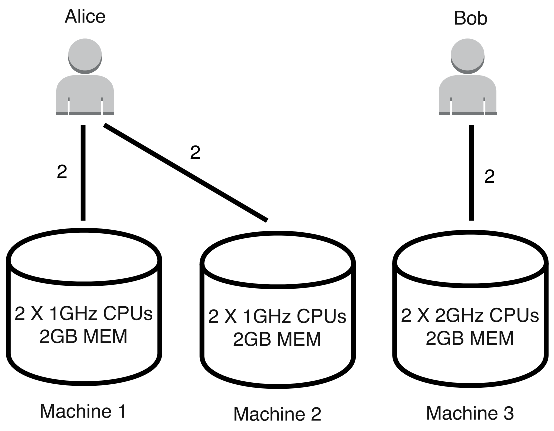
To this end, we propose *Eunomia*, a performance-variation-aware fair scheduler, which takes performance variation due to server heterogeneity into considerations and aims to equalize the progress share for each user. In Eunomia, progress share is computed as the ratio between the accumulated task progress given the current allocation and the accumulated task progress if the user can monopolize the cluster. Accumulated task progress is defined as the sum of the product of the task progress on a type of servers and the allocated number of the servers of the same type. Progress share can be treated as work slowdown of a job due to resource sharing and placement constraints. Assume task execution time on <1 $\times$ 1GHz CPU, 1GB MEM> is t, and task execution time on <1 $\times$ 2GHz CPU, 2GB MEM> is $t/2$, Table 5.1 gives the per-CPU task progress as a function of different types of nodes. If Alice is allocated with 2 <1 $\times$ 1GHz CPU, 1GB MEM> and 1 <1 $\times$ 2GHz CPU, 2GB MEM>, then the accumulated task progress of Alice's job $= 2 \times 1 + 1 \times 2 = 4$.

According to the progress share of a user, Eunomia applies the max-min fair allocation, that is, maximizes the lowest progress share first, then the second lowest, and then the third lowest, and so on. Let's continue to use the example shown in Figure 5.1 to illustrate how the allocations are undertaken in Eunomia.

As shown in Figure 5.2a, assume Machine 1 is allocated to Alice and Machine 2 is allocated to Bob respectively. Then, Alice's progress share is $(2 \times 1)/(2 \times 1 + 2 \times 1 + 2 \times 2) = 2/8$, Bob's progress share is $(2 \times 2)/(2 \times 1 + 2 \times 1 + 2 \times 2) = 4/8$. It is clear that Alice has the lowest progress share. Then 1 <1 $\times$ 1GHz CPU, 1GB MEM> on Machine 2 is further



(a) After 1st round allocation



(b) Final allocation

Figure 5.2: An example of Eunomia performance-variation-aware fair resource sharing

allocated to Alice. Then Alice's progress share is $(3 \times 1)/(2 \times 1 + 2 \times 1 + 2 \times 2) = 3/8$, which is still the lowest progress share. Then one more <1 × 1GHz CPU, 1GB MEM> on Machine 2 is allocated to Alice. Then Alice's progress share is $(4 \times 1)/(2 \times 1 + 2 \times 1 + 2 \times 2) = 4/8$, and both Alice and Bob have the same progress share now, as shown in Figure 5.2b.

5.3.2 Offline and Online Eunomia Algorithm

Figures 5.2a and 5.2b give an intuitive example to demonstrate how resources are allocated by Eunomia with progressive filling. The basic idea of "progressive filling" is to incrementally reach target fair sharing through multiple rounds, and it is widely used in various max-min fair job schedulers including Choosy [24] and TSF (Task Share Fairness) [53]. In the first round, Eunomia computes and equally raises progress shares for all the users based on their resource allocations until the maximum progress share is achieved. Then the users whose progress shares cannot be further raised are treated "inactive" users, and their resource allocations are frozen. In the second round, Eunomia continues to further recompute and equally raise progress shares of the remaining active users while keeping those of the inactive user(s) unchanged. Eunomia repeats the process round by round until all the users become inactive. The progressive filling method is considered an offline algorithm since it is impractical to

Table 5.2: Terminology in the offline Eunomia algorithm with progressive filling

Term	**Description**
N	The total number of users
M	The total number of machines in the cluster
R	The total types of resources
A	Resource amount vector, where $A_{m,r}$ is the amount of resource r on machine m
P	Normalized performance vector, where $P_{u,m}$ is the normalized performance of machine m for user u
D	Resource demand vector, where $D_{u,r}$ is the demand of resource r for user u
C	User constraint vector, where $C_{u,m} = 1$ if machine m can run tasks of user u. Otherwise, $C_{u,m} = 0$
W	User weight where W_u is the weight of user u
T	Task vector, where $T_{u,m}$ is the number of tasks of user u scheduled on machine m
H	Task vector, where $H_{u,m}$ is the number of tasks of user u scheduled on machine m as if the cluster is monopolized by user u without any constraints
S	Progress share vector, where S_u is the progress share for user u

implement due to its prohibitively high computation overheads. Table 5.2 and Algorithm 1 give terminology and a formalized algorithmic description of the offline Eunomia algorithm with progressive filling.

