TU Delft

Delft University of Technology

Optimizing the Performance of Data Analytics Frameworks

Ghit, Bogdan

DOI
10.4233/uuid:2d9ac8e0-b922-4fcc-a33d-44a67f7bffad

Publication date
2017

Document Version
Final published version

Citation (APA)
Ghit, B. (2017). *Optimizing the Performance of Data Analytics Frameworks*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:2d9ac8e0-b922-4fcc-a33d-44a67f7bffad

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Optimizing the Performance of Data Analytics Frameworks

Optimizing the Performance of Data Analytics Frameworks

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
maandag 8 mei 2017 om 12:30 uur

door

BOGDAN IONUȚ GHIȚ

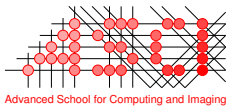
Master of Science in Parallel and Distributed Computing Systems,
Vrije Universiteit Amsterdam, the Netherlands
geboren te Boekarest, Roemenië

Dit proefschrift is goedgekeurd door de:

Promotor: Prof. dr. ir. D. H. J. Epema

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. ir. D. H. J. Epema	Technische Universiteit Delft, promotor
Onafhankelijke leden:	
Prof. dr. ir. P. F. A. Van Mieghem	Technische Universiteit Delft
Prof. dr. ir. H. E. Bal	Vrije Universiteit Amsterdam
Prof. dr. ir. A. Iosup	Vrije Universiteit Amsterdam Technische Universiteit Delft
Dr. G. Casale	Imperial College London
Prof. dr. E. Elmroth	Umeå Universitet
Dr. A. N. Tantawi	IBM Research, USA
Prof. dr. K. G. Langendoen	Technische Universiteit Delft, reservelid



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 370.

COMMIT/

This work was supported by the COMMIT project.

Cover designed by Ioana Monica Trușcă.
Translation of the Summary in Dutch by Irina van Elst.
Printed by ProefschriftMaken, Vianen.

© 2017 Bogdan Ionuț Ghiț

ISBN: 978-94-6295-640-7

Acknowledgments

A PhD challenges your resilience in research. Imagination and thirst for discovery support resilience and help you sidestep any roadblock. I am thankful to all who have been a tremendous source of inspiration during my PhD journey. Without your encouragement and dedication, I wouldn't have gotten this far.

I would like to express my gratitude to my advisor, Dick Epema, who guided me tirelessly throughout the preparation of this thesis. Dick provided excellent guidance in all aspects of research, encouraging me to never settle for less than high-quality work. He has taught me the rigor of doing scientific research, and he has helped me find the balance between theoretical and experimental work. I especially admire his uncanny ability to push the ideas one level further when a task seems to be finished, or to provide a new perspective whenever the solution is out of reach. Finally, his patience to polish the rough material with a fine-toothed comb greatly improved the quality of the explanations in this thesis.

Next, I would like to thank Alexandru Iosup, who inflicted a contagious energy in our numerous discussions about research and beyond. Alex provided a boost of confidence whenever I felt I was off track with my PhD. He has taught me how to take a step back and reflect on the big picture, not only when presenting my work, but also as a guide to open new horizons in my career.

I was fortunate to have Asser Tantawi as my mentor during the internship at IBM Research, which was definitely one of the highlights of my graduate career. Asser introduced me to the universe of cloud containers technology and provided fantastic advice on how to approach a complex problem from a mathematical perspective. I would like to thank Asser for our fruitful collaboration during my internship which paved the way to a patent application and a forthcoming article.

A few other professors have also influenced my graduate career. I would like to thank Henk Sips for his advice on career planning and for always lightening the atmosphere in our office. I am thankful to Spyros Voulgaris for being an excellent MSc supervisor and for encouraging me to start a PhD. I would also like to thank Florin Pop and Valentin Cristea for guiding my first steps in research and for giving me the opportunity to come to the Netherlands through the collaboration between Politehnica University of Bucharest and Vrije University of Amsterdam.

When I joined the PDS group at TU Delft I was truly impressed by the variety of research topics that my new colleagues were working on. Mihai Capotă got me excited about the BTWorld project, which not only led to a couple of articles, but was also an interesting use case for my own research. Tim Hegeman had a major contribution to BTWorld and deserves as much credit as Mihai for making this project real. The collaborations with Alexey Ilyushkin, Ahmed Ali-Eldin, and Lipu Fei helped me explore new topics in cloud computing. Nezih Yiğitbaşı introduced me to the Koala project and provided useful insights from his industry interactions.

In addition to direct collaborators, many other people made TU Delft an unforgettable experience. I am thankful to my good friends and colleagues Yong Guo, Jie Shen, and Jianbin Fang for sharing this adventure with me. I will always remember the Chinese New Year we celebrated together in Delft and all the wonderful stories about the Chinese culture. I have also enjoyed the witty conversations with Dimitra Gkorou, Lu Jia, Elric Milon, Riccardo Petrocco, Siqi Shen, and Otto Visser.

I would like to thank my family and friends for their never-ending love and encouragement. My parents have always supported me in all my endeavors in life and I would not be where I am today without their unwavering dedication. My fiancée, Elena, sprinkled this adventure with love, happiness, and optimism. We are about to start a lifetime journey together and I am very excited by what lies ahead for us. Finally, I am extremely grateful for the amazing memories I have made with my friends in all these years.

Contents

1 Introduction **1**
- 1.1 Datacenter Trends . . . 4
- 1.2 Data Analytics Frameworks . . . 7
- 1.3 Scheduling Architecture . . . 9
- 1.4 Problem Statement . . . 12
- 1.5 Research Methodology . . . 13
- 1.6 Thesis Outline . . . 14

2 Scheduling Multiple Frameworks in Datacenters **19**
- 2.1 Introduction . . . 19
- 2.2 Design Considerations . . . 20
- 2.3 Dynamic MapReduce Clusters . . . 22
- 2.4 Experimental Setup . . . 26
- 2.5 Experimental Evaluation . . . 28
- 2.6 Related Work . . . 32
- 2.7 Conclusion . . . 33

3 Balancing the Service Levels of Multiple Frameworks **35**
- 3.1 Introduction . . . 35
- 3.2 System Model . . . 37
- 3.3 Balanced Service Levels . . . 39
- 3.4 Experimental Setup . . . 43
- 3.5 Micro-Experiments . . . 48
- 3.6 Macro-Experiments . . . 55
- 3.7 Related Work . . . 61
- 3.8 Conclusion . . . 63

4 Size-based Resource Allocation in MapReduce Frameworks **65**
- 4.1 Introduction . . . 65
- 4.2 Problem Statement . . . 67

4.3 Size-based Scheduling . . . 68
4.4 The Tyrex Scheduler . . . 69
4.5 Experimental Setup . . . 74
4.6 Experimental Evaluation . . . 77
4.7 Related Work . . . 86
4.8 Conclusion . . . 87

5 Reducing Job Slowdown Variability for Data-Intensive Workloads 89
5.1 Introduction . . . 89
5.2 Job Slowdown Variability . . . 92
5.3 Scheduling Policies . . . 93
5.4 Experimental Setup . . . 96
5.5 Experimental Evaluation . . . 101
5.6 Related Work . . . 110
5.7 Conclusion . . . 112

6 Checkpointing In-Memory Data Analytics Applications 113
6.1 Introduction . . . 113
6.2 System Model . . . 117
6.3 Design Considerations . . . 119
6.4 Checkpointing Policies . . . 123
6.5 Experimental Setup . . . 126
6.6 Experimental Evaluation . . . 129
6.7 Related Work . . . 141
6.8 Conclusion . . . 141

7 Conclusion 143
7.1 Lessons Learned . . . 143
7.2 Future Directions . . . 145

Bibliography 145

Summary 161

Samenvatting 165

Curriculum Vitæ 169

List of Publications 171

Chapter 1

Introduction

With the capabilities of digital devices widening while their prices plunge, computer systems are facing a fundamental shift from information scarcity to data deluge. Computers, gadgets, and sensors are major data sources as they generate lots of digital information that was previously unavailable, which is increasingly used to innovate business, science, and society. The data volumes soaring all around us unlock new sources of economic value, cutting-edge findings in science, and fresh insights into human behaviour. As a consequence, analyzing large data volumes has become attractive for ever more organizations from both academia and industry.

As large companies frequently disclose that their datasets are growing vaster ever more rapidly, the amount of digital information is reckoned to increase tenfold every five years [120]. The major web services and applications that are accessed by millions of users every day and the scientific communities that seek new methods of data-driven discovery are today's richest data sources. For example, Facebook has reported storing 300 PB of data at a rate of 600 TB per day in 2014, facing a threefold increase in one year only [126]. Moreover, projects such as the Large Synoptic Survey Telescope and the Large Hadron Collider are acknowledged for generating tens of petabytes of data yearly [18, 121].

The storage size required to warehouse such large amounts of data exceeds the capabilities of single machines. Analyzing the data, to spot patterns and extract useful information, is harder still. Any given computer has a series of limitations in processor speed, memory size, and disk size, and so more and more applications are forced to scale out their computations on datacenters of thousands of machines. As the hardware is relatively cheap and easy to replicate, distributed computing has become the most successful strategy for analyzing large datasets. For example, one computer can read a 1 PB dataset from disk at a typical speed of 120 MB/s in roughly 3 months. Tackling the same problem with 1,000 machines in parallel reduces the processing time to less than 3 hours.

Most of the computations required by data analytics applications are conceptually straight-forward. Various domains such as online social networks, web search engines, and crime investigations rely on simple algorithms to process more data and provide accurate results [31]. For example, Google's search engine is partly guided by the number of clicks on a given item to determine its relevance to a search query. However, programming applications to run on thousands of machines is difficult even for experienced programmers. Many messy details occur in large-scale datacenters that tend to obscure the actual data processing: parallelizing the computation, distributing the data, and handling failures, to name just a few.

As a reaction to this complexity, Google designed MapReduce [31], a simple *data analytics framework* that captures a wide variety of large scale computations. The MapReduce model is very appealing for data processing because it scales and tolerates failures remarkably well on inexpensive off-the-shelf machines. MapReduce allows users to create acyclic data flow graphs to pass the input through a set of operators. The first MapReduce programs were executed in 2004 on a cluster of 1,800 machines, each of which had two 2 GHz Intel Xeon processors, 4 GB of memory, two 160 GB disks, and a gigabit Ethernet link [30]. MapReduce and the open-source implementation Hadoop [10] have rapidly seen significant usage. Ten years later, Yahoo! reported the largest MapReduce cluster in the world. Having more than 40,000 servers dedicated to MapReduce computations, Yahoo! stores 455 PB of data and runs 850,000 MapReduce jobs every day [13].

As has been abundantly clear, MapReduce has been poorly suited for applications that reuse a dataset across multiple operations, such as interactive queries and iterative jobs. Spark [114] was designed to support such applications, while providing similar scalability and fault-tolerance properties to MapReduce. The key contribution of Spark is that of a resilient distributed dataset which allows users to cache the dataset in memory across distributed machines and reuse it in multiple MapReduce-like computations. Spark handles failures through a lineage graph that captures the chain of operators used to transform the input dataset. As datacenters have low memory utilization and hardware is cheap, Spark has been quickly adopted by many companies. The largest Spark cluster has 8,000 machines and is owned by Tencent, a company with almost 1 billion users. Moreover, Alibaba employed Spark to process 1 PB of data in 2015.

The operation of data analytics frameworks is determined by two main properties: *resource ownership* and *job parallelism*. Resource ownership implies that data analytics frameworks assume control over subsets of the datacenter resources in order to deploy their own runtime systems and distributed filesystems. Job parallelism means that data analytics frameworks execute jobs consisting of multiple sets of parallel tasks which may have precedence constraints among them. As a consequence, allocating resources to *multiple* frameworks and scheduling multiple (sets of) jobs in *single* frameworks are complex problems in datacenters. Although both academia and industry dedicated significant ef-

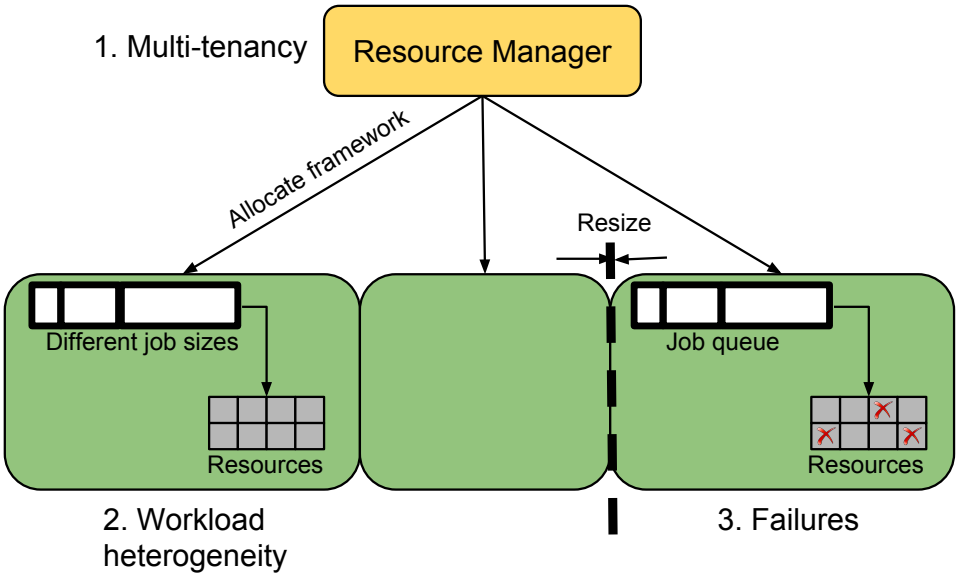


Figure 1.1: An overview of the main challenges addressed in this thesis: (1) multi-tenancy, (2) workload heterogeneity, and (3) machine failures.

fort towards improving the performance of these frameworks in datacenters, there is still a variety of opportunities for performance optimization.

An important performance metric to capture the operation of data analytics frameworks is the response time of a job, which is defined as the time between when the job arrives and when the job completes service. As users are extremely demanding and unforgiving, they may tolerate delays for long jobs, but most likely expect timely results for short jobs. To capture the delay sensitivity of jobs of different sizes, the primary metric of interest is the slowdown of a job, that is, the job response time normalized by its runtime in an empty system. We consider reducing the job slowdown a more fundamental design goal of today's computer systems than response time. Whereas response times are continuously being improved by better hardware, large job slowdowns will continue to exist in systems with contention for resources.

Scheduling has been a fundamental component in modern computer systems for minimizing the job slowdown, and the study of scheduling policies developed a rich literature of analytic results. However, the datacenter setting brings new challenges for optimizing the job performance. In Figure 1.1 we show an overview of these challenges. The first one is *multi-tenancy*: datacenters are typically shared among a large number of users, which may require isolated frameworks to process their data and to run their jobs. As workloads may vary considerably over their lifetimes, deploying them on static partitions of the system may lead to an imbalance in the service levels they receive.

A second challenge is *workload heterogeneity*: typical workloads consist of a few long jobs that consume the vast majority of resources and many short jobs that require only limited resources. As a consequence, short jobs may experience slowdowns that are an order of magnitude larger than large jobs do, while users may expect slowdowns that are more in proportion with their processing requirements (sizes).

Finally, unlike in other parallel systems, an important challenge in datacenters is *faults*: outright node failures are common in datacenters and may have a significant impact on the performance of the applications. As checkpointing to stable storage may lead to prohibitive costs, users seek to only keep their data in memory and to recover those data when needed through recomputations of work already done.

In this thesis we address these three challenges.

1.1 Datacenter Trends

The widespread adoption of cloud computing has led to massive computing infrastructures known as datacenters that power today's most popular applications such as email, social networks, and online games. During the past decade, computer systems faced a fundamental shift, with large companies developing and providing services to millions of users rather than distributing software that runs on personal computers. In this section we present an overview of the main datacenter trends.

The hardware along with the cooling systems and the power distribution equipment are located in buildings that resemble large warehouses. A 2011 survey [90] of 300 enterprises that own datacenters of a variety of sizes indicates an average datacenter size of over 15,000 square feet. The influx of new applications forced datacenter operators to give more attention to power related issues. According to the same study, 84% of firms were likely to expand their datacenters mainly to include energy efficiency and power capability. As an example of the datacenter expansion trend, we consider the growth of Facebook's datacenter in Prineville, Oregon [37]. In 2010, the 147,000 square foot facility counted roughly 30,000 servers and reached 500 million users. Within a year, Facebook doubled the datacenter size in both space and server count.

With a rich array of compute, storage, and networking off-the-shelf products, leading cloud computing vendors service millions of users daily [3]. Although companies have been reluctant to reporting concrete numbers of their server counts, recent estimations [84] based on the energy consumption in their facilities indicate that companies such as Amazon, Google, and Microsoft own roughly one million servers. To meet low-latency requirements and to cope with massive failures due to natural disasters, these servers are geographically distributed across multiple datacenters each of which hosts between 50,000 to 100,000 servers. For instance, Google operates 15 datacenters world-wide, 9 of which are located in U.S. [50].

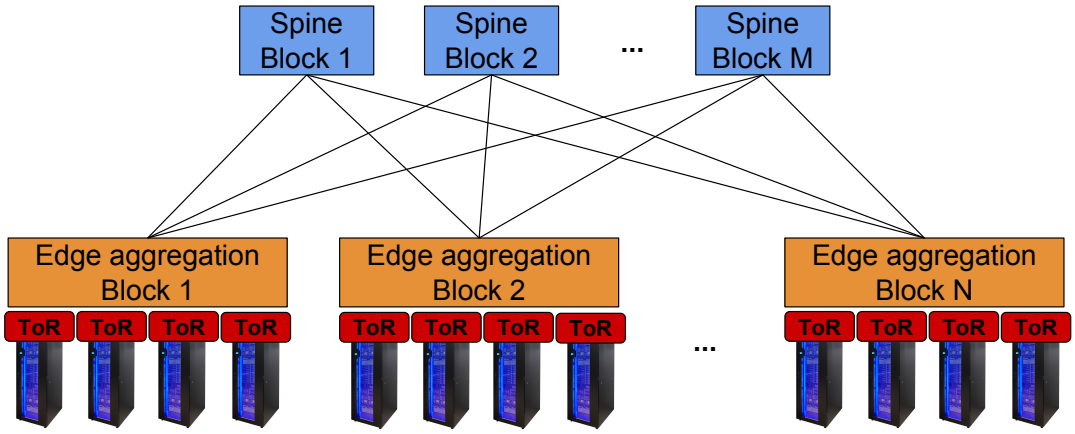


Figure 1.2: The blueprint of the network topology employed in all generations of Google's datacenters (adapted from [108]).

The ever-growing user demand calls for high availability which in turn leads to large amounts of energy waste, thus making the datacenters very expensive to operate. The power supply and the removal of waste heat for servers exceeds $12 million per month for a 100,000 server datacenter [51,53]. This means that a datacenter may consume tens of mega-watts at its peak, the same amount of power being used in roughly 10,000 homes. To reduce the hurdle of deploying and maintaining a large datacenters, there is a new trend of building micro-datacenters with thousands of low-end servers grouped in shipping containers that consume less than 500 KW. In this way, datacenters are likely to become affordable for an increasing number of organizations from research institutes to small and medium enterprises.

The datacenter sizes reported by industry organizations show a 250-fold increase from the typical size of clusters used by research institutes to perform large-scale collaborative experiments. As an example, the Distributed ASCI Supercomputer (DAS) in the Netherlands [15] is a 200-node distributed system which consists of 6 clusters with roughly 3,200 cores in total. For nearly two decades, the DAS system has been an excellent research vehicle for studying various topics in distributed systems from resource management to big data to parallel computing.

The relatively small scale of the DAS system has two main advantages. The first one is the low operation cost. As research institutes have strict budget constraints, running a few dozens of nodes per cluster makes the system cheaper to accommodate and run. The second advantage is the reproducibility of the results. The DAS system allows researchers to focus on the design, implementation, and validation of their models without worrying about the many issues that occur when running large-scale infrastructures such as performance variability and imperfect user isolation [66, 131]. In all chapters of this thesis we validate our ideas through real-world experiments on the DAS system.

Table 1.1: An overview of data analytics frameworks for different application types.

Framework	Model	Caching	Failures
MapReduce	map-reduce	no	replication
Dryad	DAG of tasks	no	replication
Pig	SQL operators	no	replication
DryadLINQ	SQL operators	no	replication
Twister	long-running tasks	yes	checkpoints
HaLoop	iterative MapReduce	yes	checkpoints
MR Online	snapshots	yes	checkpoints
Naiad	timely dataflows	yes	sync. checkpoints
Spark	RDDs	yes	lineage graph

In addition to doing experiments, we also perform large-scale simulations using workload traces from Facebook.

A modern datacenter typically hosts thousands of individual servers each of which consists of a number of multi-core processors, local memory, network interfaces, and storage. The hardware reflects the improvements made by the industry, and so they may be very different even within a single organization. Nonetheless, the architectural organization of these components has been relatively stable over the past decade [17]. Servers are co-located with disks drives and flash devices globally managed by a distributed filesystem such as Google's GFS [42] and Hadoop's HDFS [107]. These distributed filesystems achieve fault-tolerance through replication across multiple machines. Thus, not only the data remains available even after the failure of multiple servers, but the system enables higher aggregate read bandwidth by reading the same data from multiple sources.

Figure 1.2 depicts a generic Clos network architecture used in datacenters with top-of-rack (ToR) switches, aggregation blocks, and spine blocks. A rack consists of 40 to 80 servers, connected by a local 1-10 Gbps Ethernet ToR switch. A bulk of ToR switches are linked to each other through an edge aggregation switch, that is an Ethernet switch with high port counts that can span thousands of individual severs. Finally, the aggregation switches are connected to each other through a set of spine switches.

Datacenter networks are a key enabler of web services and modern storage infrastructures. With the Internet and mobile devices generating more and more content, bandwidth demands are doubling every year. Google recently reported [108] a 50-fold growth rate of the network traffic in their infrastructure since 2008. All generations of network fabrics deployed in the Google datacenters follow variants of a Clos architecture [17, 108]. In particular, the latest Jupiter switching infrastructure can deliver 1.3 Pbps of aggregate bisection bandwidth across an entire datacenter of roughly 100,000 servers, each of which being connected the network at 10 Gpbs.

With tens of servers connected to the aggregation switch by only a few links, the over-subscription factor for communication across racks is relatively high. As a consequence, running large-scale data analytics in such networks calls for network-aware placement to reduce the inter-rack communication [128, 144]. Alternatively, one may remove the inter-rack bottlenecks by investing more money in high-speed interconnects such as InfiniBand or large-scale Ethernet fabrics. While this approach incurs prohibitive costs for large-scale datacenters with thousands of servers, it is typically adopted by relatively small scale deployments such as the DAS system.

1.2 Data Analytics Frameworks

Datacenters of commodity hardware enable large-scale Internet services and scientific applications. While the computations required by these applications are conceptually straight-forward, running them efficiently in a datacenter is challenging. To harness the power of the datacenter, users need to write parallel applications with programming models that can capture a wide range of computations. As the input data of these applications is typically large, the users may need to speed-up their jobs by partitioning and distributing the data across hundreds or thousands of machines. Furthermore, the commodity hardware increases the likelihood of failures which calls for mechanisms to automatically re-execute lost work.

The code complexity required to deal with these issues calls for good programming abstractions that hide the details of parallelization, data distribution, and fault-tolerance [95]. As a consequence, there has been much research devoted to simplifying the datacenter runtime system by means of computing frameworks. Some of the widely used frameworks are MapReduce [31], Dryad [70], and Spark [145]. These frameworks greatly simplify application deployment, thus allowing non-expert users to utilize the resources of the datacenter in a transparent way.

Table 1.1 presents a summary of several data analytics frameworks tailored for a broad range of applications. Google's MapReduce [31] has been rapidly adopted by researchers and practitioners for large-scale data analytics. Hadoop [10], the open-source implementation of MapReduce, allows users to parallelize their applications in a single computation step by implementing a map function and a reduce function to transform and aggregate data, respectively. Dryad [68] sought to generalize MapReduce by extending the basic model to arbitrary directed acyclic graphs (DAG) of tasks. Similarly to MapReduce, Dryad tolerates machine and task failures through replication and re-execution, respectively. In addition, Dryad enables runtime optimizations by allowing tasks to modify the DAG at runtime based on the amount of data they read.

Programming complex data flows with low-level primitives like MapReduce and Dryad is difficult even for experienced programmers. High-level programming interfaces built

Table 1.2: The terminology used to characterize data analytics frameworks.

Term	Definition
Task	Atomic unit of computation
Phase	Set of independent tasks that may run in parallel
Shuffle	All-to-all communication between inter-dependent phases
Job	A direct acyclic graph of tasks with precedence constraints
Machine	Server with co-located processors and storage
Slot	Atomic compute unit allocated to a task on a machine
Locality	Co-locating computation with their input
Straggler	Slow task due to heterogeneity

on top of these frameworks, including Pig [91] and DryadLINQ [70], allow users to manipulate datasets in parallel with a rich set of SQL-like operators that remove the hurdle of expressing complex computations in a procedural style. These frameworks generate a low-level execution plan of MapReduce computations and optimize the data flow by pipelining data across operators in the same query, but they fall short when the data needs to be shared across queries.

Although both MapReduce and Dryad support a rich set of operators, they share data across computations through stable storage systems such as HDFS. As a consequence, these frameworks are rather inefficient for computations that require iterative data processing such as machine learning, image processing, and data clustering applications. Several frameworks, including Twister [34] and HaLoop [19] remove this shortcoming by chaining an arbitrary number of MapReduce steps in a single computation. This approach eliminates the overhead caused by re-creating new map-reduce tasks and re-loading data in each iteration. Twister keeps data in-memory by using long-running map-reduce tasks, similarly to MPI processes that remain alive during the entire lifetime of the application. HaLoop provides support for adding multiple MapReduce steps, enables caching of intermediate datasets, and achieves data locality when scheduling tasks.

The low latency requirements of streaming applications were also out of reach for MapReduce. MapReduce Online is an adaptation of MapReduce that pushes data between map and reduce tasks. Furthermore, online aggregations are performed by taking periodic snapshots using the data received so far and saving them to stable storage. Map and reduce task failures are mitigated either through checkpointing or through re-execution, respectively. Naiad [86] is able to run both iterative and streaming computations incrementally using a timely dataflow model. The framework implements a unique directed graph structure with stateful vertices that send and receive timestamped messages along directed edges, but only provides fault-tolerance by means of a synchronous checkpoint-restart mechanism.

Recently, a new use case of interactive data analytics emerged defined by users that need to run multiple ad-hoc queries on the same subset of data. In most of the previous systems, sharing the data between computations (e.g., between two MapReduce jobs) is rather inefficient as it requires persistent writes to an external storage system. Although some of the systems (e.g., Twister, Naiad) perform in-memory data-sharing, they are tailored for specific computation patterns, thus lacking any generalizations. Instead, Spark proposes a fault-tolerant abstraction called resilient distributed dataset (RDD) to efficiently share datasets across a broad range of applications. The RDDs are created through coarse-grained transformations on other RDDs. In this way, each RDD has a lineage graph which logs the information required to recompute any lost partition. In addition to being able to manipulate RDDs with a rich set of operators, users may decide on their own which RDDs to persist in memory and how to partition those RDDs.

1.3 Scheduling Architecture

From web servers to clusters to datacenters, resource management is a fundamental technique for improving the system performance. As large-scale datacenters are expensive to operate, system administrators often strive to achieve both high utilization and efficient use of resources. The ever more challenging environment consists of thousands of machines required to service both end users requests and complex infrastructure services such as monitoring, storage, and naming.

The diverse set of frameworks that emerged over the past few years reflects the continuously changing workload trends. The framework diversity hardens the scheduling problem in datacenters and calls for ever more sophisticated resource management techniques. Currently, many datacenter schedulers, including KOALA [85], Yarn [127], and Borg [131] inherited from their HPC predecessors such as Maui and Moab a monolithic architecture, with a single centralized scheduling logic for all jobs in the system. Monolithic schedulers are, however, notoriously difficult to modify in order to add new policies for specialized framework implementations.

Apparently, having a two-level scheduling architecture, with a resource manager allocating resources to multiple frameworks and allowing each framework to deal internally with their own scheduling logic is to be preferred because it provides both flexibility and parallelism. However, two-level schedulers, including Mesos [63] and Hadoop-On-Demand [75] are in practice rather limited, as they either lead to starvation of jobs or to poor utilization of resources. We provide a detailed description of the state-of-the-art datacenter schedulers in Chapters 2 and 3, which address the first challenge of this thesis.

Table 1.2 explains the terminology used to describe a data analytics job, which is the typical datacenter job model. A data analytics job shares the abstract model of a DAG of tasks, with one or more synchronization points dividing the computations into multiple

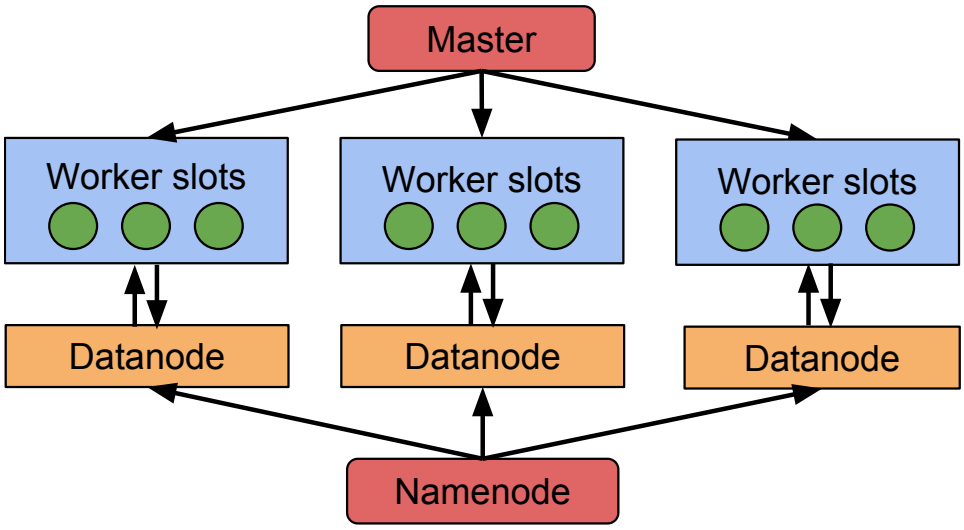


Figure 1.3: The main design components of a MapReduce framework: a centralized architecture employed by both the distributed filesystem and the runtime system.

phases. In this job model, *tasks* are atomic units of computation that compute a user supplied function for a given input partition. A *processing phase* typically consists of a set of independent tasks that execute the same function on disjoint partitions of the input dataset. A subtle point in the design of data analytics jobs is the *shuffle* that enables an all-to-all communication between phases that have precedence constraints between them. The task's input data may be read either from a distributed filesystem or may be transferred during a shuffle from other phases.

In Figure 1.3 we show the system architecture of a typical data analytics framework with its runtime system and an external storage system. Data analytics jobs run on datacenter machines with co-located processor and storage, which are conveniently sliced into multiple *compute slots* used to execute tasks. The runtime system consists of a master node that coordinates multiple workers, each of which may be configured with a fixed number of slots used to execute multiple tasks in parallel. In addition, the framework may use an external stable storage system such as the Hadoop Distributed Filesystem (HDFS) to share large datasets across different computations. HDFS employs a NameNode to manage the system metadata, and multiple DataNodes to store and provide access to the actual data. HDFS divides large files into multiple data blocks each of which is replicated three times on different DataNodes for fault-tolerance.

The fine-grained granularity of data analytics jobs exposes multiple opportunities for improving the response time of a job. An early concern was preserving the data locality by means of co-locating tasks and their input data. As memory bandwidths are at least tenfold faster than the fastest network bandwidths [128], scheduling tasks with data locality is crucial in datacenters because the network bisection bandwidth may be a bottleneck. The standard technique employed to preserve data locality is to delay the task execution for a short time until a slot that is closer to the task's input data becomes available [144].

Early reports from production systems at Facebook and Microsoft [6, 8] show evidence of straggler tasks that run eight times slower than the median task runtime, thus delaying the job completion by up to 47%. Proactive techniques that attempt to detect the root cause of stragglers [9, 140] are ineffective for many workloads in production clusters in Facebook and Microsoft. Therefore, the widely adopted technique to deal with stragglers is speculation [8, 147]. This is a reactive technique that executes speculative copies for tasks that are likely to straggle.

There is a vast literature on data analytics frameworks that tries to assess the average job runtime, but comparatively little work on optimizing the slowdown performance of time-varying workloads. The default job schedulers in many frameworks are suboptimal as they employ either a *first-in-first-out* (FIFO) scheduling discipline or an adaptation of the *processing-sharing* (PS) scheduling policy. Chapters 4 and 5, which address the second challenge of this thesis, provide more insights on the plethora of scheduling policies employed by data analytics frameworks.

A key feature driving the wide adoption of data analytics frameworks is their ability to guard against compute node failures through data replication and/or task re-scheduling. To operate at large scales, MapReduce was designed to tolerate server failures by replicating the input and output data of jobs in a distributed filesystem. Since the early days of MapReduce, three main research directions emerged towards improving the performance of these frameworks under failures. A significant amount of work has been done to alleviate hot spots, i.e., frequently accessed nodes, by adaptively replicating popular data [1,4]. Furthermore, the ability to run data analytics on inexpensive but rather transient hardware, including spot instances in clouds and peer-to-peer networks, has also led to several improvements of these frameworks [25, 79, 82, 105]. Current frameworks [7, 145], however, aim at running interactive analytics and so, they favor keeping the data in memory instead of replicating it to disk. Without replication, frameworks maintain the history of data transformations with the dependencies among them and recover lost data by re-scheduling lost computations. Chapter 6, in which we address the third challenge of this thesis, explains in detail the re-scheduling mechanism employed by data analytics frameworks to handle task failures.

1.4 Problem Statement

The research questions we address in this thesis involve both fundamental and applied aspects of the scheduling architecture in datacenters. These questions investigate the three challenges in datacenters of *multi-tenancy* (RQ1 and RQ2), *workload heterogeneity* (RQ3 and RQ4), and *failures* (RQ5) that we have identified in Figure 1.1. In particular, we address the following research questions.

[RQ1] How to build a scheduling architecture for data analytics frameworks in a datacenter? Datacenter resource managers typically employ a single scheduling algorithm for all job types in the system. This is in sharp contrast with the design of MapReduce and similar frameworks, which may require specialized scheduling policies crafted to exploit the features of the job model. Thus, isolating multiple frameworks while having the flexibility to add new specialized policies is an attractive, yet challenging target for many organizations.

[RQ2] How to achieve balanced service levels across multiple data analytics frameworks? Running multiple instances of the MapReduce framework concurrently in a datacenter enables data, failure, and version isolation. As the workloads submitted to those instances may vary considerably over their lifetimes, deploying them on static partitions of the system may lead to an imbalance in the levels of service they receive. In order to achieve performance isolation in the face of time-varying workloads, a mechanism for dynamic resource (re-)allocation to those instances is required.

[RQ3] How to design and implement a framework scheduler that overcomes the limitations of existing schedulers? With existing framework schedulers, jobs of small size with processing requirements counted in the minutes may suffer from the presence of huge jobs requiring hours or days of compute time. This may lead to a job slowdown distribution that is very variable and that is uneven across jobs of different sizes. To achieve fair performance in the face of such workloads, we need to design a scheduling policy that dynamically splits up the resources of a datacenter across specific job size ranges.

[RQ4] How to explore the design space of scheduling policies that reduce the job slowdown variability for data analytics workloads? Size-based scheduling policies reduce the job slowdown variability for sequential or rigid parallel jobs in single-server and distributed-server systems by isolating the performance of jobs with very unbalanced processing requirements. One of these policies is the well-known TAGS task assignment policy in distributed-server systems [55], which we will use to answer RQ3. Because the analytic results for such size-based policies may not hold in a datacenter setting with jobs that have elastic demands, it is not clear which is the best policy for reducing the job slowdown variability in datacenters. Thus, there is a need to bridge the gap between the operation of data analytics frameworks and former size-based policies studied in scheduling theory.

Table 1.3: An overview of the research methods employed in this thesis.

Challenge	Research question	Chapter	Queueing theory	Experiments	Simulations	Software
Multi-tenancy	RQ1	2	no	yes	no	KOALA-MR [74]
	RQ2	3	yes	yes	no	FAWKES [38]
Workload heterogeneity	RQ3	4	yes	yes	no	TYREX [124]
	RQ4	5	yes	no	yes	
Failures	RQ5	6	no	yes	yes	PANDA [93]

[RQ5] How to improve the fault-tolerance of data analytics applications without sacrificing performance? Datacenters are mainly built from commodity hardware, and so they may experience relatively high failure rates. The lack of checkpointing for in-memory data analytics frameworks is an inherent characteristic of their design, as users prefer fast in-memory computations rather than slowly writing to disk. Therefore, such frameworks rely on recomputations when they need to recover lost partitions after failures, which can be time-consuming.

1.5 Research Methodology

In this section we explain the research methods employed to answer the research questions identified in the previous section. As we show in Table 1.3, the scope of our research spans from analytical study of queueing models to experiments and simulations to software solutions. We discuss each of these aspects, in turn.

Although each chapter has its own flavour, throughout this thesis we follow the standard experimental research methodology. To this end, we always seek to identify the most general version of a new scheduling problem and to search for policies that are both theoretically grounded and practical. Moreover, we incorporate these policies in software solutions which we evaluate and compare with baselines by means of experiments on the DAS or large-scale simulations.

The research presented in this thesis aims at optimizing different aspects of the datacenter scheduling architecture, with a focus on the allocation of resources to different (sets of) jobs. Thus, this thesis is composed of five research chapters, each of which addresses a separate research question by designing and implementing a system or a mechanism, and then by analyzing (parts of) the entire design space. At a higher level, Chapters 2, 4, and 6 present systems with specific policies, whereas Chapters 3 and 5 seek for the best solution among classes of policies. The contributions of each chapter are discussed in detail in Section 1.6.

In this thesis we propose solutions that are motivated by first principles analysis of theoretical queueing models. More precisely, in Chapter 3 we describe a two-level schedul-

ing system from a queueing perspective and we derive three classes of policies for provisioning resources to multiple MapReduce frameworks. These policies take into account the dynamic load conditions as perceived by the input to the system, the utilization of the system, and the performance of the system. In Chapters 4 and 5 we go back to multi-queueing scheduling disciplines that have been studied in the context of single and distributed-server systems, and search for adaptations of those policies to datacenters.

We demonstrate the validity of the proposed solutions with comprehensive sets of experiments or simulations using realistic workloads and state-of-the-art benchmarks. All chapters of this thesis, with the exception of Chapter 5, present a thorough experimental evaluation on the DAS multicluster system. To cover a wide range of scenarios, we always seek to understand the performance of different components of the system in isolation before testing the complete system. In Chapter 5 we create an event-driven MapReduce simulator that allows us to evaluate several size-based scheduling policies at larger scales and longer durations of the workloads. We validate these MapReduce simulations through experiments on the DAS multicluster system. Chapter 6 is the only chapter in which we do both experiments on the DAS and large-scale simulations mimicking the size and operation of a Google cluster.

The work presented in this thesis has led to several software solutions that may be deployed in a datacenter computing stack. In particular, in Chapter 2 we implemented a KOALA runner for scheduling elastic MapReduce frameworks, which later evolved into a standalone research prototype called FAWKES that incorporates the resource balancing mechanism presented in Chapter 3. The study of the size-based scheduling policies in Chapters 4 and 5 materialized into a new MapReduce job scheduler called TYREX which we implemented in Hadoop. Finally, in Chapter 6 we enhanced Spark with a dynamic checkpointing system called PANDA.

Researchers have always sought to design new scheduling policies in order to improve the system-level performance in many computing systems, including operating systems, web servers, peer-to-peer systems, and databases. However, optimizing the performance of data analytics frameworks is even more challenging because datacenters are shared among many users, run heterogeneous workloads, and are prone to failures. These challenges are the root cause of the large number of disconnects between the new datacenter setting and the rich analytic results in scheduling theory. In this thesis we seek to reduce this gap between analytic results and real-world system requirements.

1.6 Thesis Outline

After attaining the necessary background on the datacenter scheduling model, we move to the core of the thesis. In this thesis, Chapters 2 and 3 address the *multi-tenancy* challenge, Chapters 4 and 5 deal with the *workload heterogeneity* challenge, and Chapter 6

investigates the *failures* challenge. More specifically, we answer the research questions through the following research contributions.

Scheduling Multiple Frameworks in Datacenters. In Chapter 2 we answer research question RQ1, by designing and implementing a two-level scheduling system for deploying elastic MapReduce frameworks in a datacenter. The architectural description includes a model for running elastic MapReduce computations and a presentation of the protocol between the resource manager and the MapReduce frameworks. The resource manager is responsible for allocating resources to each MapReduce framework, while frameworks assume ownership over those resources in order to deploy their runtime systems and their distributed filesystems. Furthermore, the resource manager supports three provisioning policies to dynamically grow and shrink the running MapReduce frameworks. We prototype the scheduling system in the KOALA grid and cloud scheduler [39, 85] and we evaluate its performance through experiments on the DAS multicluster using standard MapReduce benchmarks. This chapter is based on the following publication:

Bogdan Ghiț, Nezih Yiğitbaşı, and Dick Epema, "Resource Management for Dynamic MapReduce Clusters in Multicluster Systems", *Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS)* in conjunction with *IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

Balancing the Service Levels of Multiple Frameworks. In Chapter 3 we answer research question RQ2 by designing and implementing a resource balancing mechanism called FAWKES that attempts to balance the allocations to MapReduce instances so that they experience similar service levels. FAWKES proposes a new abstraction for deploying MapReduce instances on physical resources, the MR-cluster, which represents a set of resources that can grow and shrink. The MR-cluster has a core on which MapReduce is installed with the usual data locality assumptions but that relaxes those assumptions for nodes outside the core. FAWKES dynamically grows and shrinks the active MR-clusters based on a family of weighting policies with weights derived from monitoring their operation. We empirically evaluate FAWKES on the DAS and show that it can deliver good performance and balanced resource allocations, even when the workloads of the MR-clusters are very uneven and bursty, with workloads composed from both synthetic and real-world benchmarks. This chapter is based on the following publication:

Bogdan Ghiț, Nezih Yiğitbaşı, Alexandru Iosup, and Dick Epema, "Balanced Resource Allocations Across Multiple Dynamic MapReduce Clusters", *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

Size-based Resource Allocation in MapReduce Frameworks. In Chapter 4 we answer research question RQ3 by designing and implementing a scheduling system called TYREX that is inspired by the well-known TAGS task assignment policy in distributed-server systems. In particular, TYREX partitions the resources of a MapReduce framework, allowing any job running in any partition to read data stored on any machine, imposes runtime limits in the partitions, and successively executes parts of jobs in a work-conserving way in these partitions until they can run to completion. We develop a statistical model for dynamically setting the runtime limits that achieves near-optimal job slowdown performance. We evaluate TYREX on the DAS and show that it cuts in half the job slowdown variability while preserving the median job slowdown when compared to state-of-the-art MapReduce schedulers. This chapter is based on the following publication:

Bogdan Ghiț and Dick Epema, "Tyrex: Size-based Resource Allocation in MapReduce Frameworks", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

Reducing Job Slowdown Variability for Data-Intensive Workloads. In Chapter 5 we answer research question RQ4 by analyzing a class of scheduling policies that are rightful descendants of existing size-based scheduling disciplines in single and distributed-server systems with appropriate adaptations to data analytics frameworks and clusters. The main mechanisms employed by these policies are partitioning the resources of the datacenter, and isolating jobs with different size ranges by means of timers. We evaluate these policies with realistic simulations of representative MapReduce workloads from Facebook. Under the best of these policies, the vast majority of short jobs in MapReduce workloads experience close to ideal job slowdowns, even under high system loads while the slowdown of the very large jobs is not prohibitive. We validate our simulations by means of experiments on the DAS, and we find that the job slowdown performance results obtained with both match remarkably well. This chapter is based on the following publication:

Bogdan Ghiț and Dick Epema, "Reducing Job Slowdown Variability for Data-Intensive Workloads", *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2015.

Checkpointing In-Memory Data Analytics Applications. In Chapter 6 we answer research question RQ5 by designing and implementing PANDA, a checkpointing system tailored to the intrinsic characteristics of in-memory data analytics frameworks. In particular, PANDA employs fine-grained checkpointing at the level of task outputs and dynamically identifies tasks that are worthwhile to checkpoint rather than being recomputed. We present three policies for selecting tasks for checkpointing, derived from task properties observed in data analytics workloads. We first empirically evaluate PANDA on a multiclus-

ter system with single data analytics applications under space-correlated failures, and find that PANDA is close to the performance of a fail-free execution in unmodified Spark for a large range of concurrent failures. Then we perform simulations of complete workloads, mimicking the size and operation of a Google cluster, and show that PANDA provides significant improvements in the average job runtime for wide ranges of the failure rate and system load. This chapter is based on the following publication:

Bogdan Ghiț and Dick Epema, "Better Safe than Sorry: Grappling with Failures of In-Memory Data Analytics Frameworks", *ACM International Symposium on High Performance Parallel and Distributed Computing (HPDC)*, 2017.

Conclusions. Finally, Chapter 7 draws the main conclusions of this thesis and presents the remaining open questions.

Chapter 2

Scheduling Multiple Frameworks in Datacenters

2.1 Introduction

With the considerable growth of data-intensive applications, the MapReduce programming model has become an exponent for large-scale many-task computing applications, as it not only eases the management of *big data*, but also simplifies the programming complexity of such applications on large cluster systems. Despite the high scalability of MapReduce frameworks, isolating MapReduce workloads and their data is very attractive for many users. In this chapter, we design support for deploying multiple MapReduce frameworks within multicluster environments through extensions to our KOALA grid scheduler [85]. Furthermore, we develop a dynamic resizing mechanism for MapReduce frameworks, and we introduce three resource provisioning policies. We evaluate the performance of the system through experiments conducted on a multicluster system [15, 118] managed by KOALA.

Running multiple MapReduce frameworks concurrently within the same physical infrastructure enables *four types of isolation*. First, different (groups of) users each working with their own dataset may prefer to have their own MapReduce framework to avoid interference, or for privacy and security reasons, thus requiring *data isolation*. A second type of isolation is *failure isolation*, which hides the failures of one MapReduce framework from the applications running in other concurrent MapReduce frameworks. Third, with the multiple MapReduce frameworks approach, *version isolation*, with different versions of the MapReduce framework running simultaneously, may be achieved as well. Finally, it can enable *performance isolation* between streams of jobs with different characteristics, for instance, by having separate MapReduce frameworks for large and small jobs, or for production and experimental jobs.

Making efficient use of the resources is mandatory in a multicluster environment. Therefore, to improve resource utilization, we provide the MapReduce frameworks with a resizing mechanism. The problem of dynamically resizing MapReduce frameworks brings two challenges. First, we need to determine under which conditions the size of a MapReduce framework should be modified. When the dataset exceeds the storage capacity available on the nodes of the MapReduce framework, or the workloads are too heavy, grow decisions should be taken. On the other hand, in case of an underutilized MapReduce framework, some of its nodes may be released. Secondly, we need to address the issue of rebalancing the data when resizing a cluster. When resizing, we distinguish between *core* nodes and *transient* nodes. Both types of nodes are used to execute tasks, but only the core nodes locally store data. Using transient nodes to provision a MapReduce framework has the advantage of not having to change the distribution of the data when they are released. On the downside of this approach, data locality is broken, as all the tasks executed on transient nodes need to access non-local data.

The contributions of this chapter are as follows:

1. The design of a KOALA runner that provides isolation between the deployments of multiple MapReduce frameworks within a multicluster system.

2. The dynamic resizing mechanism for MapReduce frameworks with three distinct provisioning policies that avoids high reconfiguration costs and handles the data distribution in a reliable fashion.

3. An evaluation of the performance of KOALA with MapReduce support on a real infrastructure (the DAS multicluster system).

2.2 Design Considerations

MapReduce frameworks may be isolated in two different ways in a multicluster system, as illustrated in Figure 2.1: either *across* different physical clusters (inter-cluster isolation), or *within* single physical clusters (intra-cluster isolation). In both cases, four types of isolation can be identified: performance isolation, data isolation, failure isolation, and version isolation, which we now describe in turn.

Performance isolation. With the standard FIFO scheduling technique, MapReduce frameworks executing long running jobs may delay the execution of small jobs. To overcome this shortcoming, current frameworks typically employ the FAIR scheduler [144]. With its dynamic resource allocation scheme based on a *processor-sharing* (PS) scheduling discipline, the small jobs receive their share of resources in a short time by reducing the number of nodes occupied by the large jobs. The execution time of the large jobs is increased, as they are forced to spawn across a smaller number of nodes than the actual

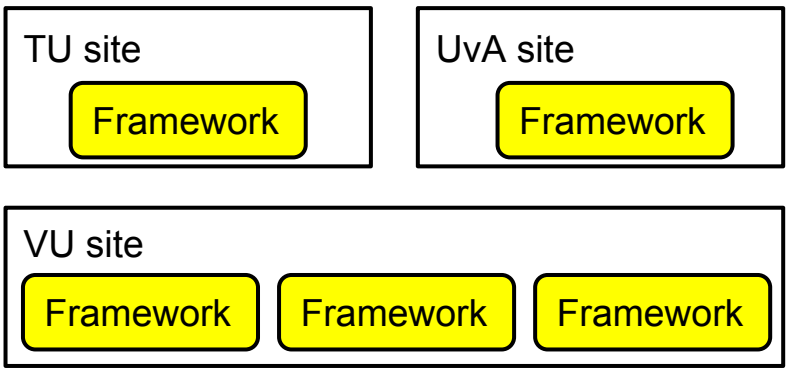


Figure 2.1: The two types of isolation enabled in our DAS multicluster system: inter-cluster isolation (two MapReduce frameworks deployed within the TU and UvA sites) and intra-cluster isolation (three MapReduce frameworks deployed within the VU site).

MapReduce framework capacity. In this way, the small jobs gain their fair share of resources without long delays, at the expense of reducing the performance of the large jobs.

The trade-off between performance and fairness in MapReduce workloads can be avoided by isolating the streams of small and large (or potentially even more classes of) jobs within separate MapReduce frameworks. Similarly, deploying multiple MapReduce frameworks allows large companies to isolate their production workloads from experimental jobs. Jobs in the development phase may need thorough testing and debugging before being launched on a production cluster. Thus, isolating them within a separate MapReduce framework first preserves the performance of the production MapReduce framework, and secondly, may reduce the debugging time for the developer.

Data isolation. Users may form groups based on the datasets they want to process, such that the jobs of a certain group are executed within a separate MapReduce framework. Therefore, several datasets may be stored in different MapReduce frameworks, while the stream of jobs is demultiplexed into multiple substreams based on the dataset they need to process. In particular, data isolation is important for systems research groups that want to analyze the performance of single applications under controlled conditions.

According to recent usage trends [2], users may request their own MapReduce frameworks to run experiments within isolated environments, or to guarantee the privacy and the security of their data. In the case of a single MapReduce framework deployment, the data is uniformly distributed across the nodes of the system, and the distributed filesystem is visible to all users. Thus, there is no privacy among the users of the system. Also, due to the lack of protection, users may intentionally or unintentionally alter the data of other users. For these reasons, there is a need to isolate the datasets within different MapReduce frameworks and to selectively allow access to read and process them.

Failure isolation. A third type of isolation is the failure isolation. The MapReduce deployments are prone to both software (implementation or configuration errors) and hard-

ware failures (server or network equipment failures). In both cases, failures of the system may cause loss of data, interruption of the running jobs, and low availability. By deploying multiple MapReduce frameworks, only the users of a specific MapReduce framework suffer the consequences of its failures.

Version isolation. Finally, with multiple MapReduce frameworks we can enable access to different versions of the MapReduce framework at the same time. This is useful when upgrades of the framework are being made, or when new implementations are being developed. Testing, debugging, and benchmarking the frameworks, while having at the same time a running stable production MapReduce framework is enabled by our approach.

2.3 Dynamic MapReduce Clusters

In this section we present our approach for achieving isolation between multiple MapReduce frameworks. First, we explain the system model, then we describe the components of the KOALA resource management system, and finally, we propose three dynamic resizing policies.

2.3.1 The Koala Grid Scheduler

KOALA is a grid resource manager developed for multicluster systems such as the DAS with the goal of designing, implementing, and analyzing scheduling strategies for various application types. The scheduler represents the core of the system and is responsible for scheduling jobs submitted by users by placing and executing them on suitable cluster sites according to its scheduling policies. Jobs are submitted to KOALA through specialized runners for certain application types (e.g., cycle scavenging jobs [112], workflows [113], and malleable applications [20]). To monitor the status of the resources, KOALA uses a processor and a network information service.

To develop a MapReduce runner for KOALA we took as a reference the design of the CS-Runner [112], which enhances KOALA with mechanisms for the efficient allocation of otherwise idle resources in a multicluster to Cycle-Scavenging (CS) applications (e.g., Parameter Sweep Applications). The CS-Runner initiates *Launchers*, which are a kind of pilot jobs, to execute the required set of parameters. The CS-Runner implements a grow-and-shrink mechanism that allows increasing or decreasing the number of Launchers when a resource offer or a resource reclaim is received from KOALA.

To schedule jobs, KOALA interfaces with the local resource managers of the clusters in the multicluster grid system. However, KOALA does not fully control the grid resources, as users may submit their jobs directly through the local resource managers deployed on each physical cluster.

2.3.2 System Model

A MapReduce framework relies on a two-layer architecture: a compute framework to facilitate an execution environment for MapReduce applications, and an underlying distributed filesystem that manages in a reliable and efficient manner large data volumes. Both layers are distributed across all nodes of a MapReduce framework, such that each node may execute tasks and also store data. In Hadoop, the open-source implementation of MapReduce, a central machine runs a master (JobTracker) that coordinates a number of worker nodes (TaskTrackers). A worker node of a MapReduce framework can be configured with multiple task slots such that each slot corresponds to a core of the processor available on the node. Based on the type of tasks to be executed, we distinguish map and reduce slots. The distributed filesystem has a similar design, with a NameNode that manages the metadata and multiple DataNodes co-located with the TaskTrackers that store the actual data.

When growing or shrinking the MapReduce framework, the two layers need to be adjusted accordingly. While the execution framework can be easily resized without significant reconfiguration costs, changing the size of the distributed filesystem is more complex, because it may require rebalancing the data. As this operation is expensive, and may have to be performed frequently, we propose a hybrid architecture for MapReduce frameworks.

In this hybrid architecture, we distinguish two types of nodes: *core* nodes and *transient* nodes. The core nodes are the nodes that are initially allocated for the MapReduce framework deployment. They are fully functional nodes that run both the TaskTracker and the DataNode, and so they are used both for their compute power to execute tasks, and for their disk capacity to store data blocks. The transient nodes are temporary nodes provisioned after the initial deployment of the MapReduce framework. They can be used as compute nodes that only run the TaskTracker, but do not store data blocks and do not run the DataNode. Their removal does not change the distribution of the data.

2.3.3 System Architecture

This section explains how we have extended the original KOALA architecture to include MapReduce support. Figure 2.2 illustrates the interaction between the existing KOALA components and the additional components that extend the scheduler with support for deploying MapReduce frameworks on a multicluster system. The new components are the following: a specific KOALA runner called MR-Runner, a specific MapReduce framework configuration module called MR-Launcher, and the global manager of all active MR-Runners called KOALA-MR.

KOALA is responsible for scheduling jobs, which in this case are complete MapReduce frameworks, received from the MR-Runners. Based on the desired size (number of nodes) of the MapReduce framework, KOALA schedules the job on the adequate physical clus-

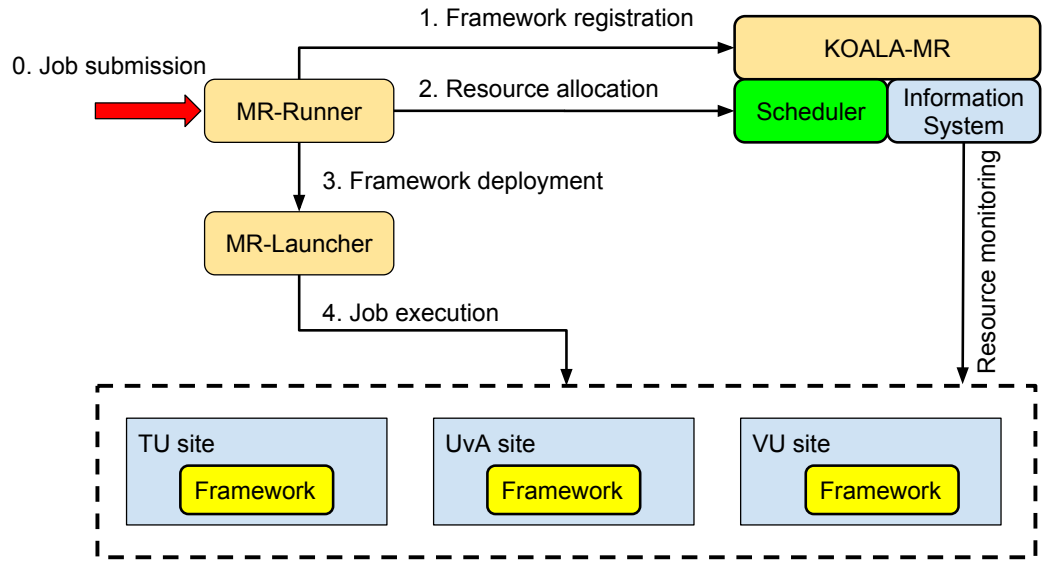


Figure 2.2: The two-level scheduling architecture employed by the KOALA grid scheduler to schedule MapReduce frameworks.

ter by applying one of its placement policies. To reduce the overhead of redistributing the data, we assume that the size of the MapReduce frameworks never decreases below the initial number of core nodes. Nevertheless, MapReduce frameworks may be resized by adding or removing transient nodes. The grow or shrink requests to the active MR-Runners are initiated by the scheduler itself, which tries to achieve fairness between multiple MapReduce frameworks.

KOALA monitors the availability of the resources through the KOALA Information System (KIS) module. When idle nodes are identified, the MR-Runners may receive grow requests. In contrast, in order to open up space for new job submissions, the scheduler may send shrink requests to the active MR-Runners.

After KOALA allocates nodes for the MapReduce framework deployment on a certain physical cluster, the MR-Runner interfaces with the local resource manager (e.g., Grid Engine) in order to proceed with the deployment of the MapReduce framework. The MR-Runner distinguishes one of the nodes as the master node, while the others are marked as slaves.

From this point, the actual configuration of the MapReduce framework is realized through the MR-Launcher. The MR-Launcher configures the core nodes in two phases: first, it mounts the distributed filesystem on the local storage of the nodes, and then it installs the compute framework for executing MapReduce applications. Furthermore, the MR-Launcher is also able to configure and remove transient nodes, or to shut down the entire MapReduce framework. In the current implementation, the MR-Launcher uses the

Hadoop daemons to configure the MapReduce framework: the NameNode and DataNodes for the HDFS, and the JobTracker and TaskTrackers for the compute framework.

Besides the scheduling and deployment functions, the MR-Runner also monitors several parameters of the MapReduce framework: the total number of (real) MapReduce jobs, the status of each such job, and the total number of map and reduce tasks. The monitoring process feeds a runner-side provisioning mechanism based on which the MR-Runner takes resizing decisions. We propose three provisioning policies, which we describe in detail in the next section.

KOALA-MR is a central entity running on the scheduler side in order to maintain the metadata of each active MapReduce framework. To submit MapReduce jobs to a MapReduce framework scheduled through KOALA or to manipulate the data within the distributed filesystem, the user needs access to the corresponding metadata: the unique cluster identifier, the location of the configuration files, and the IP address of the master node. All the commands to submit MapReduce jobs or to access the distributed filesystem are executed on the master node of the MapReduce framework.

2.3.4 Resizing Mechanism

KOALA enables a two-level scheduling architecture. On the scheduler side, KOALA allocates resources for the MapReduce framework deployments based on a fair-share policy, such as the Equipartition-All or Equipartition-PerSite [112]. By monitoring the multicluster system utilization, the scheduler periodically offers additional nodes to the MR-Runners through grow requests, or reclaims previously provisioned nodes through shrink requests.

We define the *resizing ratio* K_r between the number of running tasks (map and reduce tasks) and the number of available slots (map and reduce slots) in the MapReduce framework. The resizing mechanism employs a *grow-and-shrink* policy called GS that dynamically tunes the value K_r between a minimum and a maximum threshold by accepting grow and shrink requests from the KOALA grid scheduler. To this end, the user sets two constants of the MR-Runner, the *growing size* δ^+ and the *shrinking size* δ^-. These constants represent the number of nodes the MR-Runner adds or removes whenever it receives a grow or shrink request. GS scales the framework with transient nodes which may frequently join or leave the system, and so they do not contribute to the storage layer.

We compare GS with two basic policies, which accept every resource offer, and shrink only after the workload execution is completed. The *greedy-growth* (GG) policy enables the MR-Runner to accept every resource offer regardless of the utilization of the MapReduce framework and to ignore all shrink requests from KOALA. As a consequence, the MapReduce framework may only grow in size, and it shrinks after the workload is

Table 2.1: The node configuration in the DAS multicluster system.

Processor	Dual quad-core Intel E5620
Memory	24 GB RAM
Physical Disk	2 ATA OCZ Z-Drive R2 with 2 TB (RAID 0)
Network	10 Gbps InfiniBand
Operating system	Linux CentOS-6
JVM	jdk1.6.0_27
MapReduce framework	Hadoop 0.20.203

finished. Similarly to the GS policy, the provisioning is supported by transient nodes which do not contribute to the storage layer.

The *greedy-grow-with-data* (GGD) policy makes the MapReduce framework grow in size every time a resource offer is received like in our GG policy. Unlike GG, the GGD policy is based on provisioning with core nodes instead of transient nodes. When a resource offer is received, the provisioned nodes are configured as core nodes, running both the TaskTracker and the DataNode. As a consequence, to avoid data redistribution, all shrink requests are declined.

2.4 Experimental Setup

This section presents the description of the DAS multicluster system, the Hadoop configuration parameters, and the workloads we generate for our experiments.

2.4.1 System Configuration

The infrastructure that supported our experiments is a wide-area computer system dedicated to research in parallel and distributed computing. The Distributed ASCI Supercomputer (DAS), currently in its fourth generation, consists of six clusters distributed in institutes and organizations across the Netherlands. As shown in Table 2.1, the compute nodes are equipped with dual-quad-core processors at 2.4 GHz, 24 GB memory, and a local storage of 2 TB. The networks available on DAS are Ethernet at 1 Gbps and the high-speed QDR InfiniBand at 10 Gbps. The Grid Engine is configured on each cluster as the local resource manager.

In order to schedule and execute jobs, we deploy KOALA as a meta-scheduler that interfaces with the local schedulers on each cluster. The MR-Runner is implemented in Java and currently configures Hadoop frameworks (version 0.20.203). The actual MapReduce framework configuration is realized through bash scripts which are executed within Java processes.

Table 2.2: The workload employed in our experiments consists of nine job types.

Job type	Input size (GB)	Block size (MB)	Maps
0	100	128	800
1	50	128	400
2	40	128	320
3	40	64	640
4	20	64	320
5	10	64	160
6	5	64	80
7	2.5	64	40
8	1	64	16

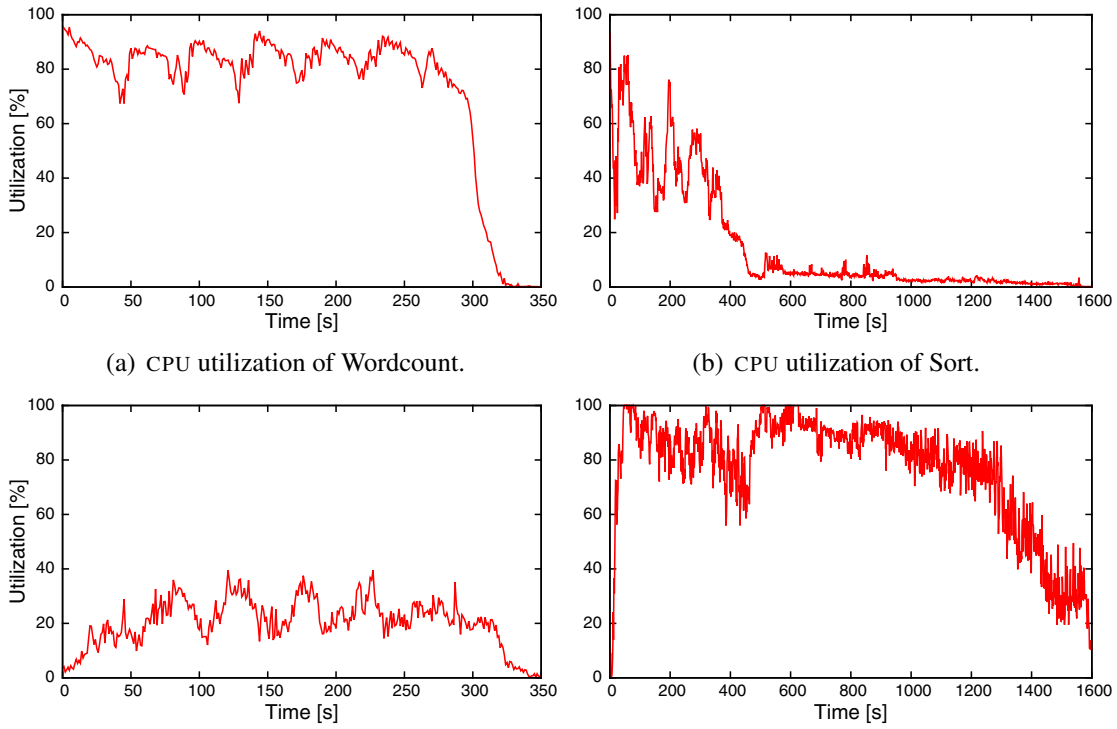
We configure the HDFS on a virtual disk device (with RAID 0 software) that runs over two physical devices, with 2 TB storage in total. The data is stored in the HDFS in blocks of 64 MB or 128 MB with a default replication factor of 3. With 16 logical cores per node enabled through hyperthreading, we configure the TaskTrackers with 6 up to 8 map slots and 2 reduce slots, respectively. To avoid issues such as network saturation due to the limited bandwidth, we setup Hadoop on the InfiniBand network.