The offline Eunomia with progressive filling needs to recompute and raise progress shares for the users remaining active in each round. From practice, it is not necessary. Resource allocations and job scheduling only come into play when (1) a new job arrives and at least one server meets task resource demands of the job; and (2) a server completes one task and the resource is freed for re-allocation. Similar to other fair schedulers, we develop a simple online Eunomia algorithm, that is, whenever resources on a server become available, Eunomia allocates the resources to the user with the current lowest progress share, whose constraints can be met by the server. We measure the fairness and performance of the online Eunomia algorithm in the following.

Algorithm 1 Offline Eunomia Scheduler Using Progressive Filling

```
1: procedure EUNOMIA($A, P, D, C, W$)
2:     $t \leftarrow 1$                                         ▷ Current round
3:     $\mathcal{U}^t \leftarrow \{1, 2, ..., N\}$                ▷ Initialize active user set
4:     for $u \in \mathcal{U}^t$ do
5:         $T_u \leftarrow 0$                           ▷ Initialize task vector T for each user
6:     end for
7:     while true do
8:         $(T_{u,m}, S^t) \leftarrow LP(t, \mathcal{U}^t, T_u, A, P, D, C, W)$
9:         $(T_u, \mathcal{U}^{t+1}) \leftarrow SATURATED(t, S^t, \mathcal{U}^t, A, P, D, C, W)$
10:        if $\mathcal{U}^t = \emptyset$ then                        ▷ All users saturated?
11:            return $T_{u,m}$     ▷ Return number of tasks scheduled on each node for each user
12:        end if
13:    end while
14:    $t \leftarrow t+1$
15: end procedure
16: procedure SATURATED($t, S^t, \mathcal{U}^t$, A, P, D, C, W)
17:    $\mathcal{I}^t \leftarrow \emptyset$                              ▷ inactive user set after round t
18:    for $u \in \mathcal{U}^t$ do
19:        for $v \in \mathcal{U}^t \backslash \{u\}$ do
20:            $T_v \leftarrow \sum_{m=1}^{M} T_{v,m}$                         ▷ Saturate all but v
21:        end for
22:        $(T^z_{u,m}, S^z) \leftarrow LP(t, v, T_v, A, P, D, C, W)$
23:        if $S^z == S^t$ then                      ▷ If progress share cannot be increased
24:            $\mathcal{I}^t \leftarrow \mathcal{I}^t \cup \{u\}$                        ▷ User u becomes inactive
25:        end if
26:        for $v \in \mathcal{U}^t \backslash \{u\}$ do
27:            $T_v \leftarrow 0$                       ▷ Unfreeze number of tasks of active users
28:        end for
29:    end for
30:    for $u \in \mathcal{U}^t$ do
31:        $T_u \leftarrow \sum_{m=1}^{M} T_{u,m}$              ▷ Freeze number of tasks of inactive users
32:    end for
33:    return $(T^u, \mathcal{U}^t \backslash \mathcal{I}^t)$
34: end procedure
35: procedure LP($t, \mathcal{U}^t, T_u$, A, P, D, C, W)
36:    maximize $S^t$ subject to:
37:
        $$\frac{1}{W_u \sum_{m=1}^{M} H_{u,m} P_{u,m}} \sum_{m=1}^{M} T_{u,m} P_{u,m} C_{u,m} = S^t, u \in \mathcal{U}^t \quad (5.1)$$
38:
        $$\sum_{m=1}^{M} T_{u,m} C_{u,m} \geq T_u, u \notin \mathcal{U}^t \quad (5.2)$$
39:
        $$\sum_{u=1}^{N} T_{u,m} D_{u,r} \leq A_{m,r}, m \in [1, M], r \in [1, R] \quad (5.3)$$
40:    return $(\{T_{u,m}\}, S^t)$
41: end procedure
```

Node Configuration	**Type-1 nodes**	**Type-2 nodes**	**Type-3 nodes**	**Type-4 nodes**
Number of Nodes	5	5	5	5
Normalized Performance	1.0	1.5	2.0	3.0
Number of Cores	4	4	4	4
Memory (GB)	4	4	4	4

Table 5.3: Node configuration in micro-benchmark experiments

5.4 Experiment Results

5.4.1 Experimental Setup

We develop an event-driven job scheduler simulator to conduct fairness and performance evaluations of Eunomia. This simulator takes job traces as input and is able to simulate the entire process and resulting events of job schedulers, from job arrival, job queueing, dispatching tasks to nodes, receiving completion from nodes, and job departure. Currently, this simulator is able to simulate two state-of-art fair schedulers of constrained jobs: Choosy [24] and TSF [53], as well as our proposed Eunomia. It is also highly configurable, can be easily used to simulate a cluster consisting of hundreds or thousands of nodes.

5.4.2 Micro-Benchmark Experiment Results

Micro-benchmark experiments are to demonstrate how Eunomia schedules constrained jobs on heterogeneous nodes with performance variation, and compare the fairness behaviors of state-of-art fair schedulers like Choosy and TSF with the proposed Eunomia. In micro-benchmark experiments, we simulate a cluster consisting of 4-types of nodes with different normalized performance. There are 20 nodes in total, that is, 5 nodes per type. The detailed node configuration information is depicted in Table 5.3.

In the first experiment, we simulate 4 different jobs of 4 users arrive in the cluster. Each job has a different number of tasks, a different mean task execution time, and different placement constraints. The detailed job configuration information is depicted in Table 5.4.