2.4.2 MapReduce Workloads

The Wordcount and Sort applications are two common MapReduce applications included in the Hadoop distribution that are used as MapReduce benchmarks in Hibench [64]. Wordcount counts the number of occurrences of each word in a given set of input files. The map function simply emits key-value pairs for each word, while the actual counting is performed by the reducers. Sort transforms the input data from one representation to another. With both map and reduce functions implemented as identity functions, which do not modify the input data, the sort operation is performed by the MapReduce framework itself during the shuffling phase. As the framework guarantees that the intermediate key/value pairs are processed in increasing key order, the output generated is sorted.

Both small and large jobs are popular in MapReduce clusters. According to [22], 98% of the jobs at Facebook process 6.9 MB of data in less than a minute. On the other hand, Google reported in 2004 MapReduce computations that process terabytes of data on thousands of machines [31].

Based on the input size to process, we define nine types of MapReduce jobs with the characteristics given in Table 2.2. For each job, the number of reducers is set between 5% and 25% of the number of map tasks (same setting as in the workloads used in [144]). In addition, for the large jobs (more than 160 map tasks) we also use the common rule



(a) CPU utilization of Wordcount.

(b) CPU utilization of Sort.

(c) Disk utilization of Wordcount.

(d) Disk utilization of Sort.

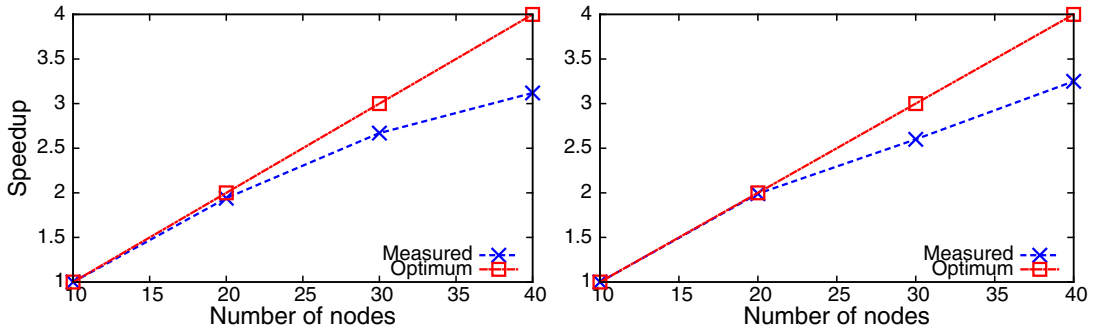
Figure 2.3: The CPU and disk utilization for the Wordcount and Sort applications on MapReduce frameworks deployed with only core nodes.

of thumb for setting the number of reducers: 90% or 180% of the number of available reduce slots in the MapReduce framework [134].

In Section 2.5 we will perform four experiments. For determining the CPU and disk utilization, we generate 100 GB input data using the RandomTextWriter and RandomWriter programs included in the Hadoop distribution. With the data block size set to 128 MB, a Wordcount or a Sort MapReduce job running on the given dataset launches 800 map tasks. The same configuration is used to determine the speedup of the applications. To evaluate the performance of transient nodes, we generate datasets of 40 GB for Wordcount and 50 GB for Sort, with the data block size set to 128 MB. Finally, for the performance evaluation of the resizing policies, we generate a stream of 50 MapReduce jobs processing datasets from 1 GB to 40 GB split into data blocks of 64 MB. We employ an exponential inter-arrival pattern with a mean value of 30 seconds.

2.5 Experimental Evaluation

In our performance evaluation we investigate the CPU and disk utilization for each of the two applications, we determine the impact on the job response time of the numbers of



(a) Speedup of Wordcount.

(b) Speedup of Sort.

Figure 2.4: The speedup for the Wordcount and Sort applications on MapReduce frameworks deployed with only core nodes.

core and transient nodes in a MapReduce framework, and we run benchmarks to compare the resizing policies.

2.5.1 CPU and Disk Utilization

First, we seek to understand the characteristics of our workloads, and to this end, we monitor the resource utilization when running them. Thus, we run the Wordcount and Sort applications on the 100 GB dataset (job type 0) and we gather CPU and disk utilization statistics from every node of the MapReduce framework at 1-second intervals. The Linux tools we use for monitoring are *top* and *netstat*. The graphs in Figure 2.3 show the CPU and the disk utilizations as average values on all nodes of the MapReduce framework. The disk utilization of a node is the average of the values measured on the two physical hard disks.

The CPU and disk utilizations illustrated in Figure 2.3 show that the Wordcount application is CPU-bound, and that the Sort application is IO-bound. For the Wordcount application we observe a long map phase with high CPU utilization, followed by a short reduce phase which has a low CPU utilization. The disk utilization is below 40% during the entire runtime of the application. On the other hand, the figures for the Sort application show that it has a short map phase during which the CPU utilization oscillates between 40% and 60%, followed by a long reduce phase which is highly disk intensive and with very low CPU utilization.

2.5.2 Performance of the Transient Nodes

First, we assess the impact of the transient nodes on the execution time of single applications using static MapReduce frameworks, without any resizing mechanism. We set up MapReduce frameworks of 10 up to 40 nodes with the large dataset as input data (job

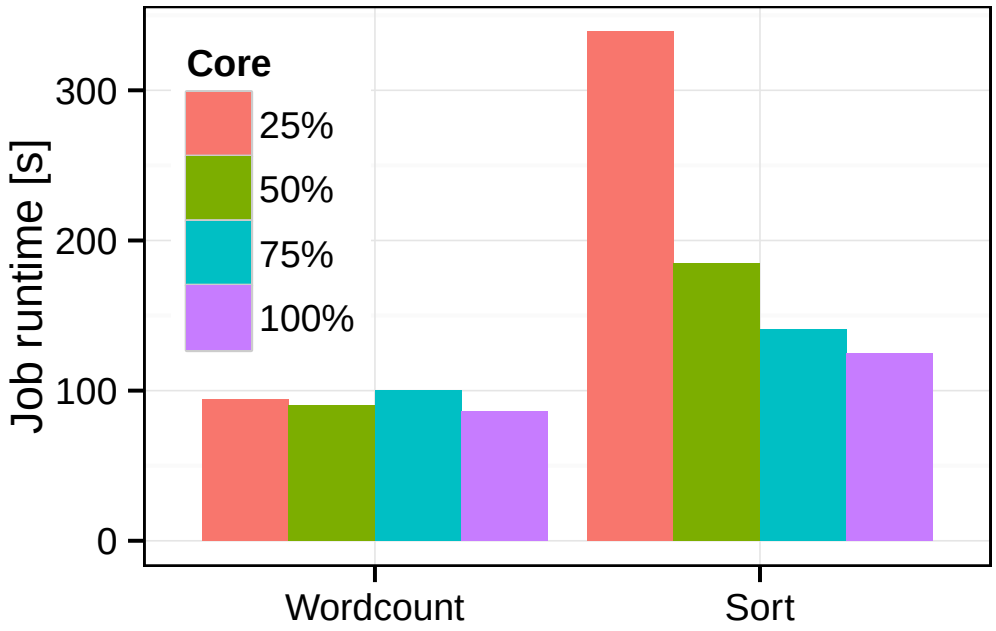


Figure 2.5: The runtime versus the fraction of core nodes for single applications on a MapReduce framework of 40 nodes in total.

type 0 for Wordcount and Sort). The Sort application launches 144 reduce tasks, representing 180% of the available reduce slots on the MapReduce framework with 40 nodes ($1.8 \times 2 \times 40$); for Wordcount we use a single reduce task. Figure 2.4 shows that the speedup for both applications on MapReduce frameworks with only core nodes is close to linear; the speedup is defined here relative to a MapReduce framework with 10 core nodes. Each data point in Figure 2.4 was obtained by averaging the measurements over 3 runs.

Secondly, we run each application on MapReduce frameworks with a total of 40 nodes with a variable number of transient nodes that do not contain data. Figure 2.5 shows that the Wordcount job (type 2 from Table 2.2) has similar runtimes no matter how many transient nodes the MapReduce framework has (from 0 up to 30). Nonetheless, the Sort job (type 1 from Table 2.2) shows a significant performance degradation when the number of transient nodes increases: the runtime for Sort doubles when the number of transient nodes increases from 0 to 30. The reason why Wordcount scales better on transient nodes than Sort is the (much) smaller amount of output data it generates. While Wordcount generates less than 20 KB for the input data of 40 GB, in the case of Sort, the size of the output data equals the size of the input data, which is 50 GB.

We will now explain why the Sort application performs poorly on MapReduce frameworks with a large number of transient nodes. To this end, we consider a MapReduce

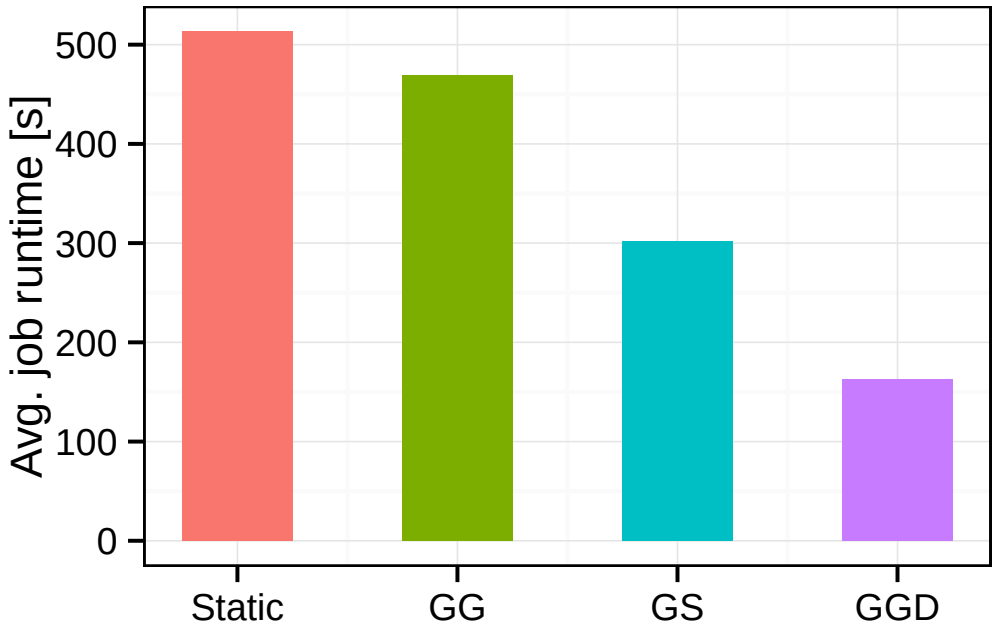


Figure 2.6: The average job runtime for an online workload of 50 jobs on a dynamic MapReduce framework of 20 core nodes.

framework with n_c core nodes and n_t transient nodes. At the end of the reduce phase, all $n_c + n_t$ nodes perform write requests on the local storages of the core nodes. The amount of output data generated by the transient nodes is $D_t = (n_t \times D_{out})/(n_c + n_t)$ Gb, where D_{out} represents the size of the entire output data for Sort. This data is transferred across the network from the transient nodes to the local storages of the core nodes where the HDFS is mounted. The time to transfer D_t is $T_t = D_t/bw$, where bw represents the bandwidth available on the DAS (10 Gbps on InfiniBand). The time spent writing the data on the disks is $T_w = D_t/(2 \times n_c \times w)$ seconds, where w denotes the write speed on the local disks and $2 \times n_c$ is the total number of disks available on the core nodes. For instance, a MapReduce framework with $n_c = 10$ core nodes and $n_t = 30$ transient nodes generates $D_t = 922$ Gb. With $bw = 10$ Gbps we obtain $T_t = 92$ seconds. According to the specification of the devices, the read/write speed on the ATA OCZ disks is 1 Gbps and 900 Mbps, respectively. Therefore, T_w is 51 seconds. Intuitively, having a small number of core nodes increases the contention on the physical disks on which the HDFS is mounted. Indeed, as we can see in Figure 2.5, with 25% core nodes of the MapReduce framework capacity, the runtime of the workload is 2.7 higher than the ideal case of a MapReduce framework with 40 core nodes.

2.5.3 Performance of the Resizing Mechanism

To assess the performance of the MapReduce cluster resizing mechanism, we run four benchmarks that execute successively a stream of jobs on a static MapReduce framework, and on a dynamic MapReduce framework with one of our three provisioning policies enabled (GS, GG, or GGD). The MapReduce framework consists of 20 core nodes, and we assume a pool of 20 transient nodes to be available for provisioning during our experiments. We create a workload with 50 jobs each of which is an instance of a job type given in Table 2.2. For simplicity, we instrument the MR-Runner to initiate resource offers every T seconds, where T is set either to 30 seconds for GS, or to 120 seconds for GG and GGD. For the last two policies, the size of the resource offer is 2 nodes. For GS, the MapReduce framework accepts a resource offer when K_r is greater than 1.5, and releases nodes when K_r is below 0.25. When the MapReduce framework frequently changes its size by adding or releasing a large number of nodes, the reconfiguration overhead may impact the performance of the running jobs. Therefore, we set δ^+ and δ^- to 5 and 2, respectively.

As can be seen in Figure 2.6, GG shows a small improvement over the static approach, while GS reduces the runtime by a factor of 1.7. Intuitively, GG is not effective because the growing decisions are taken without considering the state of the MapReduce framework (e.g., if the MapReduce framework is idle, adding nodes is useless). Also, with a large number of transient nodes, the contention on the HDFS increases considerably. On the other hand, GS monitors the number of tasks running within the MapReduce framework and initiates grow and shrink requests based on the throughput. However, the best policy is GGD, which provides local storage for the provisioned nodes, reducing in this manner the costs of transferring the output data across the network and writing it on the disks of the core nodes.

2.6 Related Work

The current state of the art work reveals several approaches for improving the performance of MapReduce clusters by different architectural enhancements that facilitate dynamic resizing or isolation.

Mesos [63] multiplexes a physical cluster between multiple frameworks such that different types of applications (MPI, MapReduce) may share access to large datasets. The scheduling mechanism enabled by Mesos is based on resource offers. The scheduler decides how many resources each framework is entitled to with respect to a fair-share policy, while the frameworks decide on their own which resources should be accepted based on the given framework side policies. Therefore, frameworks have the ability to reject an offer that does not comply with their resource requirements. By delegating the scheduling control to the frameworks, Mesos achieves high scalability, robustness, and stability.

While our multi-framework approach enables different types of isolation (performance, data, failure, and version isolation), Mesos achieves high utilization, efficient data sharing and resource isolation through OS container technologies, such as Linux Containers.

Moon [79] applies the hybrid architecture with core and transient nodes in order to deploy MapReduce clusters on volunteer computing systems. Towards this goal, a small number of dedicated nodes with high reliability are deployed behind the volunteer computing system for storage and computing power. To overcome the impact of unexpected interruptions, Moon proactively replicates tasks towards the job completion. In addition, Moon uses the dedicated nodes not only as data servers, but also to execute task replicas. In our design, the transient nodes are used to improve the performance of the MapReduce cluster deployed on the core nodes, as opposed to Moon, which uses the dedicated nodes to supplement the volunteer computing system.

Elastizer [62] estimates the impact of the cluster resource properties on the MapReduce job execution. In order to address a given cluster sizing problem, the Elastizer performs a search through the space of resources and job configuration parameters. This search is driven by a *what-if* engine that explores execution profiles of MapReduce jobs to find the optimal set of resources and job configuration parameters. As opposed to the offline reasoning of the Elastizer, our approach uses the throughput as the deciding factor for growing or shrinking the MapReduce cluster.

CAM [78] is a resource scheduler for the cloud, designed to improve the performance of MapReduce jobs by maximizing the data and job locality. The scheduler is based on a flow-network algorithm that not only optimizes the initial placement of the data, jobs and tasks, but is also able to evaluate the cost of readjusting the existing assignments. For optimal tasks assignments, CAM makes the MapReduce scheduler aware of the hidden compute, storage, and network topologies. Our approach relies on the default strategy for achieving data locality used by the FIFO scheduler incorporated in Hadoop. In particular, the scheduler attempts to place a map task as close as possible to a machine that contains a replica of the input data.

Our KOALA scheduler along with the MR-Runner provide different types of isolation that improve the performance, data management, fault tolerance and development of MapReduce frameworks. In addition, the MR-Runner provides a grow-and-shrink mechanism that dynamically changes the size of the MapReduce framework.

2.7 Conclusion

We have presented KOALA-MR, a resource manager that enables running multiple instances of the MapReduce framework in single multicluster systems. KOALA-MR is built around two design elements: a model to run elastic MapReduce computations and a protocol to enable scheduling decisions between the resource manager and the active frame-

works. Together, these elements allow KOALA-MR to respond to workload changes and to improve system utilization. In order to resize a MapReduce framework while keeping the reconfiguration costs relatively low, KOALA-MR employs transient nodes, which do not store any data and are only used to execute tasks.

We have implemented KOALA-MR through extensions to our KOALA grid scheduler and we have evaluated the performance of the grow-and-shrink mechanism on the DAS. We have found that CPU-bound jobs scale on transient nodes as well as on core nodes, while the IO-bound jobs suffer a high performance degradation when the number of transient nodes increases. Furthermore, we have showed that growing and shrinking a MapReduce framework based on its throughput achieves better performance than provisioning statically a large number of transient nodes.

Chapter 3

Balancing the Service Levels of Multiple Frameworks

3.1 Introduction

MapReduce and similar computing frameworks are now widely used by institutes and commercial companies (e.g., Google [31], Facebook [144], Yahoo! [23]) because of their ability to efficiently use large sets of computing resources and to analyze large data volumes. MapReduce workloads may be very heterogeneous in terms of their data size and their resource requirements [62], and mixing them within a single instance of a computing framework may lead to conflicting optimization goals. Therefore, *isolating* MapReduce workloads and their data while dynamically *balancing* the resources across them is very attractive for many organizations. In this chapter we present the design and analysis of FAWKES[1], a mechanism for *dynamic* resource provisioning of multiple MapReduce instances in single large-scale infrastructures.

In Chapter 2 we have identified four types of isolation that can be enabled by having multiple MapReduce frameworks within a single multicluster system: data isolation, failure isolation, version isolation, and performance isolation. Whereas the first three forms of isolation are easily enforced by a resource manager that can deploy multiple instances of the MapReduce framework along with their corresponding filesystems on disjoint sets of nodes, performance isolation is more difficult to achieve (and define)—as the workloads of the instances may vary considerably over their lifetimes, deploying them on static partitions of the system may lead to an imbalance in the levels of service they receive.

To dynamically provision multiple framework instances at runtime, FAWKES defines a new abstraction of the MapReduce framework called the *MR-cluster*. An MR-cluster is initially deployed (along with its filesystem) on a system partition of a certain minimum

[1] FAWKES is a phoenix bird which in Greek mythology is reborn from its own ashes just like our MapReduce clusters grow and shrink.

size consisting of *core* nodes when its first job is submitted, and it will remain active as long as it receives additional jobs. The allocation of an MR-cluster can grow (and later shrink) by adding (removing) *transient* or *transient-core* nodes that don't store any data or only output data, respectively, thus breaking the standard MapReduce data locality assumption but allowing fast reconfiguration.

FAWKES implements three types of policies for setting and periodically adjusting the *weights* of the active MR-clusters that indicate the shares of the resources they are entitled to, and to resize their allocations accordingly. These policies try to assess the load conditions of the MR-clusters by considering their queue lengths, their resource utilizations, or their performance (in terms of, e.g., job slowdown) when setting the weights. The most important performance metric we use to assess the actual performance of the MR-clusters is the average job slowdown.

Another possible solution for provisioning multiple MapReduce instances is to share the distributed filesystem across all frameworks and to employ two-level scheduling by delegating the scheduling control to the frameworks, as is done in Mesos [63]. There, a high-level resource manager initiates resource offers to the frameworks, which need specific policies to decide whether to accept or reject these offers. Mesos achieves near-optimal data locality when resources are frequently released by the frameworks. Instead, our solution targets performance isolation for time-varying workloads, but breaks the data locality assumptions to enable fast framework reconfigurations. Whereas Mesos has an offer-based scheduling system, FAWKES employs a feedback mechanism and balances the allocations of multiple frameworks by monitoring their operation.

The contributions of this chapter are as follows:

1. We define the abstraction of the MR-cluster which is a set of resources that can grow and shrink, that has MapReduce installed on its core in the usual way, but that relaxes the MapReduce data locality assumptions for nodes outside its core (Section 3.2).

2. We provide a comprehensive taxonomy of policies for provisioning multiple MR-clusters that take into account their dynamic load conditions as perceived from their queue lengths, the utilizations of the resources allocated to them, or the performance they deliver (Section 3.3).

3. With a set of micro-experiments in a real multicluster system, we analyze, among other aspects of FAWKES, the benefit of trading data locality for dynamicity (Section 3.5). We find that the performance penalty induced by a relaxed data locality model in MapReduce is not prohibitive.

4. With a set of macro-experiments in the multicluster system, we evaluate three classes of policies used by FAWKES for balancing the allocations of multiple MR-

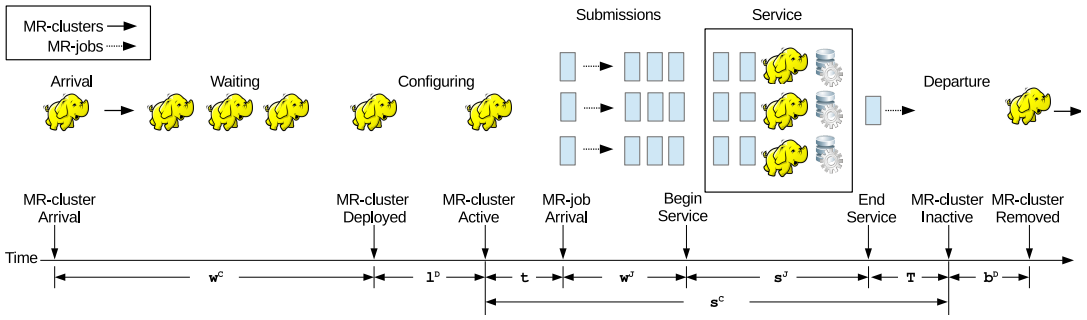


Figure 3.1: An overview of the queuing system of FAWKES. The system handles requests for deploying MapReduce frameworks (MR-clusters) with a global queue, and each active MR-cluster manages an internal queue of MapReduce jobs (MR-jobs). The Hadoop logo is taken from [10].

clusters. We show that our system delivers good results even for unfavorable workloads (Section 3.6).

3.2 System Model

In this section we propose a scheduling and provisioning structure for deploying multiple concurrent MapReduce frameworks, which can be resized dynamically at runtime.

3.2.1 Scheduling and Provisioning Structure

The system architecture assumed by MapReduce has resources where processors and storage are co-located. We assume these resources to be organized in multiple physical clusters that operate as a single distributed computing system managed by the resource manager FAWKES, which decides how resources are to be balanced across multiple MapReduce frameworks.

In Figure 3.1 we show the queueing system managed by FAWKES. FAWKES receives two types of requests, one for activating new MR-clusters, and one for executing MapReduce jobs (MR-jobs) that identify the MR-cluster in which they have to be executed. These requests are serviced in multiple queues, all using the FIFO scheduling discipline.

The system queues all requests for new MR-clusters in a global queue managed by FAWKES. The time required to activate a new MR-cluster consists of the time the MR-cluster has to wait to be deployed on physical resources (w^C) and the time required to load the input dataset from an external persistent storage (l^D). Each active MR-cluster maintains an internal queue of MR-job requests targeted at it. The response time of an MR-job is equal to the sum of its waiting time in the MR-cluster's queue (w^J) and its

execution time (s^J). If the weight of an MR-cluster in the FAWKES mechanism is equal to 0 indicating no task execution for at least a certain duration T (see Section 3.3.2), FAWKES marks it as inactive, deallocates its resources, and removes it from the system. Prior to the removal of an MR-cluster, its (output) data is saved on the persistent storage in the time interval of length b^D.

3.2.2 Dynamic MR-clusters

In order to balance resources across the active MR-clusters, FAWKES has to be able to resize them by growing and shrinking the number of resources allocated to them at runtime. In the traditional static deployment of the MapReduce frameworks, the data of the HDFS are replicated and distributed uniformly across the nodes. Then, techniques like delay scheduling [144] can maximize the number of tasks that achieve data locality. FAWKES has the ability of changing the allocations, and has a relaxed data locality model, through a new abstraction of the MapReduce framework that we call *dynamic MR-cluster*, which comprises three types of nodes.

The most important requirement for FAWKES is to provide reliable data management so that when nodes are removed from an MR-cluster and the number of replicas is small, no data are lost. However, creating numerous replicas is not desired because of the increased usage of storage space. Thus, when removing nodes from an active MR-cluster, FAWKES needs to replicate the data they store. As data-intensive applications are processing large volumes of data, their replication makes the resizing of the MR-cluster slow. To enable fast reconfigurations, the removed nodes of the MR-cluster should store relatively small amounts of data.

Similar to the static deployment of MapReduce, FAWKES permanently allocates to an MR-cluster an initial set of *core nodes* used for both executing jobs and storing (input and output) data. During the time an MR-cluster is active, FAWKES may temporarily increase its capacity by provisioning *transient* or *transient-core* nodes, which break the traditional model for data locality. The former are instantiated without local storage such that the jobs they execute read and write input/output data from/to core nodes. The latter are different from the core nodes only by the lack of input data, thus the jobs they execute also need to transfer input data from core nodes, but they can use the local storage to write output data. As a consequence, FAWKES can grow the size of an MR-cluster fast as no data movement will be involved at all. Shrinking the size of an MR-cluster by removing transient-core nodes does require saving the output data stored on them, the amounts of which is usually very small in comparison to the input dataset distributed on the core nodes. Nevertheless, large fractions of these nodes may saturate the network (both) or increase the contention on the physical disks of the core nodes (especially the transient nodes). When to resize an MR-cluster, and whether then to use transient or transient-core nodes, is ex-

plained in Section 3.3. Different from our dynamic MR-cluster approach, Amazon Elastic MapReduce [2] does not support data redistribution when shrinking the size of the cluster.

3.3 Balanced Service Levels

In this section we derive a fairness metric to determine the imbalance between concurrent MR-clusters (Section 3.3.1). Our scheduling system, FAWKES, targets weighted fair allocations, such that different MR-clusters converge to fractions of the system resources proportional to their weights (Section 3.3.2). FAWKES dynamically updates the weights based on different metrics exposed by the MR-clusters at runtime. We design three classes of weighting policies which consider the input to the MR-clusters (*demand*), the operation of the MR-clusters (*usage*), and the output of the MR-clusters (*runtime performance*) (Section 3.3.3).

3.3.1 Fairness

Fairness is a major issue in resource provisioning and job scheduling, and is an optimization target that may conflict with performance metrics such as response time [69, 144]. Although there exists a large volume of literature that analyzes the notion of fairness in communication systems [52], there is no generally agreed upon measure of the fairness between jobs. The fairness of a queuing system has been defined either as a measure of the time spent waiting [14], possibly with respect to job size [135]. A fairness metric that accounts for both job arrival times and sizes is proposed in [96]. We adapt the latter for evaluating the fairness of a provisioning policy.

The key element is the assumption that, at any moment of time t, the MR-clusters may be entitled to shares of the total datacenter capacity C, proportionally to their given weights. For an MR-cluster i, the difference between the fraction $c_i(t)$ of resources it currently has and the share of resources it should have based on its weight $w_i(t)$ (see Section 3.3.2) at moment t is defined as its *temporal discrimination*

$$d_i(t) = c_i(t) - w_i(t). \tag{3.1}$$

We define the *discrimination* of MR-cluster i during a time interval $[t_1, t_2]$ by

$$D_i(t_1, t_2) = \int_{t_1}^{t_2} (c_i(t) - w_i(t))dt. \tag{3.2}$$

Setting $t_1 = d_i$ and $t_2 = r_i$ with d_i and r_i the moments of the request for the deployment and the removal of the MR-cluster, respectively, we obtain the *overall discrimination* of the MR-cluster.

When all the resources of the datacenter are occupied all the time, every positive discrimination is balanced with negative discrimination, and so $\sum_i D_i(t_1, t_2) = 0$ for any time interval $[t_1, t_2]$, which makes the expected mean value of the discrimination $E[D] = 0$. The fairness (or balance) of the system is given by the variance of the discrimination, which we call the *global discrimination factor*:

$$Var(D) = E[D^2] - E[D]^2 = E[D^2] \tag{3.3}$$

We consider the allocations of the MR-clusters to be *imbalanced* or *unfair*, when the global discrimination factor is larger than a predefined parameter τ.

3.3.2 The Fawkes Mechanism

We want to provision resources to multiple MR-clusters in a single datacenter or multi-cluster system to give MR-clusters similar levels of service. To achieve this, we want to assign each MR-cluster a dynamically changing weight that indicates the share of the resources it is entitled to.

Admission Policy. For each MR-cluster i FAWKES assumes that there is a minimum number of core nodes m_i (and a corresponding minimum share), which may be set by a system administrator or computed based on the amount of data the cluster has to process. The system guarantees the minimum share of an MR-cluster as long as it has running jobs, even if according to its current weight it is only entitled to a smaller share.

If on the arrival of a new MR-cluster, the sum of its minimum share and of the minimum shares of the active MR-clusters exceeds 1, the new MR-cluster is queued (in FIFO order). Otherwise, the system gives it its minimum share of the resources within a time interval T from its arrival, by shrinking the active MR-clusters which are above their minimum shares proportionally to their current weights (but not going below their minimum shares). When later an active MR-cluster finishes its workload and releases the resources it holds, FAWKES checks to see if the MR-cluster at the head of the queue fits. After a new MR-cluster receives its minimum share, the system monitors its state along with the states of the other active MR-clusters. The weights of the active MR-clusters are periodically updated after every interval of length T.

Changing Shares. To ensure that MR-clusters with different workloads experience similar service levels (e.g, job slowdown), we propose three complementary mechanisms, employed by FAWKES, which target either a subset or the entire set of the active MR-clusters, and operate at different timescales.

The MR-clusters collect periodically samples of different aspects of system operation, such as demand $d(t)$, resource utilization $r(t)$, or actual performance $p(t)$. FAWKES *monitors* a specific metric related to these aspects and sets the weight (*weighting mechanism*) $w_i(t)$ of MR-cluster i at time t to the average value of the samples y_i collected

during the last time interval T:

$$w_i(t) = \frac{y_i(t)}{\sum_{k=1}^{n} y_k(t)}, \tag{3.4}$$

where n is the number of active clusters at time t.

After updating the weights, the temporal discriminations of the MR-cluster are determined as well. When the MR-clusters are imbalanced that is, the global discrimination factor exceeds the predefined threshold τ, FAWKES changes their shares proportionally to their dynamic weights. To resize an MR-cluster, FAWKES employs the *grow and shrink mechanisms* based on provisioning temporary nodes, which can be either transient or transient-core (see Section 3.2).

The shrinking mechanism guarantees that the MR-clusters with positive discrimination reach their fair shares by releasing the surplus of resources they hold. Based on the type of nodes which are removed, we distinguish two possible ways of shrinking an active MR-cluster, *fast preemption* (FP) or *delayed preemption* (DP). The former is suitable for transient nodes and simply kills the currently running tasks, which are later re-scheduled by the MR-cluster. The latter applies to transient-core nodes, which besides removing their running tasks also require the replication of their local data. FAWKES removes transient-core nodes in non-decreasing order of the amount of data they locally store.

The growing mechanism ensures that MR-clusters with negative discrimination achieve their fair shares by extending their current shares. To do so, the MR-cluster is grown either with transient (TR) or with transient-core (TC) nodes. The former have good performance only for compute-intensive workloads, which generate small amounts of data. The local storage of the latter type of nodes significantly improves the performance of highly disk intensive workloads (Section 3.5). The type of growing employed by FAWKES is a predefined system parameter.

3.3.3 Weighting Policies

To balance the allocations of multiple MR-clusters, we investigate a comprehensive design space, which covers the input to the system, the state of the system, and the output (runtime performance) of the system. We focus, respectively, on the demand of the workloads submitted to MR-clusters, on the usage of resources allocated to them, and on the runtime performance. For each of these, we propose three exemplary policies.

For all policies we investigate in this work, the weights of the MR-clusters and the global discrimination are recomputed after every interval of length T. Only when the global discrimination exceeds a threshold τ are the allocations of the MR-clusters actually changed according to the new weights.

Demand-based Weighting. Demand-based weighting policies take into account the input to the system (the demand). They establish the fair shares of the MR-clusters as proportional to the sizes of the workloads submitted to their queues. As we do not assume any prior knowledge about the workloads, we identify three ways of defining the size of a workload at time t, viz. with respect to the number of waiting jobs, the size of the input data of the jobs, and the number of waiting tasks:

1. Job Demand (JD). The JD policy sets the demand of the MR-cluster to the total *number of jobs* waiting in the queue.

2. Data Demand (DD). The DD policy sets the weight of the MR-cluster to the total *input data volume* of the jobs waiting in the queue.

3. Task Demand (TD). The TD policy gives an estimate of the MR-cluster demand at finer granularity than JD, by taking into account the total *number of tasks* waiting in the queue.

Although each of these policies is inherently inaccurate, for example JD because the duration of jobs ranges from minutes to hours, we expect demand-based weighting policies to lead to better system performance than no policy.

Usage-based Weighting. Usage-based policies monitor the state of the system; here, we propose policies that monitor the utilization of the physical resources currently allocated to MR-clusters. We identify two main resources to monitor, processor usage and disk usage, and derive three policies:

1. Processor Usage (PU). The PU policy sets the usage at time t to the fraction of *utilized processing units* (cores or slots) from the total configured capacity of the MR-cluster.

2. Disk Usage (DU). The DU policy sets the usage at time t to the ratio between the total *output data* generated by the MR-cluster and its current storage capacity.

3. Resource Usage (RU). The RU policy combines the previous two policies by accounting for *both compute and storage resources*, processor and disk, as follows:

$$u_i(t) = \psi \cdot u_i^P + (1 - \psi) \cdot u_i^D, \tag{3.5}$$

   where u_i^P and u_i^D are the (normalized) resource usages as computed by the PU and DU policies, respectively, and the parameter $\psi \in (0, 1)$ reflects the relative importance of the two resources.

Performance-based Weighting. The performance-based policies assign the fair shares of the MR-clusters based on the performance of the system at runtime, so that MR-clusters with poor performance receive larger fractions of resources and, thus, improve

their performance. We use in this work two performance metrics, slowdown (low values are ideal) and throughput (high values are ideal) to calculate the weights of the MR-clusters as follows:

1. Job Slowdown (JS). The JS policy calculates the slowdown of each running job at time t as the ratio between the elapsed time since the job started and the job execution time on a reference static MR-cluster (only for this policy, assumed known at the start of the job in the MR-cluster). We consider the weights of the MR-clusters to be all equal and positive at time $t = 0$. The weight of the MR-cluster i at time $t > 0$ is set to the *average job slowdown* of all jobs s_i which are waiting in the queue.

2. Job Throughput (JT). The JT policy considers the performance of MR-cluster i at time t to be the ratio q_i between the number of *jobs* completed and the total number of jobs waiting in the queue. The weight is, then:

$$p_i(t) = a^{-q_i(t)} \tag{3.6}$$

   where $a > 1$ is a constant (we set $a = 2$). The share of an MR-cluster i is entitled to increases inversely proportional with the measured throughput from C/a ($q_i \to 1$) to C ($q_i \to 0$).

3. Task Throughput (TT). The TT policy is similar to the JT policy. The TT policy uses a throughput computed as the ratio between the number of *tasks* completed and the total number of tasks waiting in the queue. Equation (3.6) still holds, with the ratio q_i now referring to tasks instead of jobs.

We compare our policies with two *baselines*, NoPolicy (None) and EqualShares (EQ). The former makes the MR-clusters run permanently on their minimum shares. For the latter, the available resources are always equally divided between the active MR-clusters.

3.4 Experimental Setup

In this section we present the experimental setup for assessing the performance of several aspects of system operation (Section 3.5) and of the full FAWKES mechanism for balancing resources across MR-clusters (Section 3.6). The main differences between our and previous experimental setup are the use of a comprehensive set of representative MapReduce applications (including a real, complex workflow), the design of five workloads (including several unfavorable cases), and the use of a multicluster testbed (only in one experiment). The total time used for experimentation exceeded 3 real months and over 60,000 hours system time.

Table 3.1: A summary of the MapReduce applications used in our experiments.

Job	Type	Data	Input	Output
Wordcount	compute	Random	200 GB	5.5 MB
Sort	disk	Random	200 GB	200 GB
PageRank	compute	Random	50 GB	1.5 MB
KMeans	compute, disk	Random	70 GB	72 GB
TrackerOverTime	compute	BitTorrent	100 GB	3.9 MB
ActiveHashes	disk, compute	BitTorrent	100 GB	90 KB

3.4.1 Clusters

We run experiments on the Dutch six-cluster wide-area computer system DAS (fourth generation) [118]. The system has in total roughly 200 dual-quad-core compute nodes with 24 GB memory per node and 150 TB total storage, connected within the clusters through 1 Gigabit Ethernet (GbE) and 20 Gbps QDR InfiniBand (IB) networks. The compute nodes from different clusters communicate over dedicated 10 Gbps light paths provided by Surfnet. The largest cluster in terms of the number of nodes, situated at the VU Amsterdam, has roughly 70 nodes divided into 4 racks. The GbE interconnect is based on two 48-ports 1 GbE switches (symmetric, backplane cabled). The IB network is enabled by six 36-ports InfiniBand switches, organized in a fat tree, with 4 access switches, 2 at root. This architecture, which is useful for both data processing and high-performance computing, currently services about 300 scientists.

In our experiments we restrict ourselves to well-connected datacenters and we use a standard setup of Hadoop (version 1.0.0) over InfiniBand. We configure the HDFS on a virtual disk device (with RAID 0 software) that runs over 2 physical devices with 2 TB storage in total per node. The data are stored in the HDFS in blocks of 128 MB with a default replication factor of 3. With 8 cores per node enabled (no hyperthreading), we configure the TaskTrackers with 6 map slots and 2 reduce slots.

Real-world experimentation was greatly facilitated by the DAS system. However, as the system is shared between many users, we also encountered practical restrictions. We have completed the set of micro-experiments presented in Section 3.5 reserving 20 up to 30 nodes for a week. As we explore a large design space experimentally, we summarize for the large experiments in Section 3.6 only results from single executions. It took more than 2 months to complete the macro-experiments we have designed in this chapter.

3.4.2 MapReduce Applications

The choice of MapReduce applications is crucial for a meaningful experimental evaluation. We use both simple, synthetic *applications* from a popular MapReduce benchmark,

HiBench [64] and a complex, real MapReduce-based *logical workflow*, BTWorld [61, 137]. Table 3.1 gives a high-level summary of these MapReduce applications, which we extend with a detailed description in this section.

HiBench includes a suite of simple synthetic benchmarks for data transformation, web search, and machine learning, along with automatic tools for data generation:

1. Wordcount (WT) counts the number of occurrences of each word in a given set of input files. The map tasks simply emit key-value pairs with the partial counts of each word, and the reduce tasks aggregate these counts into the final sum. Wordcount is mostly compute-intensive and shuffles a small number of bytes from map to reduce tasks.

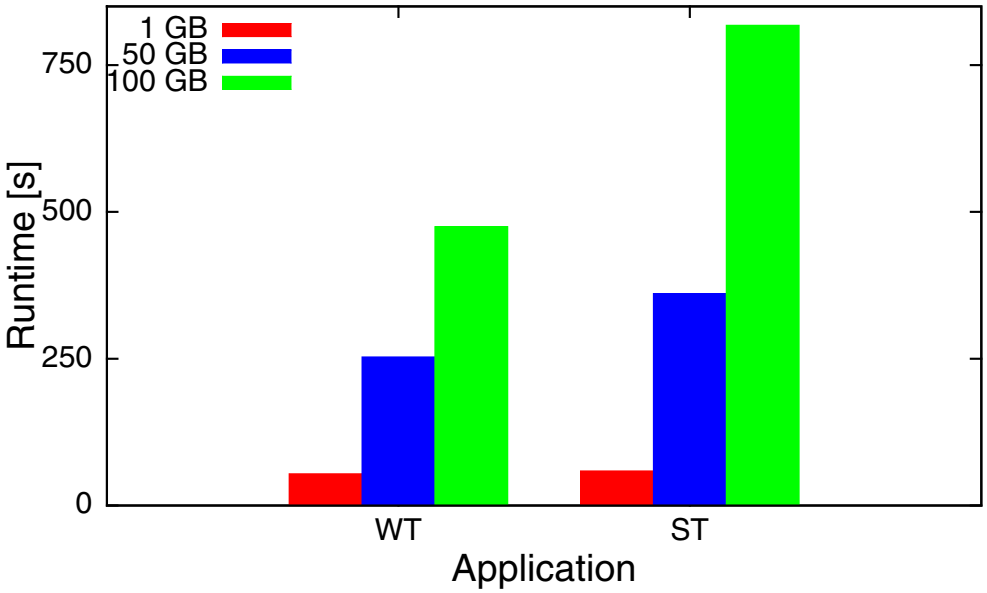
2. Sort (ST) is a disk-intensive application in which the identity function stands as both map and reduce functions and the actual sorting is executed while the data is shuffled from map to reduce tasks.

3. PageRank (PR) is a link analysis algorithm widely used in web search engines to calculate the ranks of the web pages based on the number of reference links. The MapReduce implementation of the workload consists of three compute-intensive jobs which iteratively compute the ranking scores of all pages.

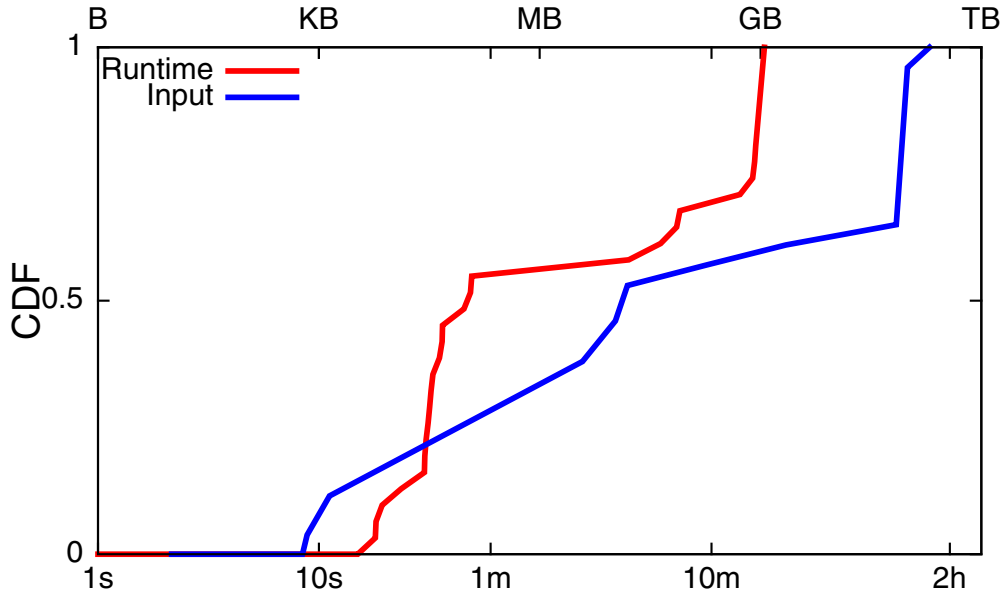
4. KMeans (KM) is a data mining clustering algorithm for multi-dimensional numerical samples. The workload employs two MapReduce jobs which resemble the characteristics of Wordcount and Sort. The former is mostly compute-intensive and iteratively computes the centroid of each cluster, thus swallowing a large fraction of the input. The latter is disk-intensive and reorders the data by assigning each sample to a cluster.

BTWorld is a complex, real-world MapReduce-based logical workflow for processing the data collected periodically over many years from the global-scale peer-to-peer system BitTorrent [137]. The dataset contains per tracker statistics ($scrapes$) stored in a multi-column layout which includes the identifier for the BitTorrent content ($hash$), the URL of the BitTorrent tracker ($tracker$), the time when the status information was logged ($timestamp$), the number of users having the full and part of the content ($seeders$ and $leechers$), and the number of downloads at the moment of sampling ($downloads$).

A MapReduce-based workflow which currently consists of 14 high-level queries expressed in Pig Latin processes data and leads to understand the evolution over time of the global BitTorrent system. The queries expressed in this MapReduce workflow cover a broad range of SQL-like operators (e.g., join, aggregation, filtering, projection), break down into more than 20 MapReduce jobs, and exhibit three levels of data dependency: inter-query (when the input of the query needs to be generated by another query), inter-job (when a query is divided into several MapReduce jobs), and intra-job (between map



(a) HiBench jobs



(b) BTWorld jobs

Figure 3.2: The runtime performance of the jobs in HiBench and BTWorld on a 10-node static MR-cluster used as reference for the job slowdowns.

and reduce tasks). The workflow combines both compute and disk intensive jobs with small (10^{-6}) and high (10^{2}) job selectivities, where job selectivity is defined as the ratio between the output and input sizes. Thus, this real workflow is very challenging for MapReduce-based data processing.

In our experiments we use not only the complete workflow, but also two single queries individually:

1. TrackerOverTime (TT) groups the input dataset by tracker, sorts it by timestamp field, and applies different aggregation functions (e.g., count, avg, sum) on the remaining fields of the records. The query translates into a single, compute-intensive *map-heavy* job, and its output is 5 orders of magnitude smaller than the dataset size.

2. ActiveHashes (AH) determines the number of active hashes in the system at every moment of time. The query is split in two MapReduce jobs, one disk-intensive with high (1) selectivity, and the other compute-intensive with very small (10^{-6}) selectivity. The first job emits all distinct hash and timestamp pairs to a second job, which further counts the number of unique hashes at every moment of time.

3.4.3 MapReduce Workloads

We consider workloads that cover many aspects (e.g., job types, data sizes, submission patterns) identified in synthetic benchmarks, production clusters, and BTWorld [23, 64]. To this end, we design three categories of workloads, based on which we generate 19 different workloads which we use in our micro- and macro-experiments:

1. Single job - Single size (SS). The SS workloads contain a number of identical synthetic or real-world jobs presented in Table 3.1. We use 6 such workloads of size one (one job) with fixed input data sizes (see Table 3.1) in Section 3.5. In Sections 3.6.1 and 3.6.2 we use SS workloads with 50 and 100 jobs, respectively, in which we employ the same submission pattern with all jobs submitted at once (*batch*), which is also used in many synthetic benchmarks.

2. Multiple jobs - Single size (MS). The MS workloads combine several types of jobs (e.g., WT and ST) with the following input data sizes: 1 GB (small), 50 GB (medium), and 100 GB (large). We generate 3 workloads of this type based on WT and ST (Section 3.6.3, where we also describe the job arrival process) which have hundreds of small jobs.

3. Multiple jobs - Multiple sizes (MM). The MM workloads combine several job types with different input data sizes which are summarized in Figure 3.2. The jobs in HiBench (e.g., WT and ST) have the same input sizes as in the MS workloads.

BTWorld employs 26 jobs with 13 distinct input sizes. We use three instances of this type of workloads in which small jobs prevail (Section 3.6.3, including the arrival process).

Given the high imbalance between the workloads we use, the discrimination threshold does not have a significant impact, thus we set τ to a small value (10 in our experiments). In our workloads, most of the jobs take between 1 and 4 minutes to complete (Figure 3.2) and a couple of them arrive every minute. Thus, we set the weight update interval T to a value in the order of a few minutes (1 or 2 minutes in our experiments).

We design the evaluation of FAWKES in two steps. First, we design four *micro-experiments* using SS workloads to assess different aspects of system operation (see Section 3.5). Then we design five *macro-experiments* using instances of all types of workloads to assess the performance of FAWKES. Towards this end, we combine highly imbalanced workloads to create extreme conditions of variable load across distinct MR-clusters (Section 3.6).

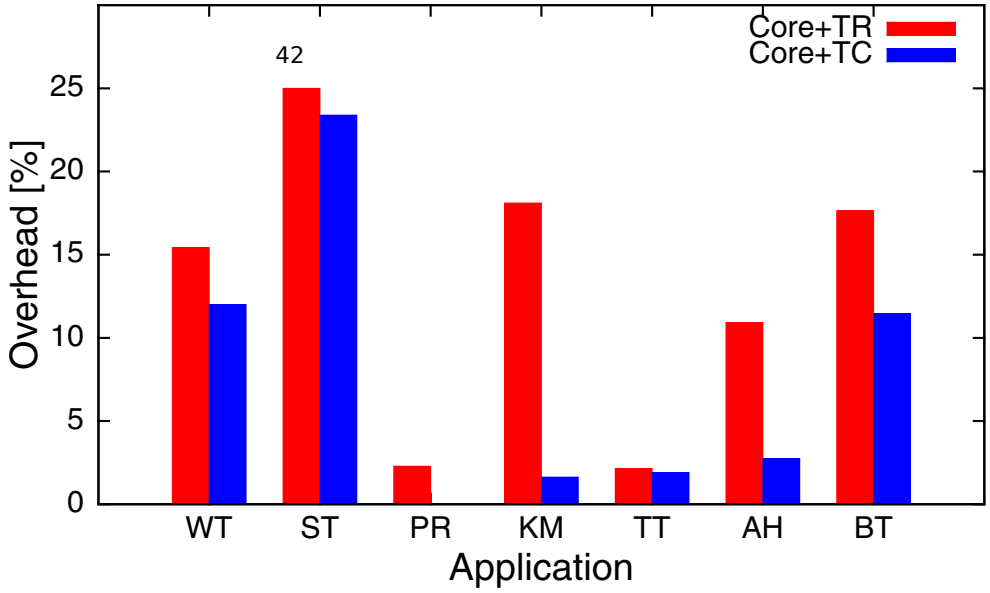
3.5 Micro-Experiments

In this section we present the results of four experiments that each address a separate aspect of the performance of single MR-clusters. We investigate the performance of several MapReduce applications in single MR-clusters with different configurations with respect to the types of nodes (Section 3.5.1) and whether a single or multiple physical clusters are used (Section 3.5.2), and we assess the performance of growing (Section 3.5.3) and shrinking (Section 3.5.4) single MR-clusters at runtime. We measure the overhead of these configurations relative to a static MR-cluster with only core nodes. For all jobs in the micro-experiments we use the input dataset sizes defined in Table 3.1.

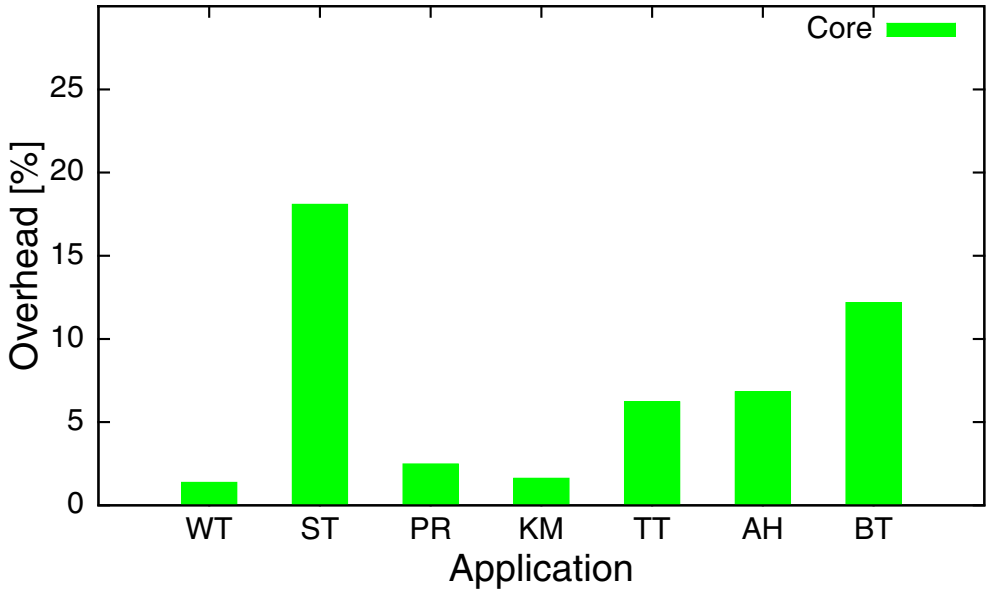
3.5.1 Node Types

We assess the impact on the runtimes of jobs of using the three types of MR-cluster nodes presented in our system model in Section 3.2. In our previous work [45], we have found that the execution time of disk-intensive jobs increases with the ratio between transient and core nodes, while the performance of compute-intensive jobs is independent of the types of nodes.

Using transient-core nodes instead of transient nodes reduces the overhead for disk-intensive jobs considerably (Figure 3.3(a)). We set up static 20-node MR-clusters with only core nodes and with equal numbers of core and transient/transient-core nodes. In the former configuration, the input dataset is distributed across all nodes of the MR-cluster, while in the latter two configurations, the input dataset is distributed on 50%



(a) Node types



(b) Multicluster deployment

Figure 3.3: The overhead of running a single MR-job on 20-node MR-clusters with equal fractions of core and transient, and of core and transient-core nodes (a), and with resources evenly co-allocated from two physical clusters (b).

of the MR-cluster nodes. Figure 3.3(a) shows that with transient-core nodes instead of transient nodes, the overhead for disk-intensive jobs relative to the job execution on only core nodes is much smaller. In particular, Sort shows a significant improvement, decreasing the overhead from 40% with transient nodes to 23% with transient-core nodes, as do KMeans and ActiveHashes.

In our model, dynamically provisioning MR-clusters by means of a grow-shrink mechanism at runtime comes at the expense of poor data locality, as the tasks executed on transient or transient-core nodes need to transfer their input across the network. Nevertheless, we have shown here, at least in a cluster with a high-bandwidth network, that the impact of running non-local tasks can be limited by using transient-core nodes.

3.5.2 Multicluster Deployment

In this section we assess the impact on job execution time of deploying single MR-clusters by co-allocating resources from different physical clusters in our multicluster system with high-speed wide-area connections. Previous work [36] has shown that co-allocation of parallel applications in multicluster systems is beneficial because of reduced job wait times if the overhead due to the slower wide-area communication is less than 25%.

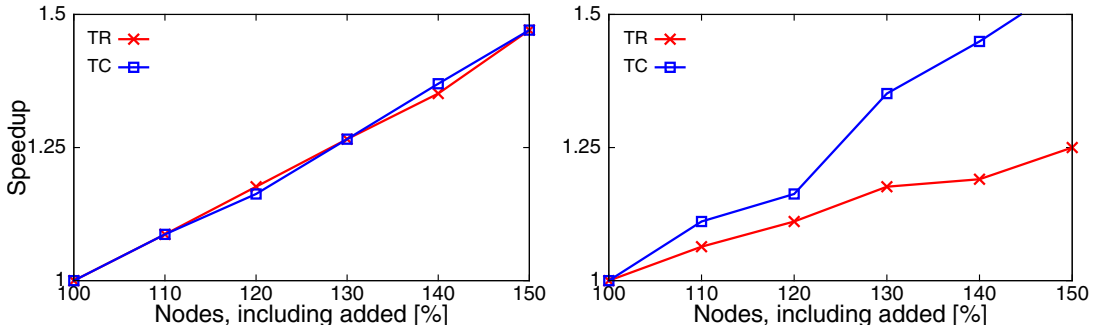
MapReduce jobs can run with low to moderate overhead in co-allocated MR-clusters over a high-speed interconnect (Figure 3.3(b)). We set up 20-node static MR-clusters, with nodes co-allocated evenly from two physical clusters located at two universities in Amsterdam. Figure 3.3(b) shows that most of the applications exhibit low overhead when they run on co-allocated MR-clusters. For the complete BTWorld workflow and Sort, which are mostly composed by disk-intensive jobs, a co-allocated MR-cluster increases their execution times by less than 20% relative to the single physical cluster deployment.

Although we have shown here that MR-clusters may be provisioned with co-allocated resources, we design the remaining experiments within a single physical cluster.

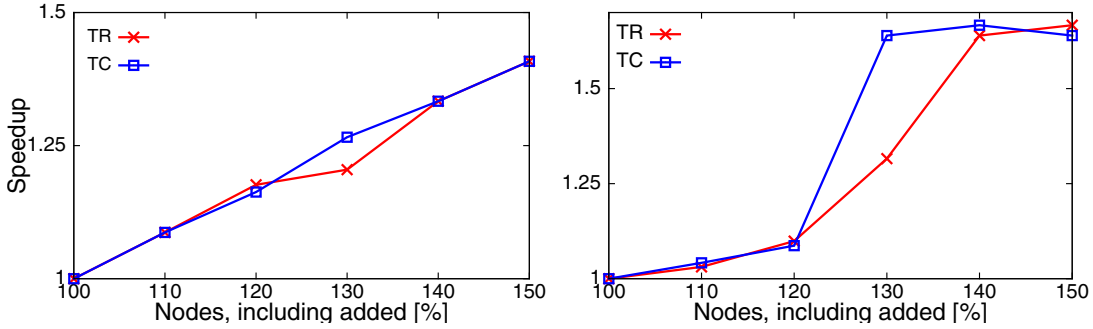
3.5.3 Growing MR-clusters

We measure the speedup of single jobs when the MR-cluster grows with different fractions of transient-core or transient nodes before the job starts. The conveniently parallel layout of MapReduce applications [31] with only a single predetermined synchronization point between the map and reduce phases, in principle makes them malleable applications [40] that can benefit from dynamic resource provisioning at runtime [25].

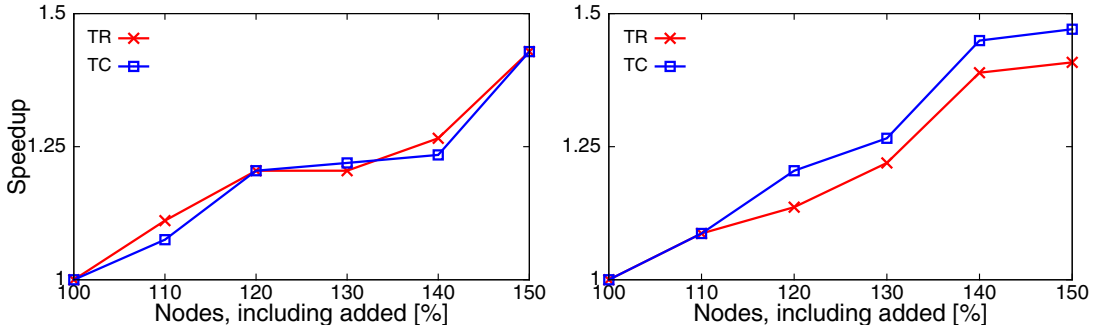
The execution time of MapReduce jobs can be improved with a growing mechanism at runtime by relaxing the data locality constraints. We set up dynamic MR-clusters with 20 core nodes which we extend at runtime with different fractions of transient or transient-core nodes.



(a) Wordcount (b) Sort



(c) TrackerOverTime (d) PageRank



(e) KMeans (f) ActiveHashes

Figure 3.4: The job speedup relative to the execution time measured on a static 20-node MR-cluster, when growing the MR-cluster with different fractions of transient (TR) or transient-core (TC) nodes.

Despite the lack of data locality, transient (TR) nodes show good performance with one exception. For Sort, which is a highly disk-intensive job, large fractions of TR nodes increase the contention on the physical disks of the core nodes (Figure 3.4(b)), thus limiting the speedup. With the relaxed data locality model of the transient-core (TC) nodes, jobs may write the data they generate on their local storage. This explains the linear increase of the applications speedups with the number of transient-core nodes (Figure 3.4). The supra-linear speedup of PageRank is an anomaly due to the nondeterministic convergence of the iterative MapReduce jobs.

We can improve the performance of a broad range of MapReduce applications by relaxing the data locality model. Moreover, even in more extreme cases of no locality, transient nodes show good performance for applications that generate small amounts of data.

3.5.4 Shrinking MR-clusters

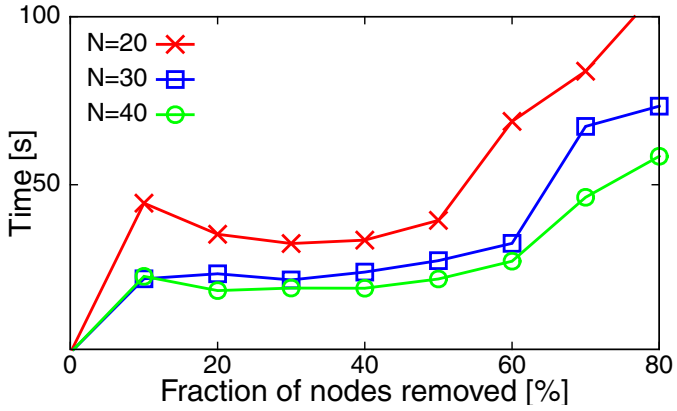
In this section we investigate the overhead of reorganizing the data within HDFS (Figure 3.5(a)) and the job slowdown when different fractions of transient-core nodes are removed from the MR-cluster at the moment the job starts running (Figure 3.5(b), 3.5(c)). Although in practice the transient-core nodes store less data than the core nodes, we assume in this micro-experiment a *worst-case scenario* in which both types of nodes store the same amounts of data.

When resizing an MR-cluster to 50% of its size, the time overhead of reorganizing the data in HDFS increases linearly with the number of nodes removed. We set up MR-clusters with different numbers of core and transient-core nodes. The former represent 20% of the cluster size and each node of the cluster stores 10 GB of data. There are no running jobs while the MR-clusters are resized.

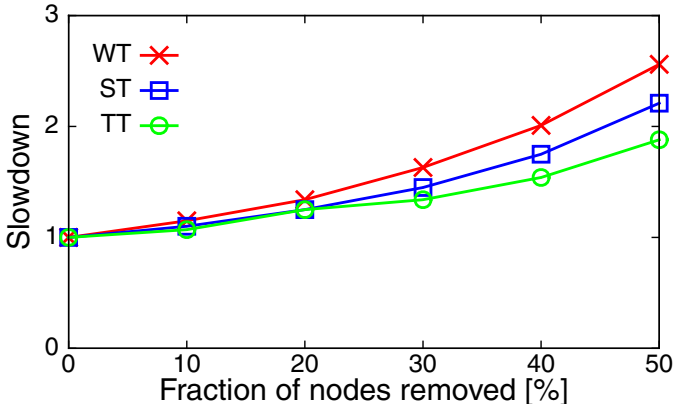
We find the average per-node removing time is constant when the MR-cluster is shrunk with up to 50% of its total size, and increases exponentially for larger fractions of transient-core nodes removed, as more data are replicated on fewer nodes (Figure 3.5(a)).

When shrinking an MR-cluster at runtime, the job runtime is determined by the total size of the replicated data. We set up 20-node MR-clusters with 10 core nodes which we shrink at runtime by different fractions of transient-core nodes. Figures 3.5(b) and 3.5(c) show that shrinking MapReduce applications at runtime increases the job slowdown linearly with the number of transient-core nodes removed. However, we observe that less compute-intensive jobs (e.g., ST and WT), which run on 200 GB, have higher slowdown than more computational intensive jobs (e.g., TT, PR), which run on less than 100 GB.

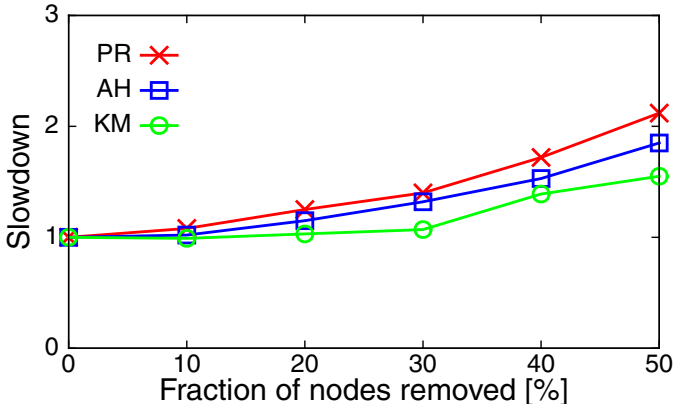
As MapReduce is usually employed for data-intensive applications, it is important to reduce the overhead of data replication by limiting the frequency of MR-cluster reconfigurations and by removing nodes with smaller data volumes.



(a) Data replication



(b) Single MapReduce computation



(c) Multiple MapReduce computations

Figure 3.5: The impact of data replication: the average per-node shrinking time of MR-clusters with N nodes in total (a). The job slowdown relative to the execution time measured on a static 20-node MR-cluster, when shrinking the MR-cluster with different fractions of core nodes for jobs that run a single MapReduce computation (b) and for jobs that chain multiple MapReduce computations (c).

Table 3.2: The design space coverage of the macro-experiments presented in Section 3.6. For each experiment, the table summarizes the provisioning policy (node type and weighting policy) employed by the resource manager, and the workload instances (application type and job types, sizes, and arrival pattern) submitted to three concurrent MR-clusters.