Job configuration	**Job 1**	**Job 2**	**Job 3**	**Job 4**
Start time(in secs)	0	10	150	150
Number of tasks	1000	150	100	100
Demand of cores per task	1	1	1	1
Demand of Memory per task	1GB	1GB	1GB	1GB
Mean task execution time on Type-1 nodes(s)	23.2	18.3	21.3	55.6
Nodes meeting constraints	All types of nodes	Type-2, 3, 4 nodes	Type-3, 4 nodes	Type-4 nodes

Table 5.4: Job configuration in micro-benchmark experiment 1

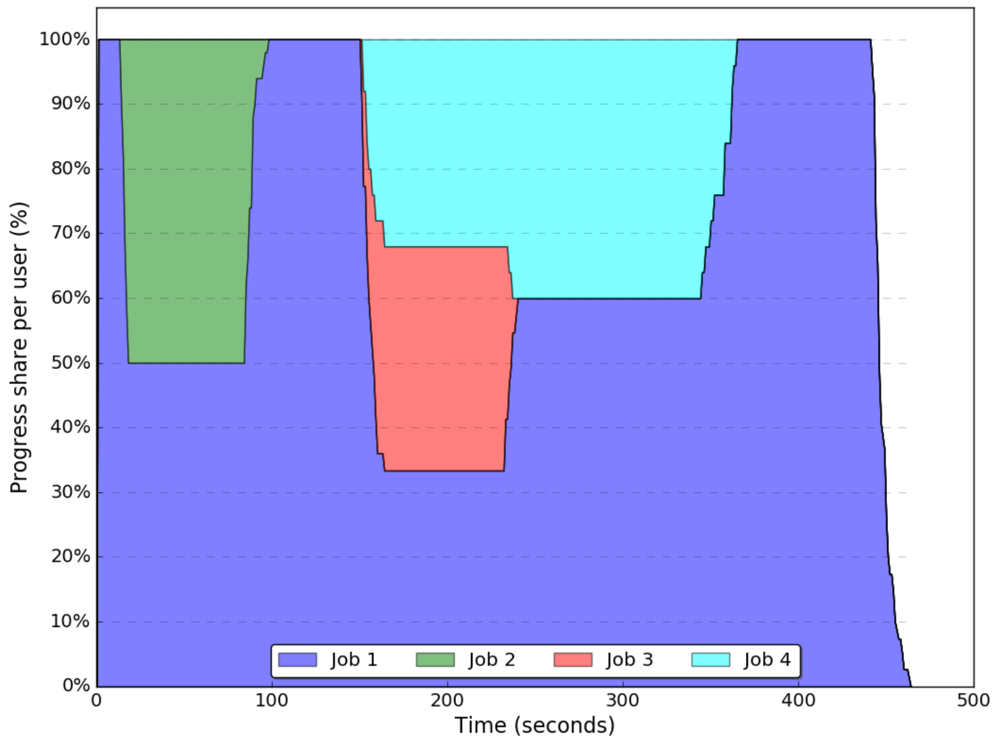


Figure 5.3: Progress shares for four jobs over time in micro-benchmark experiment 1

Figure 5.3 shows the arrival and completion of those 4 jobs. As shown in the Figure, when Job 1 arrives, the cluster is idle and then all the nodes are assigned to Job 1. After Job 2 arrives 10 seconds later, Jobs 1 and 2 reach equal progress share (50% per job) in a very short time. Because Job 2 has less number of tasks and shorter mean task execution time, it completes in Second 98. After that, Job 1 rapidly takes over the resources released by the completed Job 2. At about Second 150, Jobs 3 and 4 arrive at the same time. One can see

Job configuration	**Job 1**	**Job 2**	**Job 3**	**Job 4**
Start time(in secs)	0	0	0	0
Number of tasks	1000	1000	1000	1000
Demand of cores per task	1	1	1	1
Demand of memory per task	1GB	1GB	1GB	1GB
Mean task execution time on Type-1 nodes(s)	2	2	2	2
Nodes meeting constraints	All types of nodes	All types of nodes	All types of nodes	Type-4 nodes

Table 5.5: Job configuration in micro-benchmark experiment 2

from the figure three jobs achieve equal progress share (about 33% per job) in a very short time until Job 3 is completed in about Second 240. Then Job 4 achieves 40% progress share while Job 1 obtains 60% progress share shortly. This is because Job 4 can put its tasks on Type-4 nodes only, so the maximal progress share it can have is 40% when all Type-4 nodes are assigned to Job 4. Once Job 4 is completed in about Second 365, the rest of Job 1's tasks are assigned to all the nodes and completed in about Second 463.

In order to demonstrate the job scheduling fairness difference of different schedulers, in the second micro-benchmark evaluation, we simulate 4 jobs with nearly the same configuration arriving at the same time, except the last job explicitly places constraints on type-4 nodes. This is to compare how the fair job schedulers allocate resources if a user intentionally places job constraints on high-performing nodes and wants to take advantage of this false job constraint. The detailed job configuration information is depicted in Table 5.5.