					Job Types			Job Arrivals		
Sec.	Wkld.	Nodes	Weight	Apps.	C_1	C_2	C_3	C_1	C_2	C_3
3.6.1	A	TR	JD	WT	50 x small			batch		
3.6.2	B	all	TD	WT	90 x small	5 x medium	5 x large	batch		
	C	all	TD	ST	90 x small	5 x medium	5 x large	batch		
3.6.3	D	TC	all	WT, ST	165 x small	188 x all	555 x small	average	bursty	average
	E	TC	TD	WT, ST, BT	359 x all	26 x all	559 x small	average	sequential	average

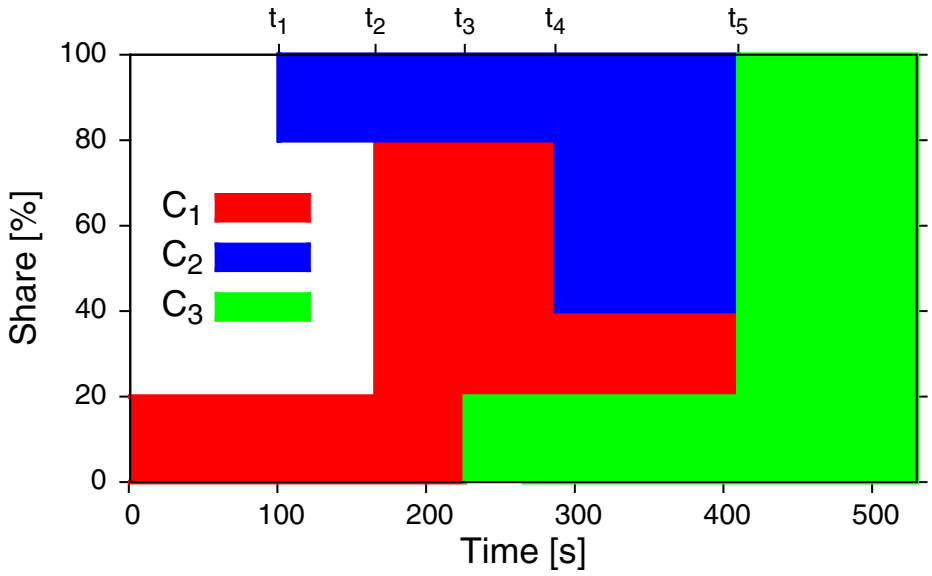


Figure 3.6: The distribution of the resources across three MR-clusters (C_1, C_2, C_3) arriving at different moments in time. Points $t_1 - t_5$ are explained in Section 3.6.1.

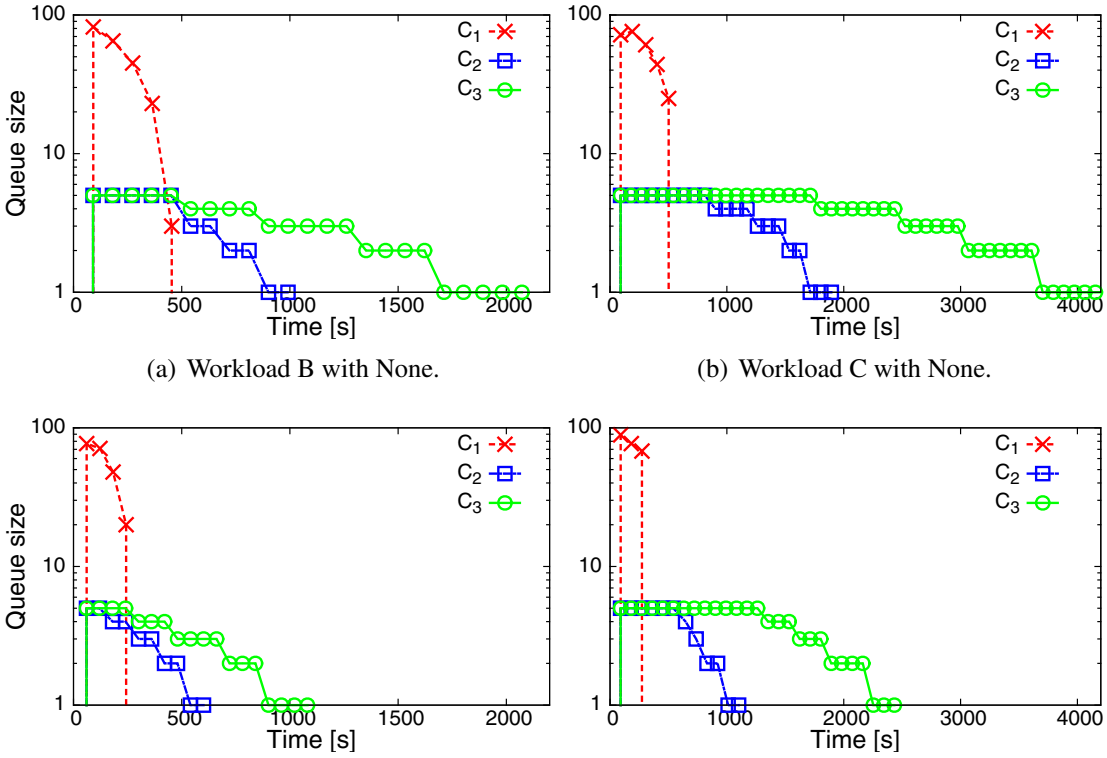
3.6 Macro-Experiments

In this section we evaluate FAWKES's resource provisioning and balancing mechanisms. Towards this end, we design a comprehensive set of scenarios, summarized in Table 3.2 w.r.t. to both system operation (e.g., nodes and weights) and experiment instrumentation (e.g., applications and workloads). We show how FAWKES effectively provisions newly arriving MR-clusters (Section 3.6.1) and achieves good balancing when the workloads are imbalanced (Section 3.6.2). Moreover, even under extreme imbalance and unfavorable conditions, we show evidence of up to 25% improvement of average job slowdown (Section 3.6.3).

3.6.1 Arriving MR-clusters

In this section we show how FAWKES balances idle resources across the active MR-clusters and gracefully shrinks them to make space for new MR-cluster deployments.

FAWKES effectively uses its grow and shrink mechanisms to dynamically provision multiple arriving MR-clusters. Given 60 resources, FAWKES receives requests for 3 MR-clusters, at intervals of 100 seconds. All MR-clusters store 50 GB of data on their minimum shares of 10 core nodes. FAWKES uses transient (TR) nodes and employs the JD weighting policy (see Section 3.3.2). The weights are updated every $T = 60$ s. We combine 3 instances of the SS workload (see Section 3.4.3) into workload A (see Table 3.2).



(a) Workload B with None.

(b) Workload C with None.

(c) Workload B with TR-EQ.

(d) Workload C with TC-EQ.

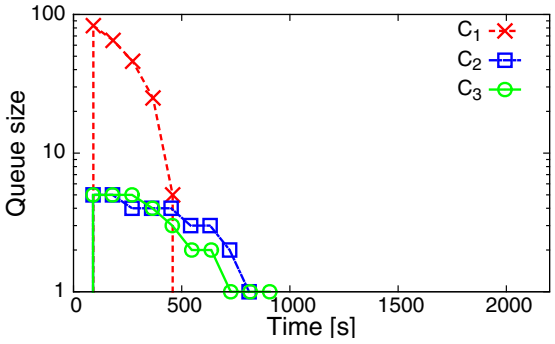
Figure 3.7: The queue sizes of three MR-clusters running workloads B and C (Table 3.2) for the baselines.

In Figure 3.6 FAWKES initially provisions 10 core nodes to C_1. While C_1 loads its data, C_2 arrives and receives 10 core nodes from remaining 50 idle resources (point t_1). MR-cluster C_1 starts running jobs when C_2 is still loading the data. Thus, all 40 remaining resources are allocated to C_1 (point t_2). Later, MR-cluster C_1 is successively shrunk to make space for C_3 (point t_3) and to allow the share of C_2 to grow (point t_4). When both C_1 and C_2 finish their workloads, C_3 grows to the full capacity of the system (point t_5).

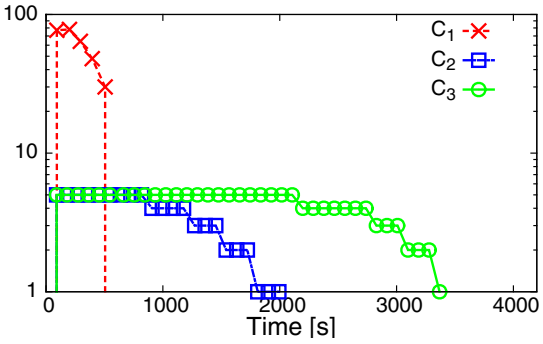
With a static partitioning approach, when the system is fully utilized, requests for new MR-clusters need to wait for active MR-clusters to complete their workloads and release the resources. Dynamic provisioning allows new MR-clusters to be deployed even when the active MR-clusters use the entire system capacity.

3.6.2 Growing and Shrinking MR-clusters

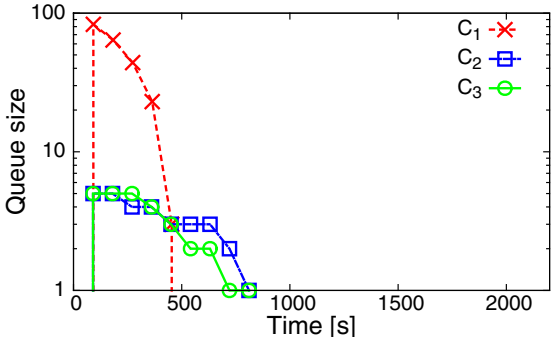
In this section we show the impact of the type of nodes (transient or transient-core) and the type of workload on FAWKES's balancing mechanism.



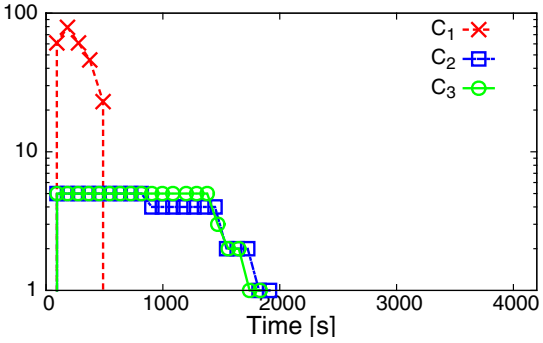
(a) Workload B with TR-TD.



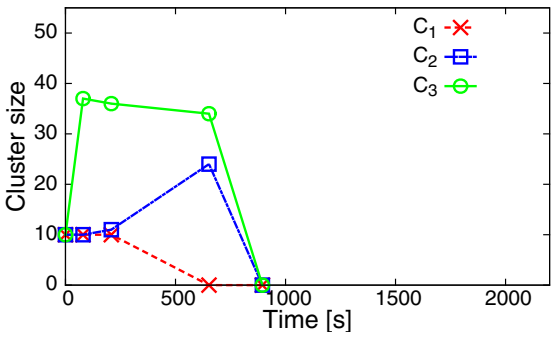
(b) Workload C with TR-TD.



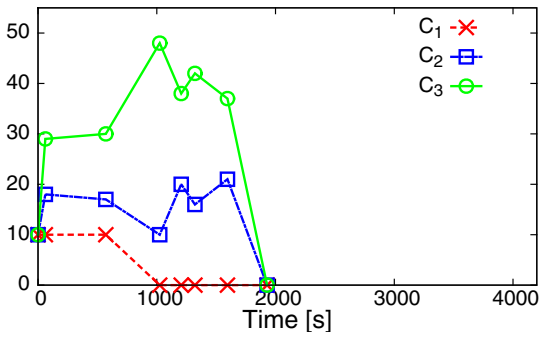
(c) Workload B with TC-TD.



(d) Workload C with TC-TD.

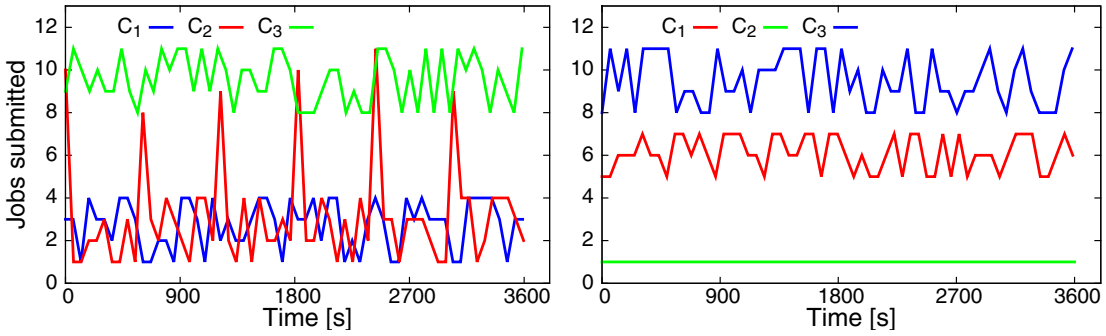


(e) Workload B with TC-TD.



(f) Workload C with TC-TD.

Figure 3.8: The queue and cluster sizes of three MR-clusters running workloads B and C (Table 3.2) for different growing and weighting types.



(a) Workload D (b) Workload E

Figure 3.9: Arrival processes (numbers of submissions per 2-minute intervals) for the two highly imbalanced workloads D and E (see Table 3.2).

FAWKES is able to balance the allocations for disk-intensive workloads with TC growing and DP shrinking, but fails to do so when using TR growing and FP shrinking (all defined in Section 3.3.2).

We consider 60 resources in total used by FAWKES to deploy simultaneously 3 concurrent MR-clusters with 10 core nodes. We submit workloads B and C which differ by the growing (TC or TR) and the application (WT or ST) type (see Table 3.2). The weights are updated every $T = 60$ s. In Figures 3.7 and 3.8, we show the queue sizes of the three MR-clusters over time for None, EQ and TD policies (all defined in Section 3.3.3). Apparently, neither the queue size nor the makespan of MR-cluster C_1 with small jobs is affected by growing or shrinking. Whereas FAWKES balances the medium and large WT workloads with both TR and TC nodes (MR-clusters C_2 and C_3 in Figures 3.8(a) and 3.8(c)), the mechanism is not effective for ST-based workloads with TR nodes (MR-cluster C_3 in Figure 3.8(b)). For the latter scenario, the data volumes shuffled by the large ST jobs (MR-cluster C_3) increase the execution overhead as we have shown in Section 3.5. With TC growing, FAWKES balances the allocations even for highly disk-intensive workloads (MR-clusters C_2 and C_3 in Figure 3.8(d)).

Without the dynamic growing and shrinking, the resources released once an MR-cluster executes its workload remain idle. Instead, FAWKES allocates the unused capacity to provision the active MR-clusters according to their weights, thus reducing the imbalance and the makespan.

3.6.3 Weighting MR-clusters

In this section we assess the balancing properties of FAWKES in two scenarios with three MR-clusters running extremely imbalanced workloads.

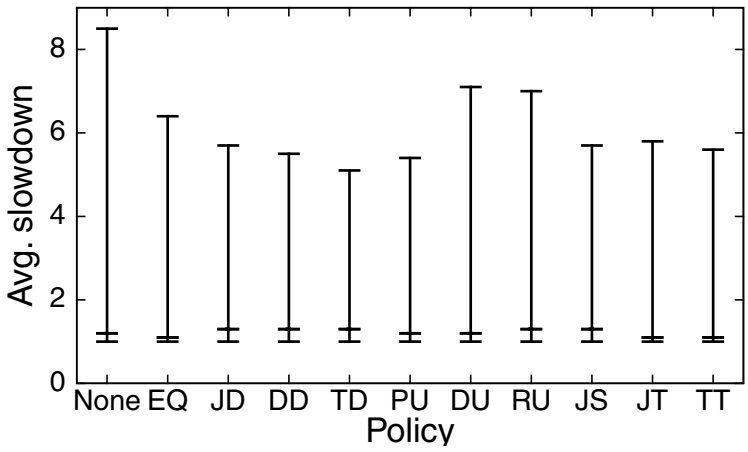
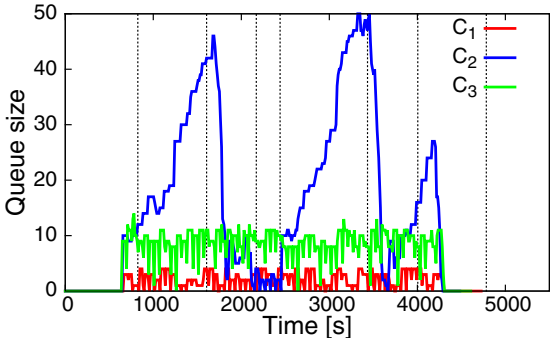


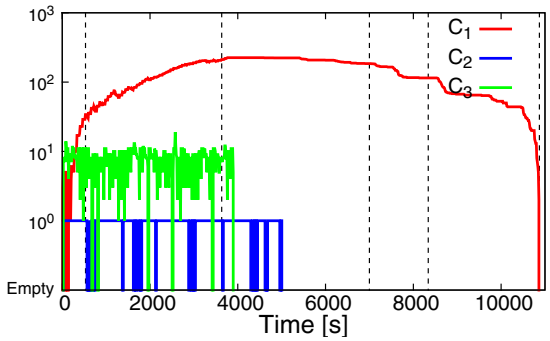
Figure 3.10: The minimum, the maximum, and the median of the average job slowdowns of three MR-clusters with workload D for all weighting policies.

FAWKES balances the allocations to workloads even in very unfavorable situations. In both scenarios, we reserve 48 resources to deploy simultaneously three MR-clusters with 10 core nodes and 200 GB of data each. FAWKES updates the weights of the MR-clusters every $T = 120$ s and provisions them with TC nodes. The two scenarios we analyze use workloads D and E (Table 3.2) with the arrival processes depicted in Figure 3.9. Workload D mixes WT and ST jobs in two MS instances submitted to C_1 and C_3 and one MM instance submitted to C_2 and has bursts with large jobs in cluster C_2. Workload E submits the complete BTWorld workflow to C_2 as an MM instance and mixes WT and ST jobs submitted to C_1 and C_3 as MM and MS instances, respectively, with C_3 permanently having a much higher load than C_1 and C_2. Both workloads are imbalanced, with the ratio between the average number of tasks executed in the clusters with the highest and lowest loads being 3 and 8, respectively.

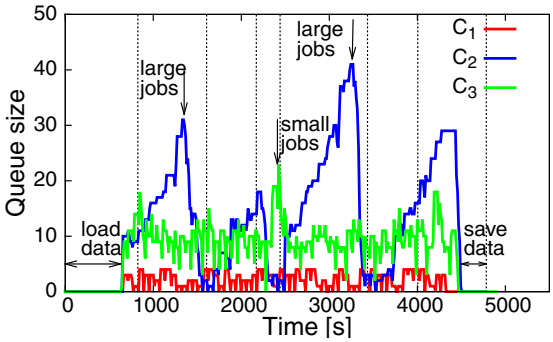
In Figure 3.10 we compare in the first scenario the weighting policies w.r.t. the average job slowdown measured for each MR-cluster running workload D. For the demand-based policies (JD, DD, TD), the finer the granularity of calculating the queue sizes (number of tasks with TD) is, the more balanced the workloads of the MR-clusters are (25% improvement of job slowdown compared with None). FAWKES achieves the best improvement of the average job slowdown of 25% with the TD policy. Furthermore, we observe that FAWKES reduces the average job slowdown in the most loaded cluster (C_2) without significant impact on the performance of the low demand clusters (C_1 and C_3). From the usage-based policies (PU, DU, RU), only PU performs reasonably well. RU ($\psi = 0.5$) and its derivative DU ($\psi = 0$) are not effective because of some small jobs which generate large amounts of data, yet are completed relatively fast (see ST in Figure 3.2(a)). Counterintuitively, the performance-based policies (JS, JT, TT) do not outperform the demand-based



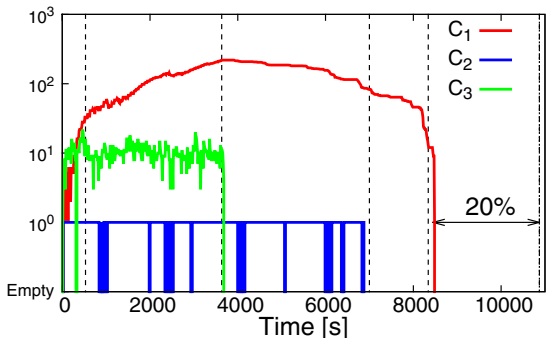
(a) Workload D with EQ weighting.



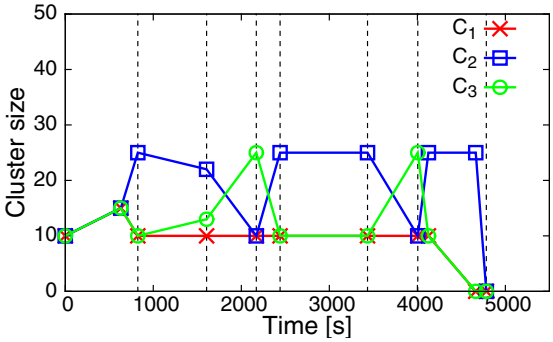
(b) Workload E with EQ weighting (logscale).



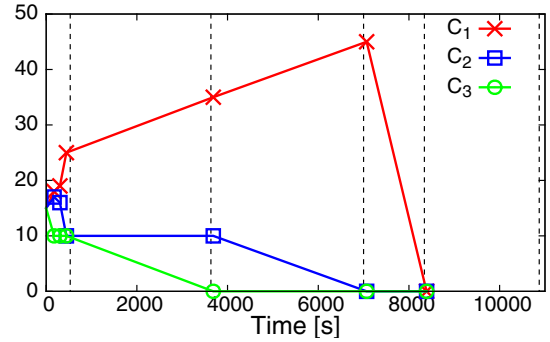
(c) Workload D with TD weighting.



(d) Workload E with TD weighting (logscale).



(e) Workload D with TD weighting.



(f) Workload E with TD weighting.

Figure 3.11: The performance of TD weighting for workloads D and E.

policies. The reason is that in workload D, small WT and ST jobs, which have the same runtimes (see Figure 3.2), prevail. Therefore, the weights assigned with the performance-based policies are similar and do not reflect the actual imbalance between the workloads.

Figure 3.11 shows the queue and cluster sizes with the weighting policy that performs best in the first scenario, TD, and the EQ baseline, for scenario 1 and 2, respectively. FAWKES makes 16 and 5 reconfigurations (excluding MR-cluster deployments and deallocations) in the two scenarios, see Figure 3.11(e) and 3.11(f). For workload D we notice that MR-clusters are reconfigured when bursts of large jobs arrive at C_2 or many small jobs are submitted to C_3. For workload E the most loaded MR-cluster (C_1) acquires most of the system resources, thus reducing the overall makespan of the experiment. In both cases, FAWKES is as effective as it can be because it moves (almost) all resources that it can move to the cluster with the highest load (C_2 in scenario 1 and C_1 in scenario 2), almost always leaving the other clusters at their minimum shares.

3.7 Related Work

In this section we summarize the related work from four aspects: resource sharing mechanisms for multicluster environments, malleability of parallel applications, cluster sizing problems, and fair-sharing provisioning policies.

To simplify cluster programming, a diverse array of specific frameworks for big data processing has been developed. Having multiple such frameworks within the same cluster or datacenter infrastructure requires a high-level resource management layer. Closest to our work are Yarn [127], Mesos [63], and KOALA [85], all having the same design goal of sharing resources between multiple cluster computing frameworks. Yarn considers the resource requests received from the applications, the resource availability, and the internal scheduling policies to dynamically allocate resources to different frameworks. Heavy users or large jobs are prevented from monopolizing the cluster through system partitioning which can be employed with the Capacity [11] and FAIR schedulers [144]. Mesos takes a different approach to resource allocation by delegating the scheduling decisions to the frameworks, rather than to individual applications. To reduce the overhead of the distributed scheduling, Mesos allows frameworks to specify through filters which are the resources they will never accept. KOALA is a resource manager that co-allocates processors, possibly from multiple clusters, to various HPC applications and to isolated MapReduce [45] frameworks. When resources are available, each framework may receive additional resources from KOALA, but it is their decision to accept or reject them.

FAWKES is fundamentally different from Yarn and Mesos. While the latter target near-optimal data locality, FAWKES achieves performance isolation and balanced resource allocations by relaxing the strict data locality assumptions through a fast and reliable grow and shrink mechanism. Instead of the request and offer-based approaches, FAWKES

controls the scheduling by observing the jobs submitted within each framework. Although the FAWKES mechanism can be incorporated into KOALA, we are currently maintaining and using it as a standalone research prototype.

To improve resource utilization, jobs that can be executed on a variable number of processors have emerged. When the number of processors assigned to a job can be increased or decreased by the scheduler at runtime, the job is called malleable [20, 40]. There are two ways to enable job malleability in parallel applications, either by creating a large number of threads equal to the cluster capacity, coupled with a multiplexing mechanism, or by inserting application specific code at synchronization points to repartition the data when the allocation changes. The poor performance of the former approach and the additional coding effort of the latter have limited the popularity of exploiting the job malleability for (tightly coupled) parallel applications. However, certain parallel applications based on the master-slave programming model, in which processors are required to execute relatively small and independent units of computations from a central scheduler (e.g., MapReduce [31]), can use malleability relatively easy. Moreover, MapReduce applications can be accelerated by using Amazon's spot instances, despite their variable and unknown lifetime [25].

Novel in this chapter, instead of dynamically changing the allocations of single jobs [20], we exploit malleability of sets of MapReduce jobs, by growing and shrinking the framework itself. To do so, we propose a data-oriented mechanism in order to gracefully remove nodes from MapReduce frameworks at runtime. Towards this goal, we relax the traditional data locality constraints and we provision the MapReduce frameworks with temporary nodes that retrieve their input data for the tasks they run from core nodes.

Cluster sizing problems are notoriously difficult because of the large parameter space related to the number of resources, the type of resources, and the job configuration. GreenHadoop [49] is mainly powered by a solar energy source and uses the electrical grid only as backup. The scheduler minimizes the consumption of electrical energy by allowing MapReduce jobs to use more resources when green energy is available and less resources during peaks of electrical energy costs. Elastizer [62] provides an offline automated technique to solve cluster sizing problems using job profiles and simulations to search through the parameter space. Datacenters may benefit from a dynamic right-sizing mechanism that limits the number of active servers during periods of low load. Towards this end, optimization-based models and corresponding online algorithms for capacity provisioning in power proportional datacenters have been proposed and analyzed in [80, 133].

Although we do not optimize for energy consumption, we investigate a similar problem of dynamic right-sizing a MapReduce cluster, but in a different setting. FAWKES attempts to find the fair share of each MapReduce cluster relative to the service levels of other concurrent clusters.

Fair-sharing algorithms have been explored in networking and operating systems domains for decades (see [144] and references therein). Datacenter schedulers like FAIR [144] and Quincy [69] provide fairness for a single resource type by maximizing the minimum allocation (*max-min fairness*) received by each user in the system. To provide fairness in more general settings in which jobs may have heterogeneous requirements and hard placement constraints, the max-min fairness model has been extended to support multiple resource types [47] and to satisfy users constraints [48]. Pisces [106] is a datacenter scheduler that isolates the performance of multiple users of a shared key-value storage and provides max-min fairness. Pisces employs weighted fair-sharing and combines complementary mechanisms (partition placement, weight allocations, replica selection, and weighted fair queuing) which operate on per-application requests. A general framework that enables weighted proportional allocations for user differentiation is analyzed from a theoretical perspective in [88].

Unlike the former schedulers, FAWKES operates at the framework level, maintains a global view of the system by observing the jobs during their lifetime, and assigns to each framework a dynamically changing weight. In this chapter we propose three elements to differentiate MapReduce frameworks at runtime, viz. based on demand, on usage, and on performance.

3.8 Conclusion

Isolating the performance of multiple time-varying MapReduce workloads is an attractive yet challenging target for many organizations with large-scale data processing infrastructures. Towards this end, we have presented FAWKES, a mechanism for balancing the allocations of multiple MapReduce instances such that they experience similar service levels. FAWKES is based on the MR-cluster, a new abstraction for deploying MapReduce instances on physical resources which assumes the usual data locality constraints for a set of core nodes, but relaxes these constraints for nodes outside the core. For the fair-sharing problem, FAWKES employs weighted proportional allocations. The specific provisioning policies assign dynamic weights to different MR-clusters that take into account their dynamic load conditions.

In this chapter we have taken an experimental approach to provisioning multiple MR-clusters in a datacenter or multicluster system. With our micro-experiments we have found that a relaxed data locality model has a limited impact on the application performance. Furthermore, our macro-experiments have showed that FAWKES delivers good performance and balanced resource allocations, even in unfavorable conditions of highly imbalanced workloads.

Chapter 4

Size-based Resource Allocation in MapReduce Frameworks

4.1 Introduction

Data-processing frameworks such as MapReduce that can be used for large-scale analytics and small interactive queries may face workloads of jobs with heavy-tailed processing requirement distributions. As has been abundantly clear both from theoretical analysis of queueing systems [55, 56] and from experience with actual deployments of MapReduce and other frameworks [116, 144], such workloads usually lead to job slowdowns of small jobs that are at least an order of magnitude larger than those of long jobs, which may be intolerable to users. In this chapter we present the design and analysis of TYREX[1], a MapReduce scheduler that aims at *reducing the slowdown variability* in workloads with many short jobs.

Despite the plethora of performance related studies of data-intensive workloads, state-of-the-art MapReduce schedulers still lead to high slowdown variability. In our experience with processing monitoring data from the BitTorrent global network using a MapReduce-based logical workflow [61], we have found that 15% of the jobs account for 80% of the total load, and that 65% of the jobs complete in a minute. Similarly, several studies on the performance of modern production clusters in commercial companies like Google and Facebook [23, 73, 97] report *heavy-tailed* workloads with highly variable runtimes. Allowing such workloads to run in non-isolated environments and to greedily share the system resources severely impacts the performance of short jobs as they experience long delays due to large jobs ahead of them.

In Figure 4.1 we illustrate the scheduling model employed by the TYREX scheduler. TYREX uses *resource partitioning* and *work-conserving job migration* across these par-

[1] Inspired by the dinosaur Tyrannosaurus rex, known for its long, heavy tail.

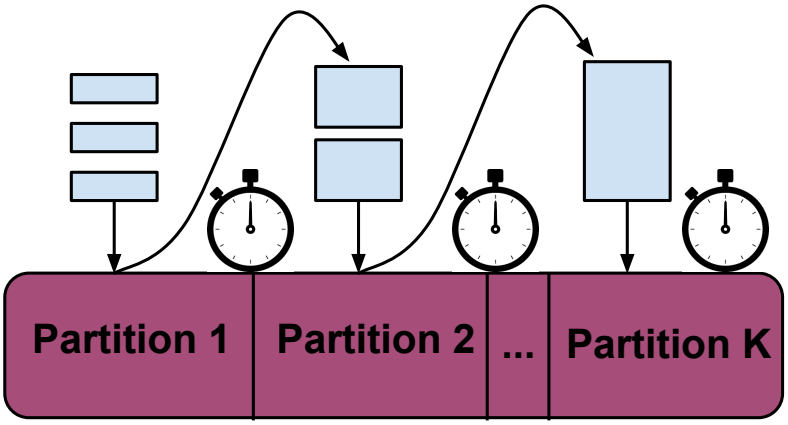


Figure 4.1: Schematic overview of the queueing model employed by TYREX with resource partitioning and timers for migrating jobs.

titions as its two main principles. A common way of partitioning the resources of a datacenter is to allocate disjoint sets of machines to multiple instances of the MapReduce framework [46]. However, this scheduling model is not attractive for jobs that are moved across partitions but still require access to the same data, as the cost of replicating the data across partitions may be prohibitive. Instead, TYREX operates within a single MapReduce framework so that jobs running in any partition may read data stored on any machine.

For isolating the sets of jobs with processing requirements (their *sizes*) in different ranges, TYREX imposes runtime limits (*timers*) with increasing values which limit the amounts of processing time jobs may receive in the partitions—jobs that exceed the timer of one partition are migrated to the next, retaining the work they have already completed. As fixed timers are difficult to configure because they require trial executions of every new workload to find their optimal values, we propose a method to *dynamically adapt the timers* based on statistical properties of the job size distribution. These are the main principles behind TYREX, a scheduler that is biased towards short jobs and that reduces the variability of the job slowdown significantly when compared to standard MapReduce schedulers like FIFO and FAIR.

TYREX is inspired by the TAGS policy [55], but there are a few important differences. First, as MapReduce frameworks are usually deployed in datacenters, TYREX has to divide the framework capacity across the partitions, rather than having predetermined single-server partitions as in TAGS. Secondly, MapReduce jobs are conveniently parallel jobs that only require synchronization between the map and reduce phases, which makes them malleable or elastic with the opportunity to run multiple jobs simultaneously as opposed to the rigid job model supported by the FIFO servers in TAGS. Finally, having a shared underlying distributed filesystem for all partitions of a MapReduce framework,

TYREX enables a work-conserving mechanism for moving a job from one partition to the next without losing the intermediate results, as happens with TAGS.

In this chapter we make the following contributions:

1. We design TYREX, a datacenter MapReduce job scheduler that places jobs across multiple partitions of a single MapReduce framework in order to isolate the sets of jobs of very different sizes. TYREX assumes no prior job size information and moves jobs across system partitions without losing their completed work (Sections 4.4.1 and 4.4.2).

2. We incorporate in TYREX two policies to migrate jobs across partitions that are differentiated by the type of timer they use in partitions, which may be *fixed* or *dynamic*. To adapt the timers dynamically, we propose a statistical technique to identify jobs that are likely to monopolize a given partition for a long time (Sections 4.4.3 and 4.4.4).

3. With a set of experiments in a real multicluster system, we evaluate TYREX relative to standard MapReduce schedulers and assess the impact of different aspects of its operation on the job slowdown variability. We show that our scheduler delivers very low slowdown variability without a large impact on the median slowdown for several representative MapReduce workloads (Sections 4.5 and 4.6).

4.2 Problem Statement

In this section we explain the problem of job slowdown variability in MapReduce workloads and we formulate the goals of our scheduler. To capture the delay sensitivity of jobs of different sizes, we consider the *job slowdown* as a metric, defined for a job as the ratio of its response time and its wall-clock time in an empty system – when a fixed set of resources (in our case, the entire capacity of the system) are allocated to it. To formulate the goal of our scheduler, let F be the cumulative distribution function of the job slowdown when executing a certain workload. We define the *job slowdown variability at the q^{th} percentile*, denoted by $V_F(q)$, as the q^{th} percentile of F normalized by the median job slowdown, that is:

$$V_F(q) = \frac{F^{-1}(q)}{F^{-1}(50)}. \tag{4.1}$$

We call $V_F(95)$ the *overall job slowdown variability* of the workload. Our target is to *minimize* both the median job slowdown and the overall job slowdown variability of MapReduce workloads.

We consider job slowdown as a more fundamental metric than response time, and the problem of large job slowdowns as a more fundamental problem than large response times. The latter are continuously being improved by better hardware, leading to higher speedups of applications and faster data transfers. In contrast, even though there may be shifts in the balance of the speeds and capacities of computer systems hardware, (large) job slowdowns will continue to exist in systems with contention for resources.

Many MapReduce clusters execute workloads that are often characterized by heavy-tailed processing requirement distributions [23, 73]. These workloads contain jobs with the amount of data to be processed or the sum of the execution times of all tasks of a job, that may vary by several orders of magnitude. In systems with such workloads, the small jobs suffer because they may be delayed for a relatively very long time due to long jobs ahead of them [144]. Nevertheless, the users of clusters or datacenters may expect their jobs to be delayed proportionally to their processing requirements.

In this chapter we will generalize the TAGS policy to scheduling MapReduce workloads with heavy-tailed job-size distributions running in partitioned datacenters with each partition having a runtime limit. We will show that this way of scheduling MapReduce jobs outperforms FAIR, the most popular MapReduce scheduler, with respect to both the median slowdown and the slowdown variability for a broad range of job size distributions.

4.3 Size-based Scheduling

In this section we present an overview of the main scheduling disciplines that optimize the mean response time or the mean job slowdown in both single-server and distributed-server systems. In Table 4.1, we show the main characteristics of several policies which have been investigated in the past. In single-server systems, policies that are biased towards short jobs, also known as SMART policies [136], are to be preferred as they prevent short jobs from experiencing long delays. The fundamental idea behind these policies is to prioritize short jobs over longer ones like Preemptive-Shortest-Job First (PSJF [135]) and Shortest-Remaining-Processing-Time (SRPT [59]). Although SRPT is optimal with respect to mean response time, the policy is rarely used in practice as it may lead to starvation of long jobs in order to help the short ones [56]. Another reason for the limited popularity of SRPT is that its effectiveness relies on preemption, which is difficult to implement for long jobs that can easily overflow the memory [57]. Instead, in supercomputers where jobs may be served by multiple hosts, size-based partitioning is often employed to isolate the performance of jobs with very unbalanced processing requirements [26].

When the job size distribution and the individual sizes of (rigid) jobs are known, the servers of a distributed-server system using the FIFO policy can be configured to serve each only the jobs whose sizes are in a specific range (the Size Interval Task Assignment policy – SITA [56]). It can be shown that when the size ranges are chosen in such a way

Table 4.1: A policy framework for optimizing job slowdown in different system models.

Policy	System	Job-size	Preempt	Waste	Utilization
PSJF [135]	single-server	known	yes	none	best
SRPT [59]	single-server	known	yes	none	best
SITA [57]	distr. servers	known	no	high	worst
TAGS [55]	distr. servers	unknown	no	high	worst
FAIR [12]	datacenter	unknown	yes	low	high
TYREX	datacenter	unknown	yes	none	high

that all servers have the same load, the SITA policy is optimal in terms of the average job slowdown if the job-size distribution is heavy-tailed. The intuition behind this result is that in SITA, the job-size variability of each server is very much reduced. The SITA policy can be generalized for unknown job sizes through a simple yet very efficient technique that guesses the job sizes by killing them when they exceed the maximum runtime of a server and restarting them on the server with the next higher runtime range (the Task Assignment based on Guessing Sizes (TAGS) policy [55]). Traditionally, these have been successfully implemented in distributed-server systems, but they have not been used so far in clusters or datacenters for fear that they may lead to system fragmentation and underutilization of resources.

4.4 The Tyrex Scheduler

In this section we present a scheduling model to reduce the job slowdown variability in MapReduce frameworks. Our scheduler, TYREX, assumes no prior knowledge about the jobs, divides the MapReduce framework computing capacity in disjoint partitions, and migrates jobs across those partitions. We propose two policies, STATICTAGS and DYNAMICTAGS, used by TYREX to confine jobs with similar processing requirements to separate partitions.

4.4.1 Design Considerations

TYREX resembles the structure of the TAGS policy, but there are three key elements in which it is different. First, TAGS was designed for a distributed-server model in which each host is a single multi-processor machine that can only serve one job at a time. In contrast, TYREX targets a datacenter environment in which the system capacity is divided across partitions with many resources. As a result, instead of only having the timers as parameters as in TAGS, in our model we also have the partition capacities as parameters.

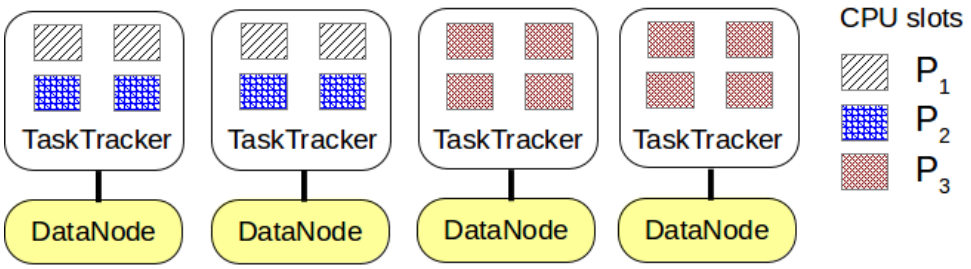


Figure 4.2: The fine-grained resource partitioning employed by TYREX with CPU slots allocated across three partitions.

Secondly, TAGS assumes simple, rigid (sequential or parallel) non-preemptive jobs that may only run on a single host until completion. In contrast, our (MapReduce) job model is more complex as there are intra-job data precedence constraints (map before reduce) and data locality preferences (of map tasks), and as jobs are elastic (or malleable) and can run simultaneously, taking any resources they can get when it is their turn.

Thirdly, TAGS does not preserve the state of a job when it moves it from one server to the next. As a consequence, long jobs will get killed at every server except at the one where they run to completion, at every step losing all the work performed and thus wasting CPU time. Instead, TYREX takes a work-conserving approach by allowing jobs that are being moved from one partition to the next to retain their work and to gracefully resume their executions without redoing previously completed work.

4.4.2 System Model

The main design elements of our scheduler are *resource partitioning* and *migration of jobs* from one partition to another. Frameworks such as MapReduce employ a fine-grained resource sharing model, with processors divided into slots and with jobs divided into short tasks that run on slots. Therefore, we partition the compute slots of the MapReduce framework into disjoint partitions of fixed sizes, while keeping the storage accessible for all processors. This way of partitioning is attractive for jobs that may be moved across partitions but still require access to the same data, as there is no need of replicating the data across the partitions. TYREX operates within a single MapReduce framework and partitions the compute slots of this framework into some number K of partitions, each with its own queue of jobs with similar processing requirements, that all share access to the same underlying filesystem. TYREX allocates fixed capacities to its system partitions. The fraction of compute slots allocated to partition P_k represents its capacity and is denoted by $C_k, k = 1, 2, \ldots, K$. We assume jobs to be served in FIFO order in all partitions. In Figure 4.2 we give an example of a standard MapReduce framework which uses fine-

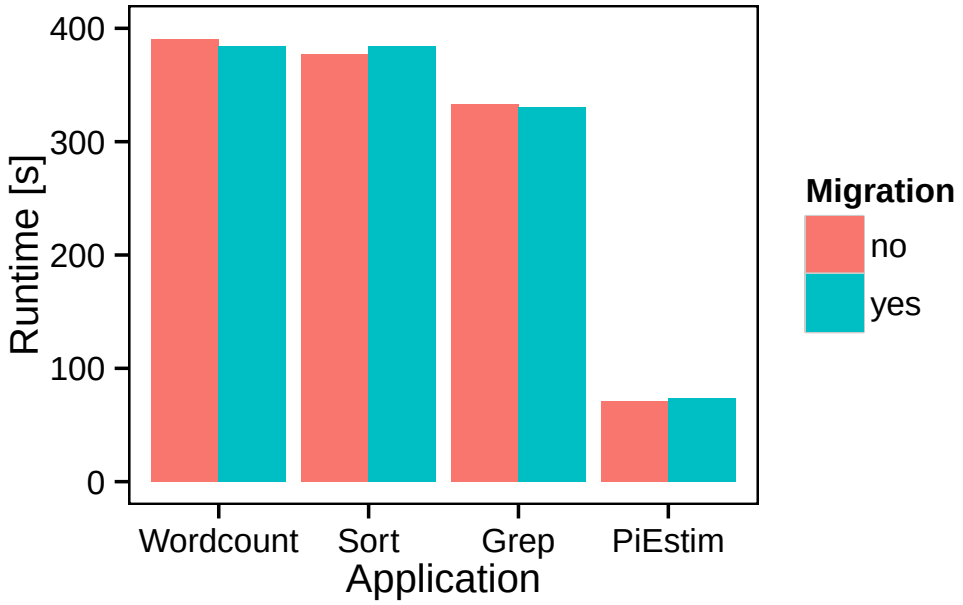
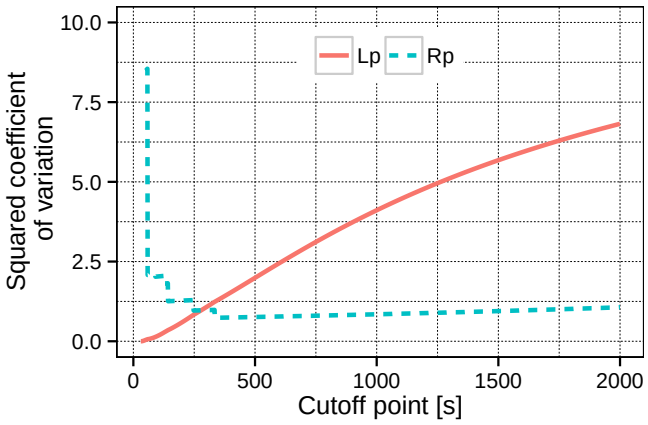


Figure 4.3: The job runtime *without* migration versus the job runtime *with* migration across two partitions in a 20-worker Hadoop framework.

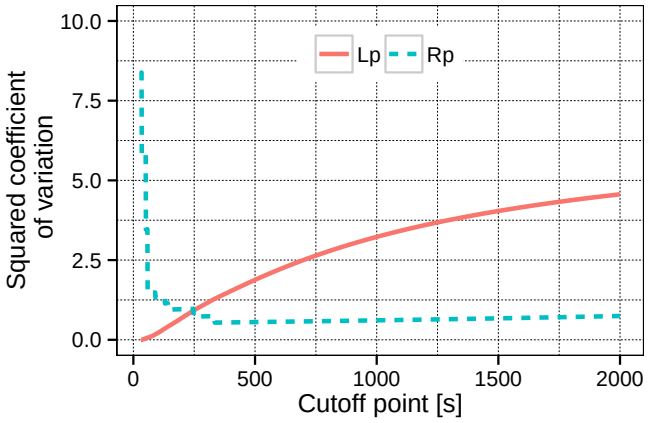
grained resource partitioning to split its CPU slots among three partitions ($C_1 = 25\%$, $C_2 = 25\%$, and $C_3 = 50\%$). All three partitions have access to the data stored in HDFS.

Job migration in TYREX is facilitated by the distributed filesystem that is shared across the whole framework, with the intermediate results of tasks executed within any partition being persistent and visible after a job has been moved to another partition. To avoid wasted work, the scheduler allows a job to finish its running tasks in a given partition after the timer has expired. As TYREX migrates jobs across partitions of a single framework, the cost of migrating a job across partitions is zero. In Figure 4.3 we assess the overhead of moving jobs across different partitions in a 20-node Hadoop framework. We set two equally sized partitions and we measure the job runtime *without* and *with* migration of four MapReduce applications (Wordcount, Sort, Grep, and PiEstimator) executing 600 map tasks and 60 reduce tasks. To execute a job without migration, we set the timer of P_1 to ∞. To migrate a job across the two partitions, we set the timer of P_1 to 50% of the job size. As expected, TYREX has no overhead in migrating jobs across partitions.

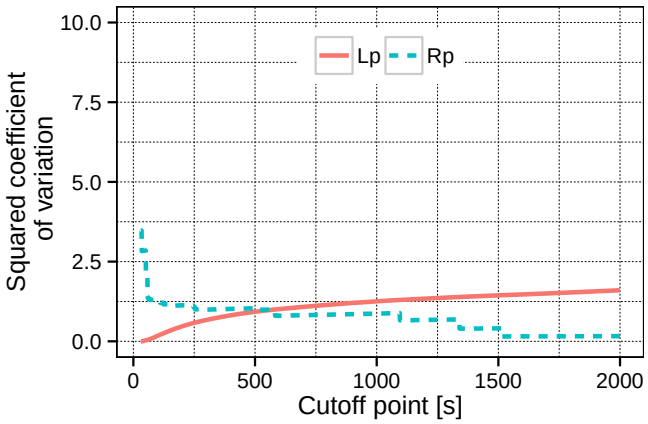
TYREX decides which jobs should be migrated from one partition to the next whenever a task of a job in any partition is completed. To do so, TYREX uses one of two policies, STATICTAGS and DYNAMICTAGS, which we present in the remainder of this section. For both our policies, we define the *current partial size* of a job in a given partition at time t as the total runtime of its tasks completed in that partition at time t.



(a) HVW



(b) MVW



(c) LVW

Figure 4.4: The squared coefficient of variation of L_p versus the squared coefficient of variation of R_p in our HVW, MVW, and LVW workloads for cutoff points p in the range between 30-2000 s.

4.4.3 The StaticTags Policy

As we assume that the processing requirements of jobs cannot be anticipated, we cannot adapt the SITA [56] policy to our situation, but we need a variation of the TAGS [55] policy in order to differentiate job sizes. Towards this end, we define for every partition $k, k = 1, 2, \ldots, K$ a timer T_k which is set to the total amount of service that jobs may receive while they are in partition P_k. Any job submitted to TYREX is first dispatched to partition P_1. When the partial size of a job in any partition P_k exceeds the timer T_k, TYREX moves the job to the next partition P_{k+1}.

As we are especially interested in workloads with heavy-tailed job size distributions and many short jobs, we can expect that the timer T_k of partition P_k should be set to a value that is (much) larger than the timer T_{k-1} of partition P_{k-1}, $k = 2, 3, \ldots, K$. We set T_K to ∞. However, setting the timers is difficult, and may in practice have to be repeated often. Unfortunately, the optimal timers of TAGS in distributed server systems already have complex forms even for well-behaved distributions like Pareto [55]. Hence, we take an experimental approach to determining the optimal timers with the STATICTAGS policy while keeping the partition capacities fixed.

4.4.4 The DynamicTags Policy

In this section we present the DYNAMICTAGS policy in which timers are dynamic, so the problem of setting their values disappears. We take a statistical approach to assessing the job size variability, which we apply to three realistic MapReduce workloads. The basic idea behind DYNAMICTAGS is to reduce the variability of the current partial job sizes in any partition by migrating those jobs that are likely to require significantly more processing time than the rest of the jobs in the same partition.

In order to present DYNAMICTAGS, we need the following definitions. Let X be a positive, real stochastic variable (e.g., corresponding to a heavy-tailed job size distribution), let CV be its coefficient of variation, and let p be a cutoff point in that distribution. Similarly to previous work [55,56], we measure the variability of the job size distribution using the squared CV, which is the ratio between the variance and the squared mean of X. We define $L_p = \min(X, p)$ and $R_p = X - p$ when $X > p$ as two random variables so that $X = L_p + R_p$ for any cutoff point p. To balance the variability across L_p and R_p, we seek a cutoff point p for which the values of the squared CV of L_p and R_p are equal, and we call that p the *optimal cutoff point* (we implicitly assume that there is a unique point with this property, which is always the case for our distributions).

To put the DYNAMICTAGS policy into perspective, we consider the distribution of job sizes in three workloads, i.e., HVW, MVW, and LVW, which we define in Section 4.5. As has been abundantly shown in recent studies, MapReduce workloads may have very variable job size distributions [24, 73]. In particular, the squared CV values of the HVW,

MVW, and LVW workloads are 20, 10, and 4, respectively. In Figure 4.4 we show the values of the squared CV of L_p and R_p for cutoff points p in each workload in the range of 30-2000 s. We see that increasing the value of the cutoff point has opposite effects on the variability in L_p and R_p, with the squared CV of L_p being maximum when the squared CV of R_p is minimum and vice versa. Of course, L_p (R_p) is close to the complete workload for very large (small) values of p, respectively.

More importantly, we find that the variabilities in L_p and R_p are unbalanced when the squared CV of L_p has a value that is higher than 2. As we are covering with our workloads a wide range of job size variabilities (see Table 4.2) for which 2 turns out to be a good value (see the results in Section 4.6.4), we conclude that having a squared CV higher than 2 in L_p is a good indicator of unbalanced job sizes. Hence, we aim for a squared CV in L_p that is lower than 2. Figure 4.4 also shows that the squared CV of R_p is flat and relatively low (below 2), which means that further splitting R_p, and so having more than two partitions, is not very promising. Indeed, several experiments we did with three partitions confirmed this conclusion.

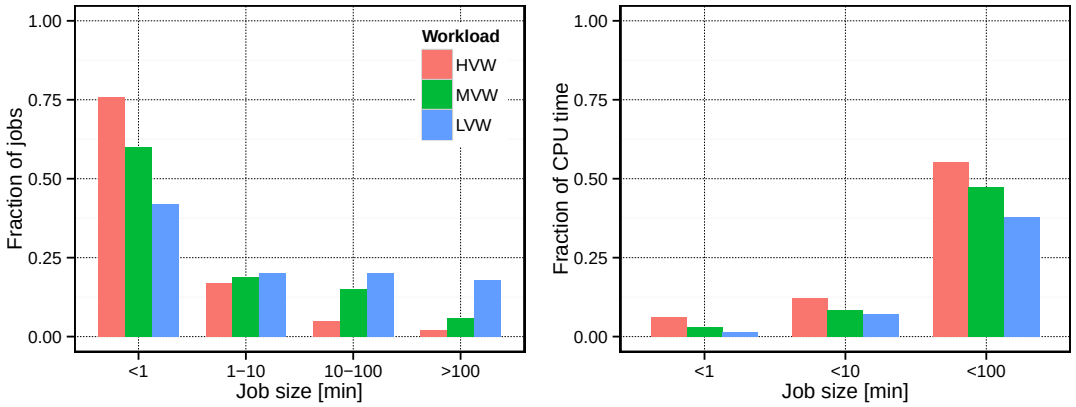
The DYNAMICTAGS policy now works in the following way. When it is invoked, it checks all partitions to see whether the value of the squared CV of the distribution of the current partial job sizes is higher than 2. If this does not hold for a partition, DYNAMICTAGS does not migrate any job from it. Otherwise, DYNAMICTAGS determines the optimal cutoff point in the distribution of the current partial job sizes, and uses that cutoff point as the value of the *dynamic timer* to migrate those jobs that exceed it to the next partition.

From a queueing-theory perspective, L_p captures the notion of young jobs, while R_p represents the residual lifetime of jobs. In particular, if the distribution of job sizes is heavy-tailed then the residual lifetime of young jobs is stochastically smaller than the residual lifetime of old jobs. As a consequence, the DYNAMICTAGS policy seeks in any partition P_k a cutoff point p so that P_k only serves young jobs that are likely to leave the system soon. In contrast, old jobs with larger residual lifetimes are migrated to the next partition.

4.5 Experimental Setup

In this section we present the workloads and the configuration of the infrastructure we use for the experimental evaluation of TYREX. To design our workloads we use a comprehensive set of representative MapReduce applications, including both standard benchmarks and complex, real-world workflows. In this chapter we take an experimental approach and we evaluate TYREX by means of experiments on the DAS multicluster system. The total time used for experimentation exceeded 20,000 hours system time.

Standard Benchmarks. The Hadoop distribution provides a set of synthetic benchmarks abundantly used in performance evaluation studies of MapReduce frameworks. We



(a) (b)

Figure 4.5: The job size distributions (a) and the sum of the CPU time of all jobs up to 1, 10, and 100 minutes as a fraction of the total CPU time (b) for all three workloads.

use the following applications from Hadoop: Wordcount, PiEstimator, Sort, and Grep. Wordcount and PiEstimator are CPU-intensive applications used to retrieve the number of occurrences of each unique word in a given set of input files and to estimate the real value of π using the quasi-Monte Carlo algorithm. In contrast, Grep and Sort are disk intensive applications used to search for a given pattern in the input files and to generate the content of the input files in non-decreasing order.

Complex Workflows. BTWorld is a complex and very challenging MapReduce-based logical workflow which we designed to observe the evolution of the global-scale peer-to-peer system BitTorrent [61]. The workflow consists of 26 MapReduce jobs with different resource bottlenecks (CPU, memory, or disk) and is used for processing monitoring data collected periodically from the BitTorrent system. We use nine BTWorld jobs which compute answers to several questions related to the operation of BitTorrent (e.g., *How does a tracker evolve in time?*, *How many active hashes are in the system?*, *Which are the most popular hashes and swarms?*). BTWorld jobs process a multi-column input dataset that contains timestamped, per-tracker statistics of the BitTorrent content: the number of users with fully and partially downloaded content and the number of completed user downloads.

MapReduce Workloads. In our experiments we will use three workloads that we create using the Wordcount, PiEstimator, Sort, Grep, and BTWorld applications and that are differentiated by their variability of job sizes. Each of these workloads consists of a stream of 300 jobs with a Poisson arrival process. In our evaluation of TYREX we designed all three workloads to impose an average load of 70%.

The three workloads we create are called HVW, MVW, and LVW for High/Medium/Low Variability Workload, and they have, going from one to the next, decreasing

Table 4.2: Summary of the execution of the HVW, MVW, and LVW workloads.

Statistics	HVW	MVW	LVW
Total jobs	300		
Squared CV	20	10	4
BTWorld jobs	33	45	10
Total maps	6,139	11,866	30,576
Total reduces	788	1,368	3,089
Temporary data [GB]	573	693	1,062
Persistent data [GB]	100	92	303
Total CPU time [h]	63.6	124.6	306.9
Total runtime [h]	3.51	3.98	5.31

squared CV values of their job size distributions. In Figure 4.5(a) we distinguish four ranges of job sizes to show the job size distributions in our workloads: “tiny" ($<$ 1 minute), “short" (1-10 minutes), “medium" (10-100 minutes), and “large" ($>$ 100 minutes). In all workloads there is a large fraction of tiny and short jobs combined (more than 60%). In Figure 4.5(b), we show the fractions of the total CPU time consumed by all jobs together with a cutoff point as defined in Section 4.4.4 of 1, 10, and 100 minutes, respectively. For example, even for the HVW workload, if the timer of partition P_1 is set to 10 minutes, the fraction of the load processed by partition P_1 is only 10%. Our workloads are representative for a broad range of computer systems workloads, having squared CV values in the range of 4 to 20 – the standard TPC benchmarks for evaluating the performance of computer systems exhibit squared CV values in this range [103]. As heavy-tailed workloads fit recent measurement of MapReduce production clusters [73], the HVW and MVW workloads with high variability of job sizes are more interesting in our evaluation.

To understand in more detail the structure of our workloads, we show in Table 4.2 execution summaries when all jobs are executed sequentially in an empty 20-node Hadoop framework. HVW spans more than 7,000 (map and reduce) tasks in total and requires more than 60 h CPU time. MVW has more than 12,000 (map and reduce) tasks in total and requires twice as much CPU time. The former two workloads are comparable in the total runtime and the amount of persistent data generated in an empty system. LVW consists of more than 33,000 (map and reduce) tasks, requires five times more CPU time than HVW, and generates three times more persistent data than both HVW and MVW.

DAS-4 Deployment. The DAS [118] cluster we use in our experiments has 20 dual-quad-core compute nodes with 24 GiB memory per node and 50 TB total storage. The DAS nodes are connected through 1 Gbit/s Ethernet (GbE) and 32 Gbit/s QDR InfiniBand (IB) networks. We use Hadoop (version 1.0.0) over InfiniBand and we configure at each node 6 map slots, 2 reduce slots, and 3 GiB memory per running task. The

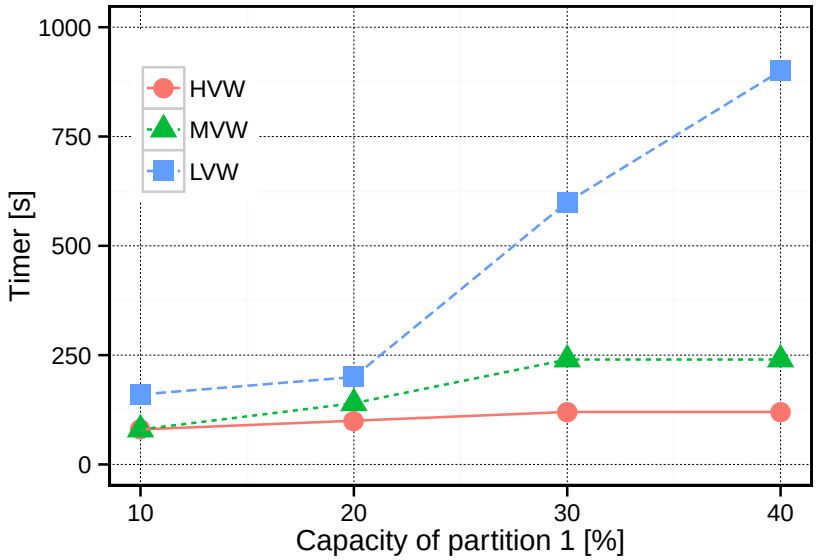


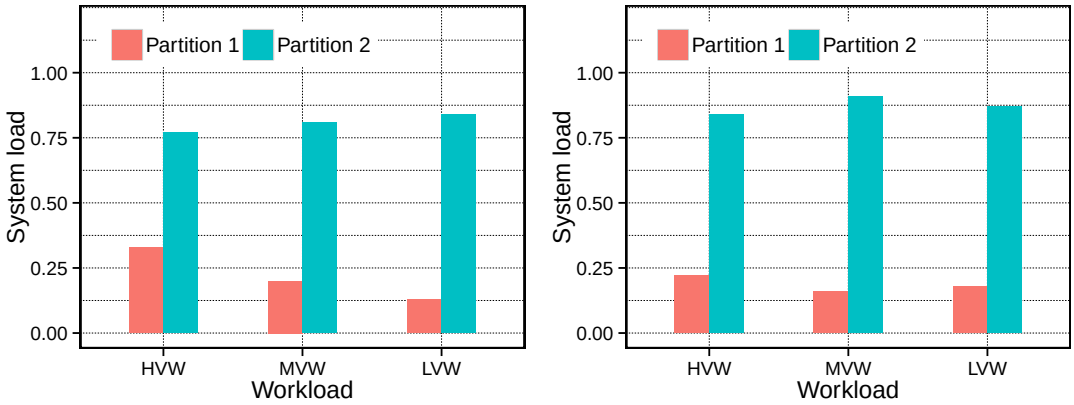
Figure 4.6: The optimal timer of partition P_1 for each capacity between 10-40% and for each workload under a system load of 0.7.

HDFS uses a virtual disk device with RAID 0 software and 2 physical devices with 2 TB storage in total per node.

Baseline Policies. We contrast TYREX with two state-of-the-art scheduling algorithms in data-intensive frameworks – FIFO and FAIR. The former is the default scheduler in Hadoop and assigns both map and reduce tasks using the first-in-first-out scheduling discipline. The later is widely used in deployment and is a discrete version of the weighted processor-sharing discipline which allocates slots to jobs proportional to the number of their tasks.

4.6 Experimental Evaluation

We evaluate TYREX using our prototype implementation in Hadoop on a 20-machine cluster. We investigate the setting of capacities and timers and the performance of the STATICTAGS policy in Sections 4.6.1 and 4.6.2. Further, we analyze the way the dynamic timers adapt over time and the performance of the DYNAMICTAGS policy in Sections 4.6.3 and 4.6.4. Finally, we compare TYREX with our two baselines, the FIFO and FAIR schedulers in Section 4.6.5.



(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

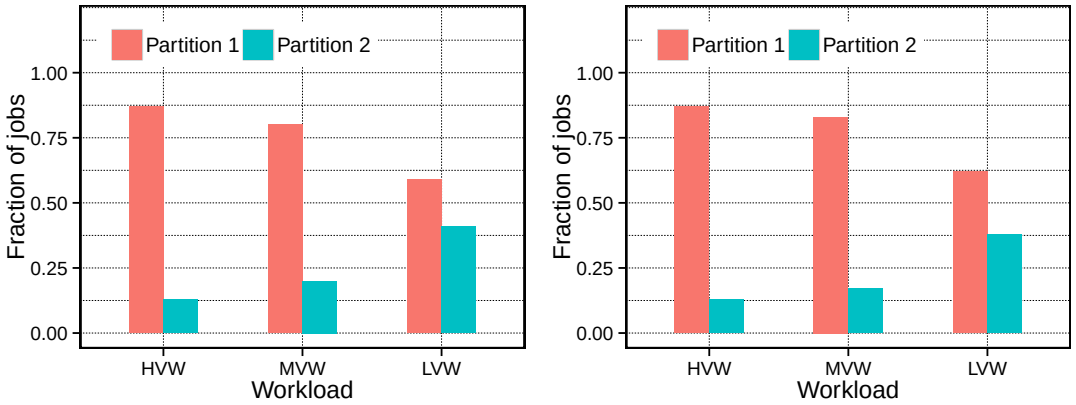
Figure 4.7: The utilizations of partitions P_1 and P_2 versus the workload variability with STATICTAGS under a system load of 0.7.

4.6.1 Setting Capacities and Timers

One of the issues in analyzing and deploying TYREX with the STATICTAGS policy is setting the number of partitions and the values of the partition capacities and timers. As we argued in Section 4.4.4, having more than two partitions is not worthwhile, so we consider having only two partitions.

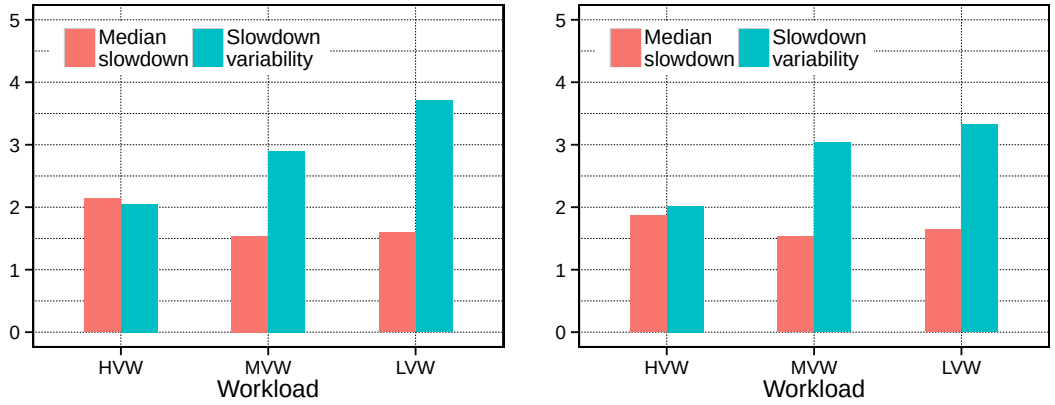
We will first investigate the relation between the partition capacity and the partition timer with the STATICTAGS policy for each workload. To this end, we seek to determine for each partition capacity its optimal timer, that is the timer that minimizes the overall job slowdown variability, as defined in Section 4.2. Figure 4.6 shows the values of the optimal timer for a range of capacities of partition P_1. We see that the optimal timer of partition P_1 is very insensitive to the partition capacity when the job size variability is high. In particular, STATICTAGS has a low optimal timer (below 250 s) of partition P_1 for a relatively long range of partition capacities with both the HVW (squared CV of 20) and the MVW (squared CV of 10) workloads. However, the optimal timer of partition P_1 is considerably more sensitive to the partition capacity when the job size distribution is more balanced. Hence, the optimal timer of partition P_1 increases by a factor of 5 when the partition capacity increases from 10% to 40% for the LVW (squared CV of 4) workload.