Figures 5.4 and 5.5 shows that the arrival and completion of 4 jobs under TSF and Eunomia schedulers respectively (We omit the result of Choosy because it is very similar to TSF). Figure 5.4 clearly demonstrates for TSF Job 4 is completed much earlier than other jobs Job 4 obtains the most progress share (40%) until it is completed. This indicates that TSF fails to achieve fairness without considering performance variation due to server heterogeneity. Figure 5.5 depicts how Eunomia schedules the same 4 jobs. One can see that each job obtains equal progress share during their execution time, and all 4 jobs are completed

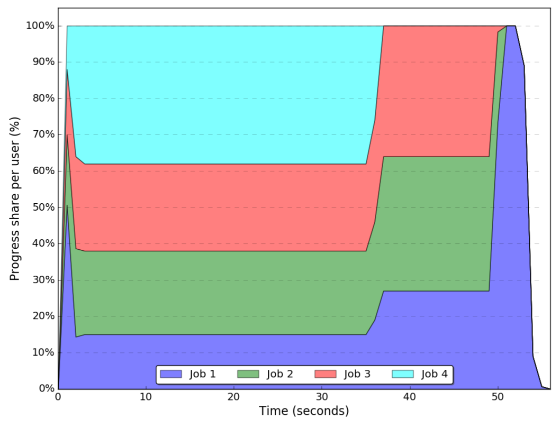


Figure 5.4: Progress shares for 4 jobs over time under the TSF scheduler in simulation 2

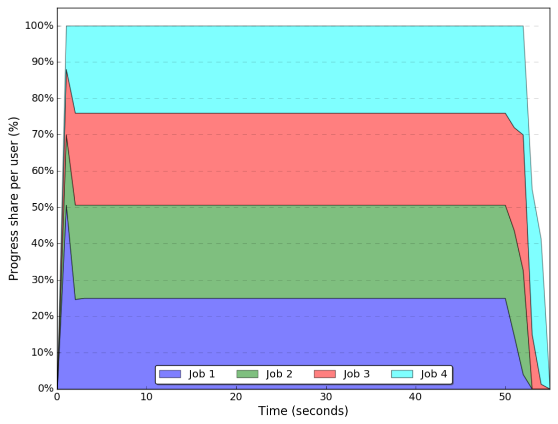
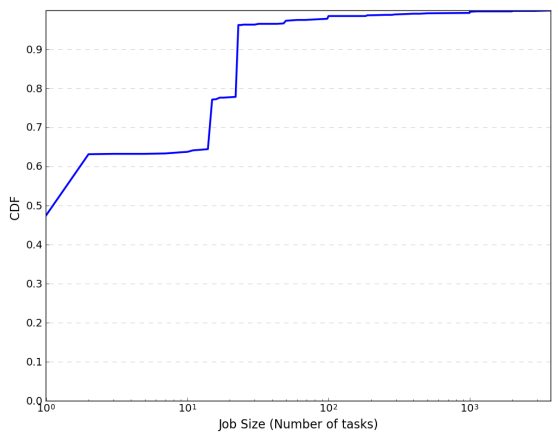
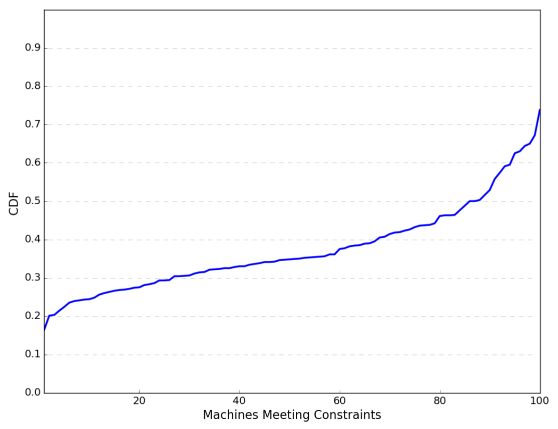


Figure 5.5: Progress shares for 4 jobs over time under the Eunomia scheduler in simulation 2



(a) Distribution of job size



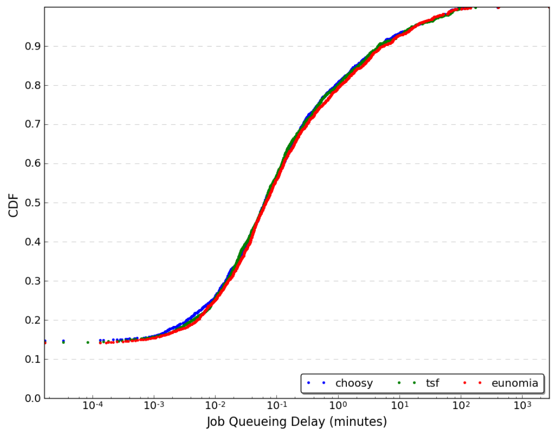
(b) Distribution of machines meeting job constraints

Figure 5.6: Distribution of job size and machines meeting job constraints in the synthesized trace

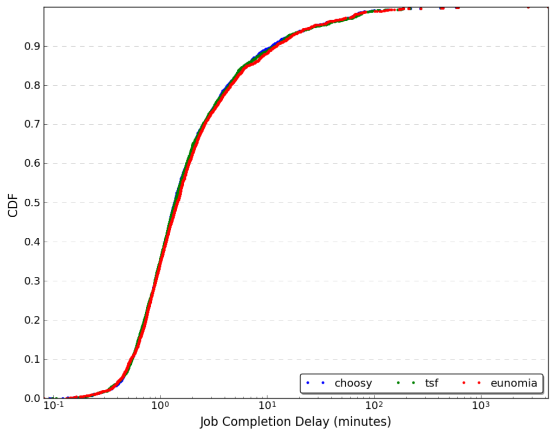
nearly at the same time. It is evident that Eunomia is able to deliver better fairness than TSF under server-heterogeneous environments and resistant to false job placement constraint requirements from greedy users.

5.4.3 Macro-benchmark experiment results

Macro-benchmark experiments are meant to validate the performance of the proposed Eunomia job schedulers. Note that the main goal of a fair job scheduler is to deliver the



(a) Distribution of job queueing delay



(b) Distribution of job completion delay

Figure 5.7: Distribution of job queueing and completion delay

guaranteed fairness instead of performance improvement, so the purpose of macro-benchmark experiments is to show the performance impact of Eunomia compared with other state-of-art fair job schedulers. To this end, we take publicly available Google cluster traces as input, synthesize and feed the workload to a simulated cluster consisting of 100 nodes. The original Google cluster traces cannot be directly used in the simulation because they are a set of sampled job events, task events, machine events, machine attributes, task constraints, etc., and synthesization work needs to be done to sample and extract the job information and compose the needed job traces including job arrival time, number of tasks, the number of CPU cores required, and the amount of memory required. The synthesized workload consists of 63,976 tasks across 2,888 jobs in 5,000 seconds. Figure 5.6a shows the distribution of job sizes. We also sample and synthesize the needed node information from Google cluster traces, for example, the number of CPU cores and the amount of memory for each node. We follow the latest way proposed by Sharma *et al.* [12] to synthesize job and node constraints. Figure 5.6b shows the distribution of nodes meeting job constraints. One can see about 18% of jobs can be run at any nodes while 50% of nodes can run 35% of jobs. In addition, we categorize 100 nodes into 10 types, and each type has different normalized performance ranging from 1.0 to 3.25, with a step of 0.25.

Figures 5.7a and 5.7b show the CDF distribution of job queueing delay and job completion delay of Choosy, TSF, and Eunomia respectively. Job queueing delay is defined as the duration between the arrival time of a job and the time when the first task of the job is scheduled. Job completion delay is defined as the duration between the arrival time of a job and the time when the last task of the job is completed. Since the cluster is idle when the simulation starts, we omit the performance results of the first 288 jobs (10% of the total jobs) and consider it as a "warm-up" period of the cluster. One can see from the figures, Eunomia achieves nearly the same job queueing delay and job completion delay as Choosy and TSF do. It implies that Eunomia does not cause any performance loss compared with state-of-art fair schedulers of constrained jobs.

5.5 Summary

In this chapter, we propose Eunomia, a performance-variation-aware fair job scheduler, to address the unfairness issue due to performance variation in heterogeneous clusters. Eunomia introduces a key metric, called "progress share", which is defined as the ratio between the accumulated task progress given the current allocation and the accumulated task progress if the user can monopolize the cluster. Eunomia aims to equalize progress share of jobs as much as possible, so as to achieve the same slowdown of jobs from different users due to resource sharing and placement constraints, regardless of performance variation. Simulation-based evaluation results show that Eunomia is able to deliver better share fairness compared with state-of-art schedulers without performance loss.

Chapter 6

Conclusions and Future Directions

In this dissertation, we address the performance and fairness issues of big data job schedulers due to the head-of-line blocking problem, straggler problem, and lack of performance variation awareness in fair scheduling as mentioned in Section 1.2. The remainder of this chapter summarizes our research contributions in Section 6.1 and points out future research directions in Section 6.2.

6.1 Contributions

Our dissertation makes the following contributions:

- We propose **Dice**, a new general performance optimization framework for hybrid job schedulers to alleviate the high job latency problem of short jobs. In Dice, we conduct trace-driven experiments to study the job latency performance behaviors of two representative hybrid job schedulers (Hawk and Eagle), and find that short jobs still encounter long latency issues due to intermittent and bursty nature of workloads. To this end, we propose Dice to address the job latency performance issue at the centralized scheduler side. Dice is composed of two simple yet effective techniques: *Elastic Sizing* and *Opportunistic Preemption*. Both Elastic Sizing and Opportunistic Preemption keep track of the task waiting times of short jobs. When the mean task waiting time of short jobs is high, Elastic Sizing dynamically and adaptively increases the short partition size to prioritize short jobs over long jobs. On the other hand, Opportunistic Preemption preempts resources from long tasks running in the general partition on demand, so as to mitigate the head-of-line blocking problem of short jobs. We enhance the two schedulers with Dice and evaluate Dice performance improvement in our prototype implementation. Experiment results show that Dice achieves 50.9%, 54.5%, and 43.5% improvement on 50th-percentile (P50), 75th-percentile (P75), and 90th-percentile (P90) job completion delays of short jobs in Hawk respectively, as well as 33.2%, 74.1%, and 85.3% improvement on those in Eagle respectively under the Google trace, at low performance costs to long jobs.