Intuitively, TYREX aims to fit in partition P_1 the vast majority of tiny and short jobs, which implies that at most 10% of the total load in all workloads has to be processed in partition P_1 (see Figure 4.5(b)). As a consequence, more than 90% of the total system load, which at an imposed load of 70% amounts to 63% of the system, has to be handled by partition P_2. Therefore, to keep the second partition stable we need to set the size of partition P_1 equal to at most 30% of the total system capacity. Indeed, it turns out that the



(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Figure 4.8: The fractions of jobs completed in partition P_1 and P_2 versus the workload variability with STATICTAGS under a system load of 0.7.



(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Figure 4.9: The median job slowdown and the job slowdown variability versus the workload variability with STATICTAGS under a system load of 0.7.

STATICTAGS policy achieves the best job slowdown performance for all our workloads when the capacity of partition P_1 is set to 20% or 30%. Therefore, in the remainder of this section we report results only for these two partition sizes.

Next, we analyze how the STATICTAGS policy distributes the system load across its partitions. In Figure 4.7 we show the utilizations in partitions P_1 and P_2 for a capacity of partition P_1 of 20% and 30% with its timer set to the optimal value according to Figure 4.6. As we expected, STATICTAGS is rather aggressive in migrating jobs from partition P_1 to partition P_2 so that short jobs which occur in large fractions in our workloads, consistently

receive service in partition P_1 under a relatively low load. In particular, we see that the utilization of partition P_1 is lower than 30% while the utilization of partition P_2 is higher than 75%. Interestingly, we see that the imbalance of the utilizations in partitions P_1 and P_2 is larger for MVW and LVW than for HVW. The reason for this result is that STATICTAGS shifts the job slowdown variability to partition P_2 so that the vast majority of short jobs in our HVW workload are completed in partition P_1.

4.6.2 Performance of StaticTags

The main purpose of setting timers is to have a decent fraction of short jobs finished in the first partition and to avoid overload in any partition. In Figure 4.8 we show the fraction of jobs that are completed in each of the two system partitions when the capacity of partition P_1 is set to 20% and 30%. The timers are set to their optimal values depicted in Figure 4.6. As we expected, the fractions of jobs that are completely executed within partition P_1 is very high for both HVW and MVW (more than 80%). In contrast, the fractions of jobs that are finished in partitions P_1 and P_2 are more balanced for LVW (less than 60% are completed in partition P_1). For all workloads and more noticeable for LVW, the fraction of jobs finished in partition P_1 increases for larger partition sizes.

We show in Figure 4.9 the slowdown performance of the STATICTAGS policy having the capacities of partition P_1 of 20% and 30% with the timers set to their optimal values according to Figure 4.6. As a hint to reading this and later similar figures, the values at 20% capacity of partition P_1 should be interpreted as having a 95^{th} percentile of the job slowdown distribution of about 4.38 (2.05 x 2.14). We see that STATICTAGS has very good performance for the HVW workload, with both the job slowdown variability and the median job slowdown being less than 2. Moreover, we observe that the performance of STATICTAGS is relatively good for the MVW and LVW workloads, with the median job slowdown and the job slowdown variability being less than 2 and 3.3, respectively.

We find that STATICTAGS is not always effective in unbalancing the load across its partitions. Clearly, if STATICTAGS uses a low capacity for partition P_1 then short jobs may experience large job slowdowns because they always run under a relatively high load. We observe this phenomenon when the capacity of partition P_1 is set to 10% for the HVW workload. Similarly, we see that it is difficult to find a good timer when the capacity of partition P_1 is set to 40% because we are at risk at overloading partition P_2.

4.6.3 Evolution of Dynamic Timers

The distinguishing element of the DYNAMICTAGS policy is its operation without the burden of finding the optimal timers for given partition capacities. In this section we will

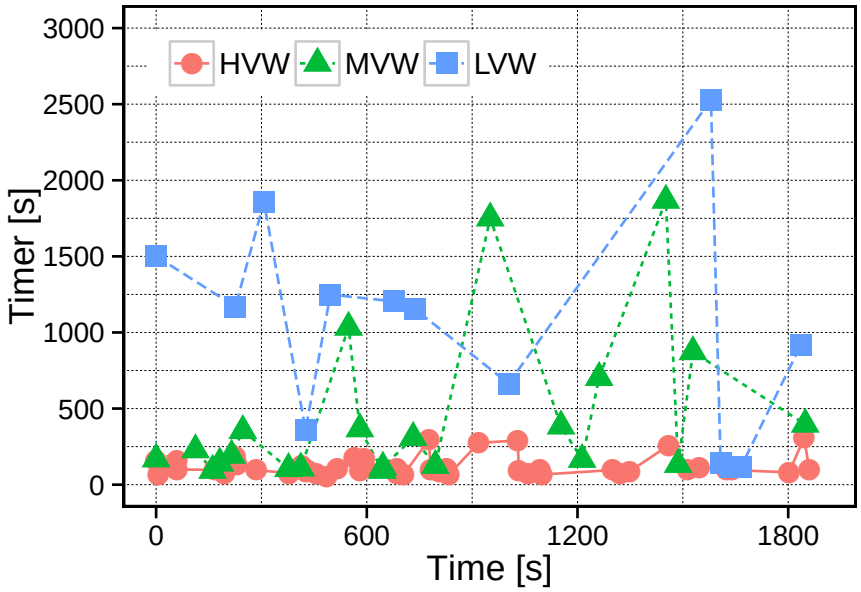
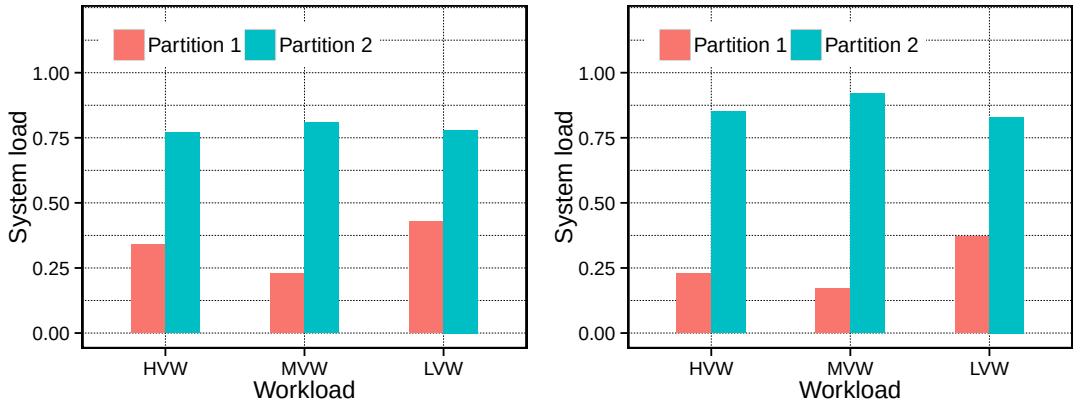


Figure 4.10: The dynamic timer of partition P_1 for a capacity of partition P_1 of 30% under a system load of 0.7.

investigate how the dynamic timers of DYNAMICTAGS adapt over time and their effect on the load across partitions.

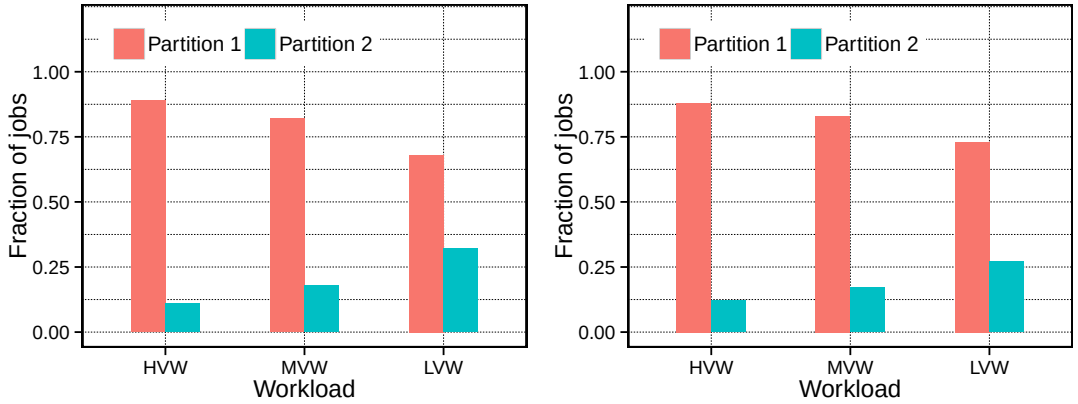
In Figure 4.10 we show for each workload how the value of the dynamic timer of partition P_1 changes during a 30-minute period when the capacity of that partition is set to 30%. Similarly to the STATICTAGS policy, we see that the timer converges to lower values when the job size distribution is more variable. Indeed, for HVW the timer is always set to some value in the range between 50 and 300 s, for MVW most of the timer values are in the range between 100 and 500 s, and for LVW the timer varies between 500 and 2500 s. These increasingly wider and higher ranges nicely match the results in Figure 4.4.

In Figure 4.11 we show how DYNAMICTAGS distributes the imposed system load across its partitions for capacities of partition P_1 of 20% and 30%. We observe that DYNAMICTAGS assigns a significantly lower load to partition P_1 (less than 30% of the total load), which is exactly the same phenomenon as in the case of the STATICTAGS policy (see Figure 4.7). Similarly to the STATICTAGS policy, DYNAMICTAGS aims at migrating long jobs from partition P_1 to partition P_2 so that large fractions of short jobs are consistently served in partition P_1 under a relatively low load. We see that the partition utilizations with DYNAMICTAGS and STATICTAGS are very close for the HVW and MVW workloads. In contrast, DYNAMICTAGS has a higher load in partition P_1 for LVW because its job size variability is only 4. As this workload is more balanced, DYNAMICTAGS has to migrate fewer jobs from P_1 to P_2.



(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Figure 4.11: The utilizations of partitions P_1 and P_2 versus the workload variability with DYNAMICTAGS under a system load of 0.7.



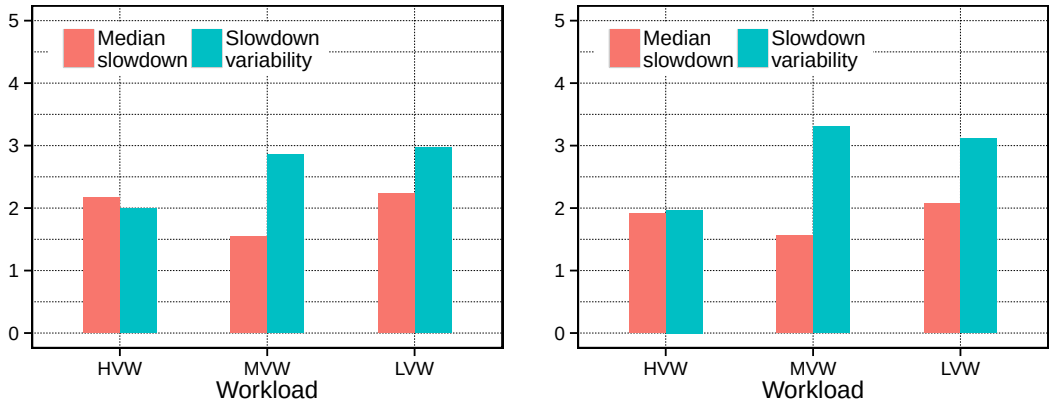
(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Figure 4.12: The fractions of jobs completed in partition P_1 and P_2 versus the workload variability with DYNAMICTAGS under a system load of 0.7.

4.6.4 Performance of DynamicTags

DYNAMICTAGS adapts the timer of partition P_1 as the jobs in the queue make progress, rather than setting a fixed value which is used for all incoming jobs, as in STATICTAGS. Although DYNAMICTAGS operates fundamentally different than STATICTAGS, we show in this section that having timers of any kind (dynamic or fixed) has the same effect on the job slowdown variability.

In Figure 4.12 we show the fractions of jobs that are completed in each partition when the capacity of partition P_1 is set to 20% and 30%. We see that with the DYNAMICTAGS policy more than 80% of jobs in HVW and MVW are completed in partition P_1. Conversely,



(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Figure 4.13: The median job slowdown and the job slowdown variability versus the workload variability with DYNAMICTAGS under a system load of 0.7.

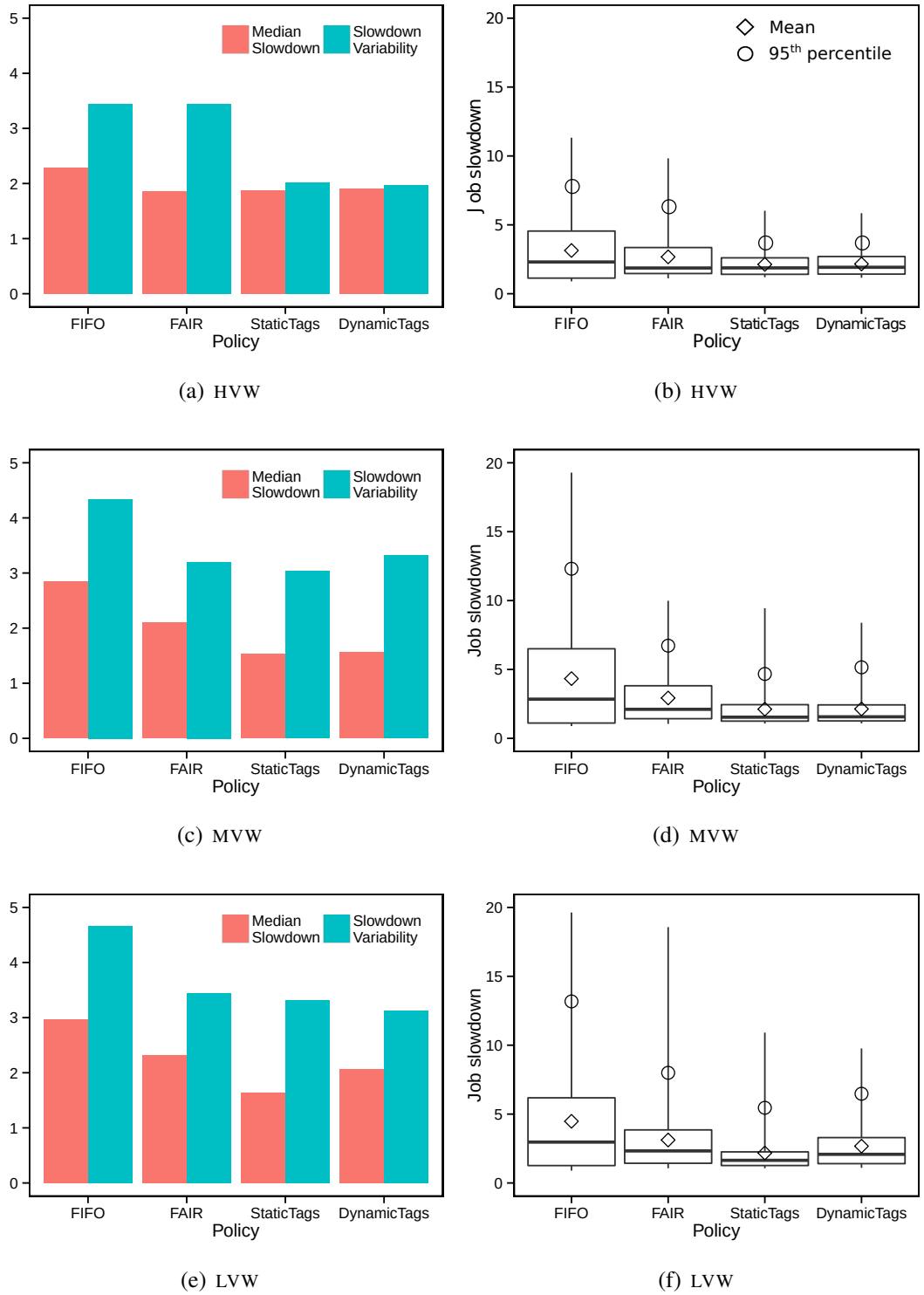
as the workload variability decreases, DYNAMICTAGS migrates more jobs to partition 2, so the fractions of jobs that are completed in different partitions are more balanced with LVW than with HVW and MVW. In particular, we observe that the fractions of jobs that are completed in partition P_1 are 10% higher with DYNAMICTAGS than with STATICTAGS for the LVW workload. This result shows that our DYNAMICTAGS policy is more conservative as it migrates jobs based on their relative progress (current partial job sizes), rather than using a fixed timer as in the STATICTAGS policy.

In Figure 4.13 we show the median job slowdown and the job slowdown variability when the capacity of partition P_1 is set to 20% and 30%. Two important things stand out. First, DYNAMICTAGS offers very low median job slowdown (below 2.2) and slowdown variability (below 3.2) for all our workloads, which are very close to the corresponding values with STATICTAGS shown in Figure 4.9. More importantly, Figure 4.13 also shows that the DYNAMICTAGS policy offers similar improvements for different capacities of partition P_1.

4.6.5 Improvements from Tyrex

In this section we compare our TYREX scheduler with two baselines – FIFO, the default scheduler in Hadoop, and FAIR, the most popular MapReduce scheduler. For TYREX we set the capacity of partition P_1 to 30% and we use the optimal timer according to Figure 4.6 (only for STATICTAGS).

In Figure 4.14 we show the median job slowdown and the job slowdown variability for each policy with all our workloads. For the HVW workload, we find that TYREX with any of its two policies cuts in half the job slowdown variability while maintaining roughly the



(a) HVW (b) HVW

(c) MVW (d) MVW

(e) LVW (f) LVW

Figure 4.14: The median job slowdown, the job slowdown variability, and the job slowdown distributions using FIFO, FAIR, STATICTAGS, and DYNAMICTAGS policies for each workload under a system load of 0.7.

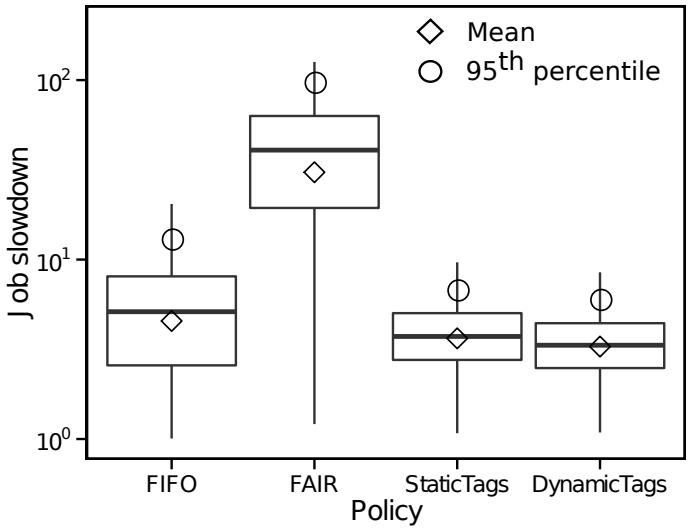


Figure 4.15: The job slowdown distributions using FIFO, FAIR, STATICTAGS, and DYNAMICTAGS policies for the HVW workload under a system load of 0.9 (vertical axis in logscale).

same median job slowdown when compared with both FIFO and FAIR. For the MVW and LVW workloads, when compared with FIFO, TYREX reduces the job slowdown variability and the median job slowdown by 30% and 50%, respectively. Somewhat surprisingly, in these cases the job slowdown variability is roughly equal with TYREX and FAIR, but TYREX with any of its two policies improves the median job slowdown by up to 30%.

In order to better understand the magnitude of the improvements we obtain with TYREX, we compare in Figure 4.14 the job slowdown distributions for each policy with all our workloads. Clearly, TYREX using any of the two policies is superior to both FIFO and FAIR as the distributions of the job slowdowns with TYREX have (much) smaller interquartile ranges and outliers than with the baselines, which was our main goal defined in Section 4.2. In particular, for all workloads TYREX reduces the job slowdown at the 95^{th} percentile by 50-60% when compared with FIFO and by 20-40% when compared with FAIR.

To be realistic, so far we imposed a moderate to high system load of 70%. Finally, we design a stress test to evaluate the performance of TYREX under heavy loads. In Figure 4.15 we show the job slowdown distributions with all policies for the HVW workload under a system load of 90%. TYREX with any of the two policies is by far the best scheduler as the job slowdown distributions are very narrow and much lower than with the baselines. In particular, with TYREX, no job in the workload has a slowdown that is higher than 10. Surprisingly, we find that FAIR has very poor performance and cannot handle the workload when the system is heavy-loaded. Not only is the job slowdown distribution very wide and unbalanced, but also the makespan of the experiment is twice as long as with TYREX. The workload has a total submission schedule of 25 minutes

and is completed by FAIR only 45 minutes after the arrival process ends. In contrast, with TYREX the workload is completed already 9 minutes after the arrival of the last job. Apparently, with FAIR large jobs may occupy underutilized reduce slots which results in very long waiting times for shorter jobs. These non-work-conserving effects in FAIR have a major impact on the stability of the system under high loads. In contrast, TYREX alleviates this issue by confining jobs to smaller partitions, so that no large job monopolizes the reduce slots in the system.

4.7 Related Work

The problem of job size variability was first identified in [144], but since then there has been little work on the performance of size-based scheduling policies in data analytics frameworks. This work aims to advocate for the need of such policies for MapReduce workloads. To this end, we investigate the design and implementation of size-based scheduling policies in MapReduce-based systems.

MapReduce workloads provide an interesting job model with intra-job data precedence constraints between the map and reduce phases which can run on any number of resources as long as the input data is accessible through a distributed filesystem. During the past decade, the performance of MapReduce became a rich exploration domain on diverse scheduling aspects: data locality [144], straggler mitigation [8, 9], resource heterogeneity [147], or elastic scaling [46, 49, 54, 76].

State-of-the art schedulers for MapReduce-based systems assume they have complete control over a fixed set of resources, thus they are typically deployed on dedicated clusters of machines. All three main schedulers incorporated in Hadoop (FIFO [10], FAIR [12], and Capacity [11]) fall into this category. Whereas FIFO executes jobs in order of their arrival with five priority levels using the full system capacity, both FAIR and Capacity divide the system capacity across a number of queues. FAIR uses a processor-sharing scheduling policy to divide the system processors across different (sets of) jobs and Capacity employs multiple statically configured queues in order to confine distinct users to single partitions. The main design motivation of the latter two was to prevent large jobs or heavy users from monopolizing the framework. However, these schedulers do not solve the problem of job slowdown variability. While FAIR hurts the overall cluster performance due to resource contention and thrashing, Capacity does not explicitly handle job sizes.

We have discussed throughout this chapter several *size-based scheduling* policies that have a strong bias towards short jobs. These policies have been analyzed for distributed-server systems [55, 56], supercomputing workloads [102], and cloud compute-intensive workloads [39]. Furthermore, size-based scheduling has been employed in Hadoop with adaptations of two policies: Shortest-Remaining-Processing-Time (SRPT) and Fair-Sojourn-Protocol (FSP) [81, 94]. The main idea behind FSP [41] is to extend SRPT with a job

aging function that virtually decreases the sizes of the waiting jobs, thus avoiding starvation of the large jobs. However, these approaches have rather limited applicability in large-scale datacenters as they require either accurate estimations of job sizes or periodic simulations in a virtually fair system.

Datacenter schedulers such as Mesos [63] and Yarn [127] employ a two-level scheduling architecture by dynamically allocating resources to different specialized frameworks (e.g., Hadoop [10] or Spark [145]). Mesos [63] employs fine-grained resource sharing across different frameworks and delegates the control of resources to the frameworks by initiating *resource offers*. Thus, the frameworks implement specific policies to decide which and how many resources to accept. In contrast, Yarn [127] takes a *request-based* approach and considers application-specific constraints (e.g., job size, hardware requirements, data locality) to allocate resources from a fixed set of machines to each application. As our work focuses on job scheduling in frameworks which have complete control over a fixed set of resources, Yarn can easily incorporate a scheduling system such as TYREX to schedule a more diverse array of applications beyond the traditional MapReduce.

4.8 Conclusion

Reducing job slowdown variability while keeping the median job slowdown relatively low is an attractive yet challenging target in MapReduce frameworks. In this chapter we have introduced a general model for exploring the job slowdown variability problem with heavy-tailed MapReduce workloads. Based on this model we have designed a scheduling system called TYREX that partitions the set of resources of the framework and isolates sets of jobs of very different sizes in those partitions. To do so, TYREX imposes runtime limits (partition timers) and successively executes parts of jobs in a work-conserving way in each partition (the STATICTAGS policy). On top of that, to remove the burden of finding the optimal timers, we develop a statistical model for migrating jobs in a dynamic way that achieves near-optimal job slowdown performance (the DYNAMICTAGS policy).

With a comprehensive set of experiments on a cluster system, we have showed that TYREX achieves very balanced job slowdown distributions for a broad range of representative MapReduce workloads. In particular, TYREX cuts in half the job slowdown variability while preserving the median job slowdown for workloads with high variability of job sizes. Furthermore, unlike FIFO and FAIR, TYREX delivers good performance and balanced job slowdowns even in unfavorable conditions of heavy load. We conclude that TYREX outperforms state-of-the-art schedulers like FIFO and FAIR with respect to both job slowdown variability and median job slowdown for typical MapReduce workloads.

Chapter 5

Reducing Job Slowdown Variability for Data-Intensive Workloads

5.1 Introduction

The ever-growing amounts of data collected and processed by clusters and datacenters cause large disproportions between the sizes of large-scale data-analytics jobs and short interactive queries executed in single systems. As is well known, in the face of a skewed or even heavy-tailed job size distribution, achieving fairness across jobs of different sizes is difficult as small jobs may be stuck behind large ones. For sequential jobs, this problem can be addressed in single-server systems by feedback queuing [99] or processor sharing [27] and in distributed-server systems by having each server process jobs of sizes in a certain range [28, 55]. In some of the policies in the latter case, when job sizes are not known apriori, jobs are restarted from scratch elsewhere, thus wasting processing capacity, when they exceed a local time limit. In contrast, data-analytics jobs using such programming models as MapReduce, Dryad, and Spark have much inherent parallelism, there is no natural way of splitting up resources of a datacenter for specific job size ranges, and jobs may be so large that wasting resources spent on partial executions is not acceptable. In this chapter we propose and simulate four scheduling policies which are rightful descendants of existing size-based disciplines for single-server and distributed-server systems with appropriate adaptations to data-intensive frameworks.

Fairness, in both single-server and distributed-server systems, and more recently in clusters and datacenters, can be considered to be satisfied when jobs experience delays that are proportional to their sizes, which in this chapter is defined as their total processing requirements. Traditionally, the performance of scheduling disciplines with respect to fairness has been measured using job slowdown as a metric. In fact, two dimensions of this metric are relevant—policies designed for highly variable workloads are considered to be fair to the extent that the total distribution of the job slowdown has a low variability,

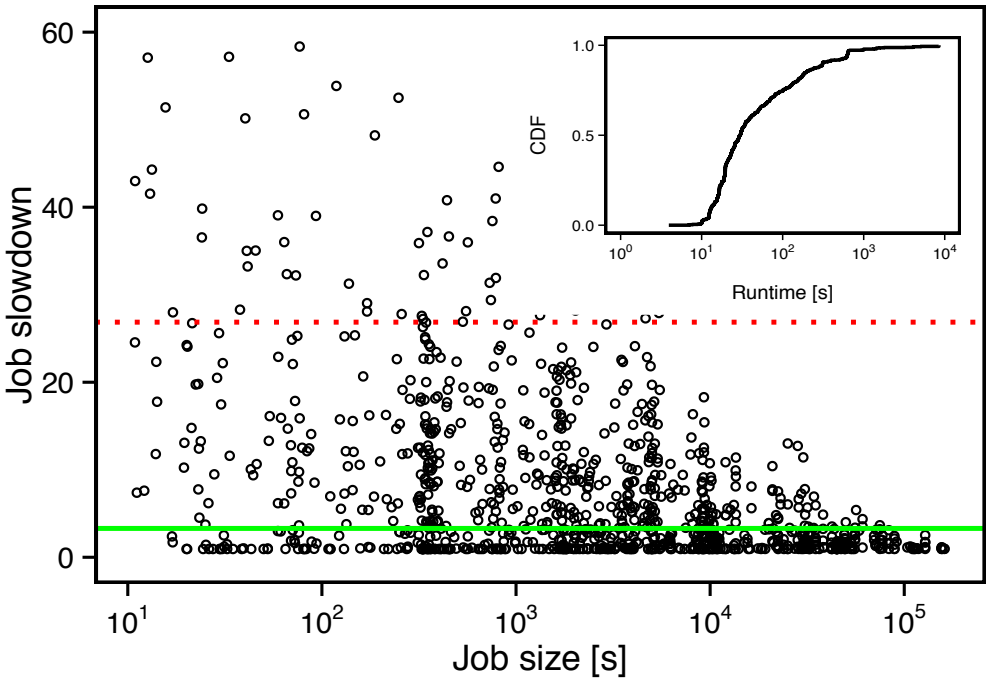


Figure 5.1: A scatter plot of the job slowdown versus the job size for a heavy-tailed MapReduce workload from Facebook (the small figure shows the CDF of the runtimes) with the FIFO scheduler under a system load of 0.7 (the horizontal lines indicate the median and the 95^{th} percentile of the slowdown).

and that it is not biased for certain job size ranges. Therefore, in this chapter the targets are to reduce the *variability of the slowdown* defined as the ratio of the 95^{th} percentile and the median of the job slowdown distribution without significantly increasing the median slowdown, and to even the job slowdowns across the whole range of job sizes.

As an example of the phenomenon we want to tackle, in Figure 5.1 we show the slowdowns versus the sizes for the jobs in a Facebook workload that is scheduled with the standard MapReduce FIFO scheduler. Here, the median and the job slowdown variability (as just defined) are 3 and 9, respectively, and clearly, the small and medium-sized jobs experience the higher and more variable job slowdowns. The inset of the figure shows the CDF of job runtimes, and shows a difference of 3 orders of magnitude between the smallest and the largest jobs. We also find that less than 7% of the jobs in the Facebook workload account for almost half of the total load.

The common denominator of policies for isolating the performance of jobs of different sizes that have been studied in the past is splitting the workload across multiple queues that only serve jobs (or parts of jobs) with processing requirements in certain ranges. Indeed, multi-level FeedBack Queueing (FBQ) [99] is a priority-based single-server time-

sharing policy that relies on preemption of jobs without loss of work already done. In contrast, the Task Assignment by Guessing Sizes (TAGS) policy [55] is designed for distributed server systems, where jobs get killed and are restarted from scratch when being moved to the next queue when they exceed a time limit. Similarly, when job sizes are known (or estimated) apriori, jobs can be immediately dispatched to the appropriate queue upon arrival as in Size-Interval Task Assignment (SITA) [56]. Another way to do the same without knowing job sizes is done by the COMP policy that compares the estimated size of an arriving job to the sizes of some number of the last previously departing jobs [72].

Data-intensive frameworks such as MapReduce have a job model that is very flexible. Jobs consist of many tasks with loose synchronization points between successive stages (e.g., map and reduce), which makes them malleable or elastic. The shared distributed filesystem of MapReduce allows any task to run on any processor in the datacenter. So the opportunity exists to run multiple tasks of a single job in parallel and to run multiple jobs simultaneously, as opposed to the rigid job model supported by FBQ and TAGS in single and distributed servers. Therefore, we have the option to partition the resources of a datacenter across queues, mimicking the operation of distributed-server systems, or to have all queues share the whole non-partitioned datacenter. Moving jobs from one partition/queue to another may be done without killing them by keeping the work previously completed in the distributed filesystem.

With the mechanisms employed by our policies the vast majority of short jobs in MapReduce workloads experience close to ideal job slowdowns even under high system loads (in the range of 0.7-0.9), at the expense of higher slowdowns for a relatively small fraction of large jobs (less than 5%). Further, our policies consistently improve the slowdown variability over FIFO by a factor of 2.