- We propose **Eirene**, another new general performance optimization framework for hybrid job schedulers to improve job latency performance of short jobs via two schemes tightly coupled with the general architecture of hybrid job schedulers. Eirene is integrated into both the distributed scheduler and worker node sides, and consists of two schemes. *Coordinated Cold Data Migration* leverages high task waiting time of short jobs under heavily-loaded periods and migrates cold data from disks to local memory for the initial phase of reading input so as to shorten task runtime and queueing time. On the other hand, *Scheduler-Aware Task Cloning* exploits spare computing resources under lightly-loaded periods and performs proactive task cloning for short jobs to mitigate the straggler problem. We implement a prototype of Eirene based on Eagle, a state-of-the-art hybrid job scheduler. Experimental results show that, under heavy loads, Eirene is able to improve P50, P75, P90 latency performance of short jobs by up to 39.2%, 79.1%, 81.3% respectively compared with Eagle under the Facebook trace with a cluster of 50000 nodes. Under moderate loads, Eirene can also improve Eagle's P50, P75, P90 latency performance of short jobs by 9.1%, 11.6%, 15.8% respectively under the Google trace with a cluster of 15000 nodes.

- We propose **Eunomia**, a performance-variation-aware fair job scheduler, to address the unfairness issue due to performance variation in heterogeneous clusters. Eunomia introduces a key metric, called *progress share*, which is defined as the ratio between the accumulated task progress given the current allocation and the accumulated task progress if the user can monopolize the cluster. Eunomia aims to equalize progress share of jobs as much as possible, so as to achieve the same slowdown of jobs from different users due to resource sharing and placement constraints, regardless of performance variation. Simulation-based evaluation results show that Eunomia is able to deliver better share fairness compared with state-of-the-art schedulers without performance loss.

6.2 Future Directions

We discuss the limitations of our proposed work and possible future research directions:

- In Dice [47], Eirene [48], and Eunomia [54], we conducted trace-driven simulation-based performance evaluations. Although the performance evaluation methodology through simulation experiments is widely accepted and yields trustworthy results, it is still very valuable to implement their prototypes on top of latest open-source job schedulers deployed in production environments like YARN [13], Mesos [14], and Kubernates [15], and evaluate the performance and fairness impacts of them with realistic applications and workloads. The experiment results from real-world implementations can help validate the effectiveness and efficiency of our proposed schemes and uncover potential issues and opportunities for further improvement;
- Dice and Eirene are proposed to address the same performance issue faced by short jobs due to the head-of-line blocking problem. In Dice, Elastic Sizing and Opportunistic Preemption are both optimization schemes at the centralized-scheduler side. In Eirene, Coordinated Cold Data Migration and Scheduler-Aware Task Cloning are optimizations

at distributed-scheduler and worker-node sides. An intuitive extension to our Dice and Eirene work is to combine these two together into a unified framework to maximize their performance potentials;

- In Dice, Opportunistic Preemption randomly selects a number of worker nodes and preempts resources of running long tasks. We feel there is much room for improvement for Opportunistic Preemption. For example, it is worthwhile to explore if there is a better way to choose long tasks for preemption by estimating the impacts of tasks to preempt on job completion delay of long jobs. For another example, it is also worthwhile to evaluate the performance impact of varying suspension and resumption delays by task preemption;

- In Eirene, the current version of Scheduler-Aware Task Cloning makes only one extra copy for every task of short jobs. It is valuable to study the impact of making two or more extra copies of every task on the latency performance of short jobs;

- Data locality plays an important role in the performance and efficiency of job scheduling. A special case of data locality is approximation analytics [83], where approximation analytics jobs need to process only a subset from combinatorially many subsets of the input data. Extending Dice and Eirene with data locality exploitation under the application workloads of approximation analytics is one interesting research direction to explore;

- Our proposed fair scheduler of Eunomia is still a centralized job scheduler, and can be easily extended to the centralized job scheduler in the general architecture of hybrid job schedulers. It will be interesting to develop a distributed version of Eunomia to ensure fair sharing among distributed schedulers in hybrid job schedulers;

- Like many conventional fair schedulers developed and deployed for private enterprise datacenters, Eunomia focuses on guaranteeing instantaneous fair sharing in heterogeneous-server environments. We face challenges when we port and extend Eunomia to datacenters for Cloud services providers where the multi-tenancy and "pay-as-you-use" model differentiates Cloud datacenters from traditional enterprise datacenters and then long-term fairness becomes more relevant. How to build a new fairness metric that takes both performance variation and monetary cost into account will be an interesting research problem to solve.

Bibliography

[1] Computing with htcondor. https://research.cs.wisc.edu/htcondor/.

[2] Slurm workload manager. http://slurm.schedmd.com/, 2019.

[3] Maui cluster scheduler. http://www.adaptivecomputing.com/products/open-source/maui/.

[4] Openlava - open source job scheduler. http://www.openlava.org/.

[5] Tom White. *Hadoop: The Definitive Guide, 4th Edition.* O'Reilly Media Inc., 2015.

[6] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *36th International Conference on Very Large Data Bases (VLDB '10)*, 2010.

[7] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source sql engine for hadoop. In *7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*, 2015.

[8] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, 2014.

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012.

[10] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[11] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC '12)*, 2012.

[12] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *ACM Symposium on Cloud Computing (SoCC '11)*, 2011.

[13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Gravesy, Jason Lowey, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *ACM Symposium on Cloud Computing (SoCC '13)*, 2013.

[14] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*, 2011.

[15] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14, 2016.

[16] Kay Ousterhout, PatrickWendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*, 2013.

[17] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *USENIX Annual Technical Conference (USENIX ATC '15)*, 2015.

[18] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *ACM Symposium on Cloud Computing (SoCC '16)*, 2016.

[19] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *37th IEEE International Conference on Distributed Computing Systems (ICDCS '17)*, 2017.

[20] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *ACM Symposium on Cloud Computing (SoCC '18)*, 2018.

[21] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[22] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013.

[23] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX conference on Networked systems design and implementation (NSDI '11)*, 2011.

[24] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *ACM European Conference on Computer Systems (EuroSys '13)*, 2013.

[25] Lustre file system. http://lustre.org/.

[26] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *1st USENIX Conference on File and Storage Technologies (FAST '02)*, 2002.

[27] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[28] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th conference on Symposium on Operating Systems Design & Implementation (OSDI '04)*, 2004.

[29] Bogdan Nicolae. Understanding vertical scalability of i/o virtualization for mapreduce workloads: Challenges and opportunities. In *2nd Workshop on Big Data Management in Clouds (BigDataCloud '13)*, 2013.

[30] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *ACM European Conference on Computer Systems (EuroSys '13)*, 2013.

[31] Hdfs architecture guide. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[32] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *ACM European Conference on Computer Systems (EuroSys '12)*, 2012.

[33] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*, 2015.

[34] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, 2009.

[35] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[36] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *ACM Symposium on Cloud Computing (SoCC '14)*, 2014.

[37] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *ACM European Conference on Computer Systems (EuroSys '16)*, 2016.

[38] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling forruntime uncertainty. In *ACM European Conference on Computer Systems (EuroSys '18)*, 2018.

[39] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *ACM European Conference on Computer Systems (EuroSys '13)*, 2013.

[40] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014.

[41] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC '15)*, 2015.

[42] Saeed Iqbal, Rinku Gupta, and Yung chin Fang. Planning considerations for job scheduling in hpc clusters. *Dell Power Solutions Magazine*, 2005.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *10th European Conference on Computer Systems (EuroSys '15)*, 2015.

[44] Zhiming Hu, Baochun Li, Zheng Qin, and Rick Siow Mong Goh. Job scheduling without prior information in big data processing systems. In *37th IEEE International Conference on Distributed Computing Systems (ICDCS '17)*, 2017.

[45] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *USENIX Annual Technical Conference (USENIX ATC '17)*, 2017.

[46] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Dont get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[47] Wei Zhou, K. Preston White, and Hongfeng Yu. Improving short job latency performance in hybrid job schedulers with dice. In *48th International Conference on Parallel Processing (ICPP '19)*, 2019.

[48] Wei Zhou, K. Preston White, and Hongfeng Yu. Eirene: Taming tail latencies of short jobs with effective cold data migration and task cloning. In *21st IEEE Intenrational Conference on Cluster Computing (CLUSTER '19)*, 2019 (Under Review).

[49] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX conference on Operating Systems Design and Implementation (OSDI '08)*, 2008.

[50] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[51] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical scheduling for diverse datacenter workloads. In *ACM Symposium on Cloud Computing (SoCC '13)*, 2013.

[52] Wei Wang, Baochun Li, and Ben Liang. Dominant resource fairness in cloud computing systems with heterogeneous servers. In *33rd Annual IEEE International Conference on Computer Communications (INFOCOM '14)*, 2014.

[53] Wei Wang, Baochun Li, Ben Liang, and Jun Li. Multi-resource fair sharing for datacenter jobs with placement constraints. In *2016 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '06)*, 2016.

[54] Wei Zhou, K. Preston White, and Hongfeng Yu. Eunomia: A performance-variation-aware fair job scheduler with placement constraints for heterogeneous datacenters. In *IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS '18)*, 2018.

[55] Shanjiang Tang, Bu-Sung Lee, Bingsheng He, and Haikun Liu. Long-term resource fairness: Towards economic fairness on pay-as-you-use computing systems. In *28th ACM International Conference on Supercomputing (ICS '14)*, 2014.

[56] Haikun Liu and Bingsheng He. Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in iaas clouds. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '14)*, 2014.

[57] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *13rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[58] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *ACM European Conference on Computer Systems (EuroSys '10)*, 2010.

[59] Chen Chen, Wei Wang, Shengkai Zhang, and Bo Li. Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees. In *IEEE International Conference on Computer Communications (INFOCOM '17)*, 2017.

[60] Mario Pastorelli, Antonio Barbuzzi, Damiano Carra, Matteo DellAmico, and Pietro Michiardi. Hfsp: Size-based scheduling for hadoop. *IEEE International Conference on Big Data (BigData '13)*, 5(1), 2013.

[61] Chen Chen, Wei Wang, and Bo Li. Performance-aware fair scheduling: Exploiting demand elasticity of data analytics jobs. In *IEEE International Conference on Computer Communications (INFOCOM '18)*, 2018.

[62] https://spark.apache.org/mllib.

[63] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '11)*, 2011.

[64] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive query processing in big data systems: A cross-industry study of mapreduce workloads. In *International Conference on Very Large Data Bases (VLDB '12)*, 2012.

[65] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.

[66] Pamela Delgado. Hawk/eagle simulator. https://github.com/epfl-labos/eagle/tree/master/simulation, 2017.

[67] Ganesh Ananthanarayanan, Christopher Douglas, Raghu Ramakrishnan, Sriram Rao, and Ion Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *ACM Symposium on Cloud Computing (SoCC '12)*, 2012.

[68] Brian Cho, Muntasir Rahman, Tej Chajed, Indranil Gupta, Cristina Abad, Nathan Roberts, and Philbert Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *ACM Symposium on Cloud Computing (SoCC '13)*, 2013.

[69] Jack Li, Calton Pu, Yuan Chen, Vanish Talwar, and Dejan Milojicic. Improving preemptive scheduling with application-transparent checkpointing in shared clusters. In *ACM/IFIP/USENIX Middleware Conference (Middleware '15)*, 2015.

[70] Yarn node labels. https://hadoop.apache.org/docs/r2.7.3/hadoop-yarn/hadoop-yarn-site/NodeLabel.html, 2019.

[71] Morris Jette and Mark Grondona. Slurm: Simple linux utility for resource management. In *ClusterWorld Conference and Expo*, 2003.

[72] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[73] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012.

[74] Nusrat Sharmin Islam, Xiaoyi Lu, Md. Wasi ur Rahman, Dipti Shankar, and Dhabaleswar K. Panda. Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '15)*, 2015.

[75] Simbarashe Dzinamarira, Florin Dinu, and T. S. Eugene Ng. Ignem: Upward migration of cold data in big data file systems. In *IEEE 38th International Conference on Distributed Computing Systems (ICDCS '18)*, 2018.

[76] Simbarashe Dzinamarira, Florin Dinu, and T. S. Eugene Ng. Dyrs: Bandwidth-aware disk-to-memory migration of cold data in big-data file systems. In *33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '19)*, 2019.

[77] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, 2012.

[78] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with skewed popularity content in mapreduce clusters. In *ACM European Conference on Computer Systems (EuroSys '11)*, 2011.

[79] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, 2015.

[80] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *ACM Symposium on Cloud Computing (SoCC '14)*, 2014.

[81] Chen Chen, Wei Wang, and Bo Li. Speculative slot reservation: Enforcing service isolation for dependent data-parallel computations. In *37th IEEE International Conference on Distributed Computing Systems (ICDCS '17)*, 2017.

[82] Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/.

[83] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *11th USENIX conference on Operating Systems Design and Implementation (OSDI '14)*, 2014.

[84] Chen Chen, Wei Wang, and Bo Li. Speculative slot reservation: Enforcing service isolation for dependent data-parallel computations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS '17)*, 2017.

[85] Wei Chen, Jia Rao, and Xiaobo Zhou. Addressing performance heterogeneity in mapreduce clusters with elastic tasks. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*, 2017.

[86] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: cost-aware container scheduling in the public cloud. In *ACM Symposium on Cloud Computing (SoCC '18)*, 2018.

[87] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2002.

[88] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *ACM Symposium on Cloud Computing (SoCC '15)*, 2015.

[89] Matteo Dell'Amico, Damiano Carra, Mario Pastorelli, and Pietro Michiardi. Revisiting size-based scheduling with estimated job sizes. In *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS '14)*, 2014.

[90] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. Medea: Scheduling of long running applications in shared production clusters. In *ACM European Conference on Computer Systems (EuroSys '18)*, 2018.

[91] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[92] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[93] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues dont matter when you can jump them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[94] Zhiming Hu, Baochun Li, Zheng Qin, and Rick Siow Mong Goh. Job scheduling without prior information in big data processing systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS '17)*, 2017.

[95] Mansour Khelghatdoust and Vincent Gramoli. Peacock: Probe-based scheduling of jobs by rotating between elastic queues. In *24th International Conference on Parallel and Distributed Computing (Euro-Par '18)*, 2018.

[96] Jinwei Liu, Haiying Shen, and Ankur Sarker. Leveraging dependency in scheduling and preemption for high throughput in data-parallel clusters. In *2018 IEEE International Conference on Cluster Computing (CLUSTER '18)*, 2018.

[97] Mario Pastorelli, Matteo Dell'Amico, and Pietro Michiardi. Os-assisted task preemption for hadoop. In *IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW '14)*, 2014.

[98] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *ACM European Conference on Computer Systems (EuroSys '18)*, 2018.

[99] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *ACM European Conference on Computer Systems (EuroSys '16)*, 2016.

[100] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A. Kozuch, and Gregory R. Ganger. Jamaisvu: Robust scheduling with auto-estimated job runtimes. Technical report, Parallel Data Laboratory, Carnegie Mellon University, 2016.

[101] Yandong Wang, Jian Tan, Weikuan Yu, Li Zhang, and Xiaoqiao Meng. Preemptive reducetask scheduling for fair and fast job completion. In *10th International Conference on Autonomic Computing (ICAC '13)*, 2013.

[102] Orcun Yildiz, Shadi Ibrahim, and Gabriel Antoniu. Enabling fast failure recovery in shared hadoop clusters: Towards failure-aware scheduling. *Future Generation Computer Systems*, 74, 2017.

[103] Aws cloud credits for research. https://aws.amazon.com/grants/.

[104] Cluster scheduler simulator overview. https://github.com/google/cluster-scheduler-simulator.

[105] Google public cluster workload traces. https://code.google.com/p/googleclusterdata/.

[106] Microsoft azure for research. https://www.microsoft.com/en-us/research/academic-program/microsoft-azure-for-research/.

[107] Quality of service (qos). https://slurm.schedmd.com/qos.html.

[108] Sparrow scheduler. https://github.com/radlab/sparrow.

[109] Yarn scheduler load simulator (sls). https://hadoop.apache.org/docs/r2.4.1/hadoop-sls/SchedulerLoadSimulator.html.