The main contributions of this chapter are as follows:

1. We derive four multi-queue size-based policies for data-intensive workloads with skewed, unknown job sizes that isolate jobs of similar sizes either by migrating them across different queues or partitions without loss of previously completed work, or by judiciously selecting the queue to join (Section 6.4).

2. With a set of real-world experiments, we show that our simulations are remarkably accurate even at high percentiles of the job slowdown distribution (Section 6.5). With a comprehensive set of simulations, we analyze and compare the effectiveness of our scheduling policies in reducing the job slowdown variability of heavy-tailed MapReduce workloads (Section 5.5).

5.2 Job Slowdown Variability

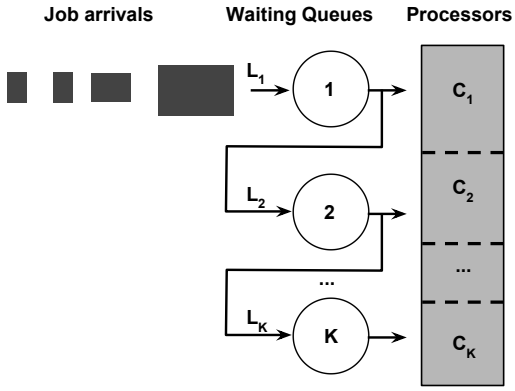
In this chapter we define the *processing requirement* or the *size of a job* to be the sum of the its task runtimes. Clusters and datacenters running frameworks for big-data applications such as MapReduce, Dryad, and Spark are consistently facing workloads with high job size variability, thus raising concerns with respect to large and/or imbalanced delays across the executed jobs [116, 144]. The tension between fast service and fair performance has been an important design consideration in many computer systems such as web servers and supercomputers [41, 102], which are known to execute workloads containing jobs with processing requirements characterized by heavy-tailed distributions.

Whereas users may tolerate long delays for jobs that process large datasets, but most likely expect short delays for small interactive queries, job slowdown, that is, the sojourn time of a job in a system normalized by its runtime in an empty system, is widely used for assessing system performance. The question then is, what statistic of the job slowdown distribution to use. In this chapter, in order to characterize *fair performance* in clusters and datacenters with data-intensive workloads, we use a metric that we call the *job slowdown variability*. Let F be the cumulative distribution function of the job slowdown when executing a certain workload in a system, and let $F^{-1}(q)$ be the q^{th} percentile of this distribution. Then the *job slowdown variability at the q^{th} percentile*, denoted by $V_F(q)$, is defined as the ratio of the q^{th} percentile of F and the median job slowdown, that is:

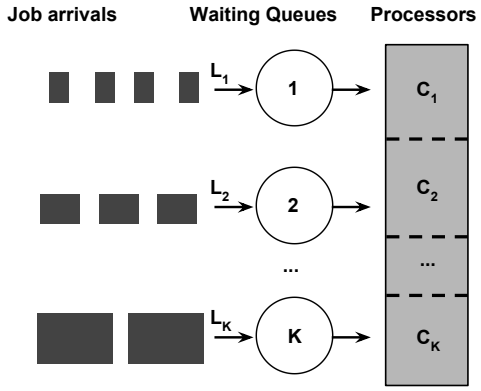
$$V_F(q) = \frac{F^{-1}(q)}{F^{-1}(50)}. \tag{5.1}$$

Intuitively, the slowdown variability at a certain percentile captures some subrange of the slowdowns of all jobs. In the ideal case, $V_F(q) = 1$ for all values of q between 0 and 100, meaning that all jobs have equal slowdowns. Then the policy employed can be called *strictly fair* (for this workload), although that notion has been previously defined when equality of slowdowns holds in expectation [135]. Our target is to minimize the job slowdown variability at different percentiles q, in particular, at $q = 95$, while keeping the median job slowdown low. In this chapter we call $V_F(95)$ the (*overall*) *job slowdown variability* of the workload.

To put fairness in large complex systems a bit in perspective, in an M/G/1 system with load $\rho < 1$, the expected slowdown for any job size under the processor-sharing (PS) discipline is $1/(1-\rho)$ [138]. Further, there is no policy that is both strictly fair and has slowdown strictly less than $1/(1-\rho)$ [135]. Obviously, such strict fairness guarantees lead to performance inefficiency when compared with the Shortest-Remaining-Processing-Time (SRPT) discipline. Not only is SRPT response time-optimal, but the improvement over PS with respect to the mean sojourn time is at least a factor of 2 [16]. Interestingly, de-



(a) FBQ and TAGS *with* feedback – *unknown* job sizes.



(b) SITA and COMP *without* feedback – *known* job sizes.

Figure 5.2: Two general queueing models for reducing job slowdown variability with a single partition or multiple partitions.

spite the general concern of starving large jobs, the degree of unfairness under SRPT is relatively small when the job sizes are drawn from heavy-tailed distributions.

5.3 Scheduling Policies

In order to reduce the job slowdown variability in data-intensive frameworks with jobs that have highly variable processing requirements, in this section we will present four scheduling policies that are inspired by multi-level scheduling with feedback in single-server systems and by size-based scheduling in distributed-server systems.

5.3.1 Mechanisms and Queueing Models

The two mechanisms used by our policies are *logical partitioning* and/or *system feedback*. With the former mechanism, we allocate the compute resources (processors or slots) across disjoint partitions and we restrict for each such partition the amount of service offered to jobs. With the latter mechanism, we use job preemption in a *work-conserving* way by pausing a running job, saving its completed work in the distributed filesystem, and later gracefully resuming its execution from where it left off. In Figure 5.2 we show the two queueing models for that result. The main difference between the two models is whether job sizes are unknown or known, possibly by means of predictions. We propose four policies by combining the two mechanisms in all possible ways.

All our policies have as a parameter some number $K, K > 1$, of waiting queues that serve jobs in FIFO order. Each queue $k = 1, 2, \ldots, K$ has an associated *time limit*

L_k, which is set to the total amount of service that jobs may receive while they are in queue k; L_K is set to ∞. To satisfy our goal, it is crucial that jobs of similar sizes reach the same queue, either through feedback to lower queues (Figure 5.2(a)) or immediately upon arrival (Figure 5.2(b)). In addition, each queue k may be associated with a *partition* of size C_k of the total set of resources (hence the dotted lines in the figures); if so, jobs from queue k are restricted to using resources in the corresponding partition.

5.3.2 The FBQ Policy

Our version of FeedBack Queuing (FBQ) is an extension of multi-level feedback scheduling for the M/G/1 queue [99] to a data-intensive framework running in a datacenter or cluster. It uses the queueing model of Figure 5.2(a) with feedback but without resource partitioning.

An arriving job is appended to queue 1, where it is entitled to an amount L_1 of service. If its processing requirement does not exceed L_1, it will depart the system from queue 1, otherwise it will be appended to queue 2, etc. When processors become available because a task completes, the next tasks to run are selected from the jobs waiting in the highest priority (lowest index number) non-empty queue.

Our FBQ policy for data clusters enables multiplexing so that multiple jobs at potentially different priorities may run simultaneously. Even with Processor Sharing as the queueing discipline in every queue, the latter is impossible with multi-level feedback scheduling in a single-server queue.

5.3.3 The TAGS Policy

Our version of the TAGS (*task assignment by guessing size*) policy is similar to our FBQ policy with the exception that now each queue has its own resource partition where its jobs have to run. As a consequence, whereas with FBQ jobs at different priority levels *may* run simultaneously (when the jobs in the highest non-empty queue cannot fill the complete system), with TAGS, jobs of different sizes *will* indeed run simultaneously in separate resource partitions.

Unlike its predecessor for distributed servers, our TAGS policy for data clusters does not require killing jobs when they are kicked out from one queue to another. Instead, jobs are allowed to gracefully resume their execution without redoing previously completed work. This is an essential design element which eliminates concerns related to inefficiencies of TAGS under higher loads.

5.3.4 The SITA Policy

Similarly to TAGS, SITA does employ per-queue resource partitions, but it does not use feedback. Unlike both FBQ and TAGS, SITA requires a way to predict the sizes of jobs upon their arrival, based on which they are dispatched to the queue of jobs of similar size. A job with predicted size between L_k and L_{k+1} is appended to queue $k+1$. Consequently, the queue time limits have a different role as in the previous policies. Whereas in FBQ and TAGS they are used to keep track of the amount of processing consumed by a job, in the case of SITA they are used as *cutoffs* in the job size distribution for directly dispatching them to the appropriate queue, where they run till completion, even if the prediction is wrong.

Job sizes can be estimated by building job profiles by running (a fraction of) their tasks and collecting samples of the average task duration and the size of the intermediate data they generate. This method has been adopted with good results in previous work for MapReduce jobs when assumptions of uniform task executions within different phases of a job can be made [129]. In practice, we find that a much simpler way of predicting jobs sizes based on their correlation with the job input size is very effective (see Section 5.4.3).

5.3.5 The COMP Policy

All previous policies require setting a number of parameters that is proportional to the number of queues, which may be prohibitive in a real system deployment. We will now present a policy called COMP which is an adaptation to MapReduce of a policy that has been studied before [72]. Similar to FBQ, COMP does not use partitioning of the resources, and similar to SITA, COMP does require job size predictions to send an arriving job immediately to the appropriate queue where it runs to completion. However, in contrast to both SITA and FBQ, COMP does not use queue time limits. Upon the arrival of a job, COMP compares its estimated size to those of the last $K-1$ jobs that have been completed. If the new job is estimated to be larger than exactly m of those jobs for some $m, m = 0, 1, \ldots, K-1$, it is appended to queue $m+1$.

5.3.6 Contrasting the Policies

Despite the common goal of reducing the job slowdown variability, we observe in Table 5.1 the main differences between our policies, which may divide the system capacity across multiple partitions (e.g., TAGS and SITA), may use preemption and relegation to another queue (e.g., FBQ and TAGS), or may use some form of prediction to anticipate job sizes (e.g., SITA and COMP). Although our policies resemble the structure of former scheduling disciplines for single-server and distributed-server systems, there are three key elements in which their correspondents for datacenters are different.

Table 5.1: A policy framework for scheduling data-intensive jobs in datacenters.

Policy	Queues	Partitions	Feedback	Job size	Param.
FIFO	single	no	no	unknown	0
FBQ	multiple	no	yes	unknown	K
TAGS	multiple	yes	yes	unknown	$2K-1$
SITA	multiple	yes	no	predicted	$2K-1$
COMP	multiple	no	no	compared	1

First, the original policies were designed for a single or distributed-server model in which each host is a single multi-processor machine that can only serve one job at a time. In contrast, we target a datacenter environment in which the system capacity may be divided across partitions with many resources. As a result, instead of only having the time cutoffs as parameters, in our model we also have the partition capacities as parameters.

Secondly, original TAGS and SITA assume simple, rigid (sequential or parallel) non-preemptible jobs that may only run on a single host until completion. In contrast, our (MapReduce) job model is more complex as there are intra-job data precedence constraints (map before reduce) and data locality preferences (of map tasks), and as jobs are elastic (or malleable) and can run simultaneously, taking any resources they can get when it is their turn.

Thirdly, original TAGS does not preserve the state of a job when it moves it from one server to the next. Hence, long jobs will get killed at every server except at the one where they run to completion, at every step losing all the work performed and thus wasting CPU time. Instead, we take a work-conserving approach by allowing jobs that are being moved from one partition to the next to retain their work and to gracefully resume their executions without redoing previously completed work. This way of operation is facilitated by the filesystem that is shared across the whole framework, with the intermediate results of tasks executed within any partition being persistent and visible after a job has been moved to another partition.

5.4 Experimental Setup

We evaluate and compare our policies with the job slowdown variability at different percentiles as the main metric for representative MapReduce workloads. Given the large space of policy configurations (e.g., the number of queues, the partition sizes, and the queue time limits), in this chapter we take an experimental approach through realistic simulations of MapReduce to completely understand the impact of our policies on the slowdown variability in a MapReduce framework when jobs have very different sizes.

Table 5.2: The characteristics of the jobs used in the validation, and the job runtime in the simulations (SIM) and in the real deployment (DAS).

Applications	Maps	Reduces	Job Size [s]	SIM [s]	DAS [s]	Jobs
GREP	2	1	63.14	36.10	43.26	26
SORT	4	1	60.20	32.70	39.97	4
WCOUNT	4	1	126.14	42.04	49.73	4
GREP	50	5	155.32	42.83	53.18	4
WCOUNT	100	10	3,790.46	86.80	93.62	3
SORT	200	20	5,194.64	149.92	156.89	3
GREP	400	40	15,697.18	233.63	239.21	3
WCOUNT	600	60	26,662.53	579.73	589.02	3

5.4.1 Simulator

We have modified Apache's open-source MapReduce simulator Mumak [87] to include our scheduling policies. Although many discrete-event simulators for traditional queueing models exist, we chose Mumak [87] for its compatibility with the popular open-source MapReduce implementation Hadoop. Thus, Mumak reproduces closely the internals of Hadoop by simulating with a discrete time scale the MapReduce job scheduling and task allocation. Furthermore, Mumak can employ without changes the standard Hadoop schedulers (e.g., FIFO, Capacity [11]).

A subtle point in simulating MapReduce is to appropriately adjust the reduce task runtimes based on the shuffle phase duration. Mumak schedules reduce tasks only when a predefined fraction of map tasks have finished (the default value is 5%). Although Mumak allows reduce tasks to occupy slots during the map phase, their runtime duration is simulated from the moment when all map tasks are finished. As reduce tasks may still be in their shuffle phase even after all map tasks have finished, Mumak conspicuously underestimates the job completion time. To reproduce closely the shuffle phase of the MapReduce job execution, we changed the simulator and incorporated the remaining duration of the shuffle phase in the reduce task runtimes.

Mumak is not completely event-based but has time-based elements, as the heartbeats through which tasks are assigned to idle slots in the framework are simulated periodically at fixed intervals. Although useful in practice to implement the interaction between the components of the MapReduce framework (e.g., JobTracker and TaskTrackers), this artifact pollutes the simulation results by leaving slots idle between two consecutive heartbeats, thus reducing the utilization of the framework. Therefore, we changed the simulator and removed the periodic heartbeat by forcing slots to initiate a heartbeat whenever they become idle.

In all our simulations (except the validation experiments presented in Section 5.4.2), we use a cluster consisting of 100 nodes on which one MapReduce framework is in-

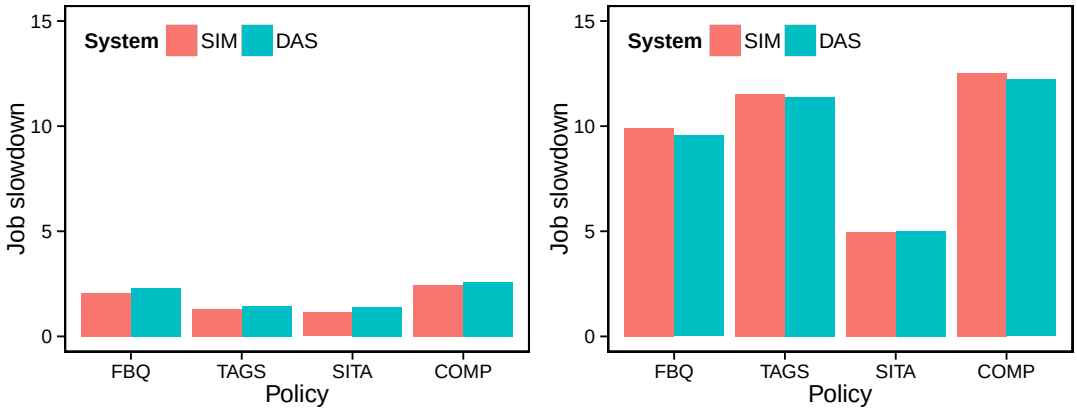
stalled, and we configure each node with 6 map slots and 2 reduce slots. This size of our MapReduce framework is comparable to production deployments of MapReduce frameworks [8, 144]. For each simulation, we report the averages of our performance metrics over three repetitions.

5.4.2 Validation

With a set of both simulations and real-world experiments, we assess the accuracy and the robustness of our modified simulator. We run real-world experiments on the Dutch six-cluster wide-area computer system DAS (fourth generation) [118]. The TU Delft cluster, which we use for this validation, has 24 dual-quad-core compute nodes, with 24 GiB memory per node and 50 TB total storage, connected within the cluster through 1 Gbit/s Ethernet (GbE) and 20 Gbit/s QDR InfiniBand (IB) networks. In our real-world experiments, we use our prototype of Hadoop (version 1.0.0), which includes the implementations of the four policies. For both our simulations and real-world deployment, we configure 10 nodes with 6 map slots and 2 reduce slots.

To evaluate how accurately our simulator approximates real-world executions of single MapReduce jobs, we compare the simulated runtimes with the job runtimes in the real system for 8 different job types. As we show in Table 5.2, these jobs are instances of three well-known MapReduce benchmarks (Grep, Sort, and Wordcount) with very different numbers of tasks (between 3 and 660) and variable total processing requirements (between 1 minute and 7.4 hours). Our simulator estimates in most cases the job run times with less than 10% error (which is comparable to the error in SimMR [130] developed at HP Labs). In fact, the difference between the job runtimes in our simulator and in the real-world system is always at most 7-10 s (columns 5 and 6 in Table 5.2). This difference represents the overhead of setting up and cleaning up a MapReduce job in Hadoop, which we do not account for in simulation.

To assess the robustness of our simulator with complete MapReduce workloads, we compare the slowdown performance of our policies in simulation and real-world deployment. The workload we use in this evaluation has 50 jobs in total and consists of different fractions of job types, as indicated in Table 5.2. We simulate and execute the workload in our cluster using each of the four policies with $K = 2$ queues. We set the size of partition 1 to 30% for both TAGS and SITA. The time limits we find to be the best for TAGS, SITA, and FBQ are 120, 160, and 160 seconds. Figure 5.3 shows the slowdown performance of our policies in both the simulator and the real-world deployment when job arrivals are Poisson and the system load imposed is 0.7. Our simulator is remarkably accurate and delivers slowdown performance that is consistent with the real-world experiments. The



(a) Median

(b) 95^{th} percentile

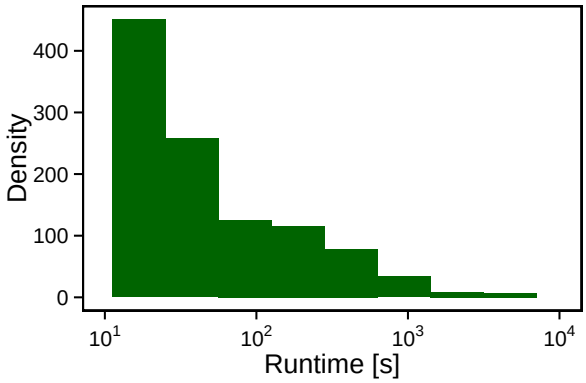
Figure 5.3: The slowdown performance of our policies for the workload summarized in Table 5.2 using both simulations (SIM) and real-world experiments (DAS).

relative error between the simulations and the real-world experiments is less than 1% for both the median job slowdown and the job slowdown at the 95^{th} percentile.

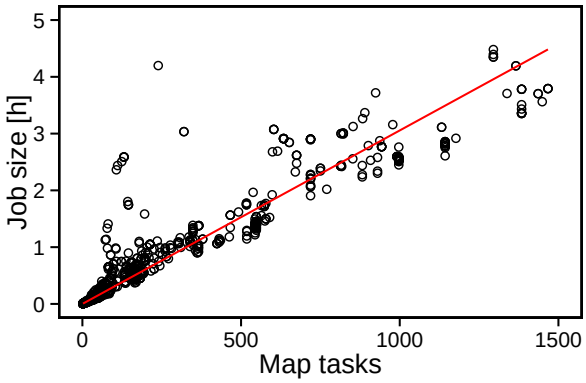
5.4.3 Workloads

The workloads we impose on the system in our simulations are based on a MapReduce trace from Facebook that spans 6 months, from May 2009 to October 2009, and contains roughly 1 million jobs. This trace is public and contains the size in bytes for every job for each MapReduce job phase. We employ the SWIM benchmark [24] which uses the records in the trace to generate synthetic jobs. We create and execute a set of 1,121 such synthetic jobs on a real 20-node Hadoop cluster and record for each job all relevant information (e.g., task runtimes) so that it can be executed in our simulator. We generate different workloads of 1,121 jobs by randomly selecting jobs without repetition from this set of synthetic jobs. The job inter-arrival time has an exponential distribution.

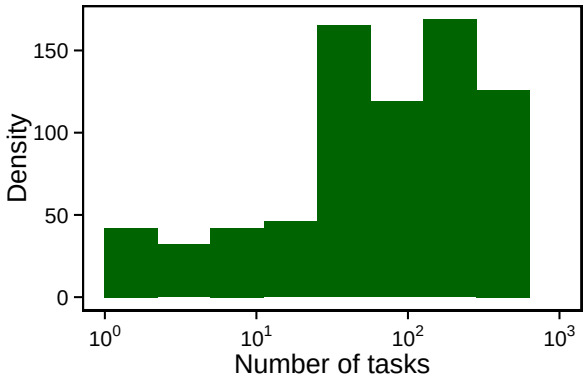
In order to compute the slowdowns experienced by the jobs in our simulations, we determine the reference runtime of each job in an empty 100-node simulated MapReduce framework from when its first task starts until its last task terminates. The sum of the reference runtimes of all jobs accounts for approximately 60 h of simulated time. In Figure 5.4(a), we show the distribution of these reference job runtimes of the jobs in our workloads, which is very variable, as its squared coefficient of variation is equal to 16.35. As reported for other data-intensive workloads at Facebook, Google, Cloudera, Yahoo! [23], the distribution of the job sizes is skewed, with fewer than 8% of the jobs in our workloads responsible for almost half of the total load. In contrast with the reference runtimes, the job sizes (total processing requirement) are less variable, with the squared



(a) The density of the reference job runtimes (horizontal axis in log scale).



(b) The job size versus the job input size.



(c) The density of the number of tasks per job (horizontal axis in log scale).

Figure 5.4: Properties of the workload generated with SWIM measured in a simulated empty 100-node system: a histogram of the reference job runtimes (a), the correlation between the input size and the processing requirement of jobs (b), and a histogram of the number of tasks per job (c).

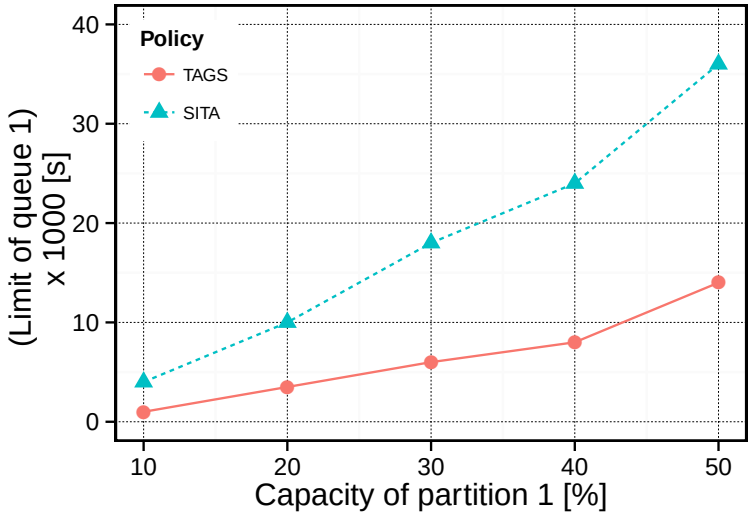


Figure 5.5: The optimal time limit of partition 1 for each capacity between 10-50% at a system load of 0.7.

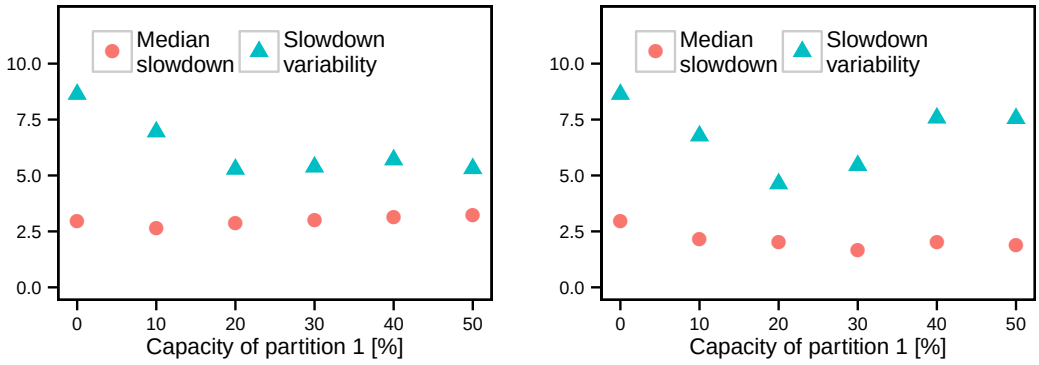
coefficient of variation equal to 3.32. Further, we find in our workloads a strong correlation between the total job input size and the total processing requirement (the size) of the job (see Figure 5.4(b)), which is used by SITA and COMP to assign arriving jobs to queues. Finally, jobs in our workload may achieve very different levels of parallelism: from less than ten tasks to more than 10,000 tasks (Figure 5.4(c)).

5.5 Experimental Evaluation

In this section we first investigate the impact of the key parameters on the slowdown variability (Section 5.5.1). Further, we analyze the effect of load unbalancing across partitions on the performance of TAGS and SITA (Section 5.5.2), the performance under heavy traffic (Section 5.5.3), and the degree of unfairness in our scheduling policies (Section 5.5.4). Finally, we analyze the performance of FBQ and COMP with more than two queues (Section 5.5.5).

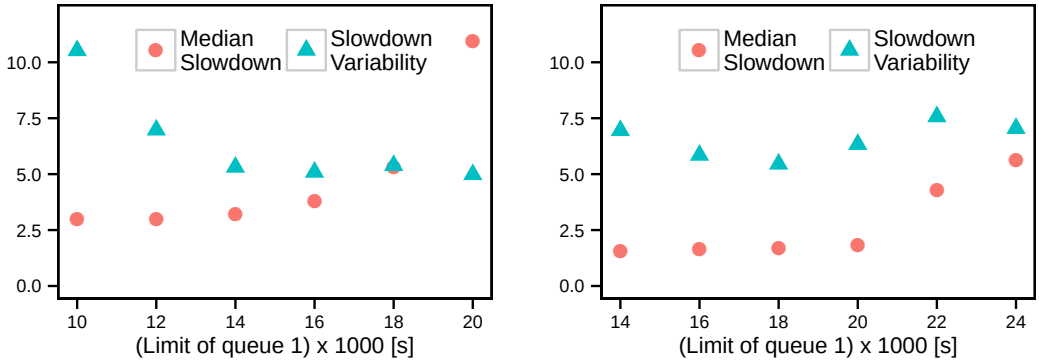
5.5.1 Parameter Sensitivity

All our policies assume a single MapReduce framework with some number K of queues. Although having multiple partitions may reduce the job slowdown variability by starting short jobs faster rather than having them wait behind relatively large jobs, in practice, configuring many partitions and setting the corresponding time limits may be difficult—when set incorrectly, system fragmentation and poor utilization of partitions may result. Fortunately, as we show in our analysis, having only two queues (and partitions) already



(a) TAGS, $\rho = 0.7$ (b) SITA, $\rho = 0.7$

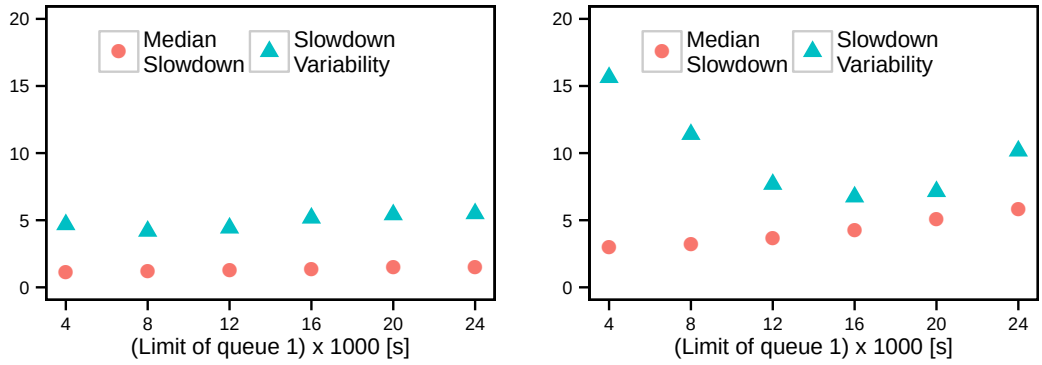
Figure 5.6: The median job slowdown and the job slowdown variability versus the capacity of partition 1 under a system load of 0.7 (capacity 0% corresponds to FIFO).



(a) TAGS, $\rho = 0.7$ (b) SITA, $\rho = 0.7$

Figure 5.7: The median job slowdown and the job slowdown variability versus the time limit in queue 1 at a system load of 0.7 (the horizontal axes have different scales). The size of partition 1 is set to 50% (TAGS) and 30% (SITA).

significantly reduces the job slowdown variability and is sufficient to reach our goal. Thus, in our simulations we set for all policies $K = 2$. In this case, for FBQ, only one parameter has to be set, which is the time limit of queue 1. In contrast with the previous implementations in single-server and distributed-server systems, our TAGS and SITA policies require an additional parameter to be set for each queue, which is the partition capacity—in our case only the capacity of partition 1. We don't have to consider COMP here as it operates without partitioning and without time limits. We seek the optimal values of the parameters (capacity and/or queue time limit of partition 1) for which our policies achieve the lowest job slowdown variability.



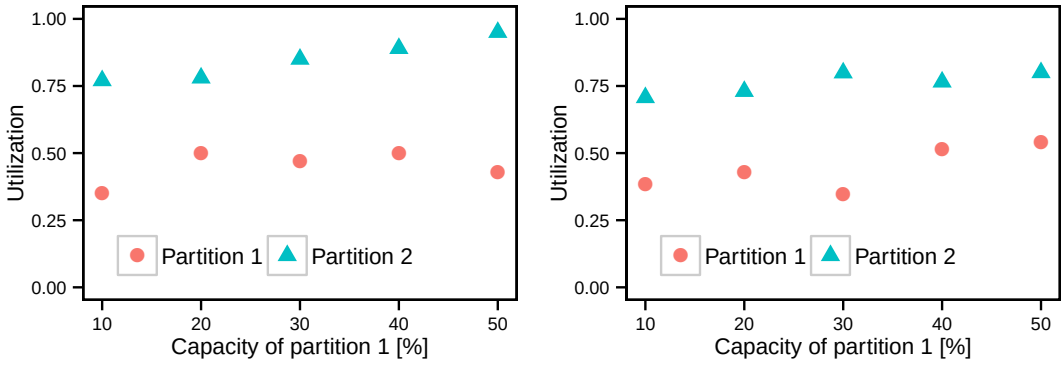
(a) FBQ, $\rho = 0.7$ (b) FBQ, $\rho = 0.9$

Figure 5.8: The median job slowdown and the job slowdown variability versus the time limit in queue 1 at a system load of 0.7 and 0.9.

We will first investigate the relation between the partition size and the queue time limit for TAGS and SITA. To this end, we show in Figure 5.5 the optimal time limit of partition 1 for a range of sizes of partition 1. Obviously, SITA has a higher time limit of partition 1 than TAGS, as jobs with SITA run to completion. Therefore, we expect TAGS to operate well even at high capacities of partition 1, but we want SITA to utilize a smaller partition 1.

In Figure 5.6 we show how TAGS and SITA actually perform in terms of the job slowdown when the capacity allocated to the first partition varies between 0-50% and the queue time limit for each size of partition 1 is set to the optimal value as indicated by Figure 5.5. As a hint to reading this and later similar figures, the values at 0% capacity of partition 1 should be interpreted as having a 95^{th} percentile of the job slowdown distribution of about 23.8 (2.8 x 8.5). We observe that the impact of the partition size is relatively small with TAGS over a wide range of sizes—a capacity of partition 1 ranging from 20% to 50% is fine. In contrast, with SITA, setting the partition sizes is much more critical; the job slowdown is significantly better when the capacity of partition 1 is 20% or 30%, depending on whether the median or the job slowdown variability is considered more important. Outside that range, the job slowdown variability is much higher.

Finally, for the TAGS and SITA policies, we investigate the setting of the time limit of queue 1 having the capacity of partition 1 set to 50% and 30%, respectively. Figure 5.7 depicts the slowdown statistics for large ranges of queue time limits. TAGS has relatively low median and high variability at low queue time limits, and the other way around at high limits. SITA is relatively stable in the range of 14-20 x 1000 seconds, but for higher values of the time limit, the median value gets very poor. The time limits we consider to be the best for TAGS and SITA are 14,000 and 18,000 seconds, respectively (as already indicated



(a) TAGS, $\rho = 0.7$ (b) SITA, $\rho = 0.7$

Figure 5.9: The utilizations of partition 1 and 2 versus the capacity of partition 1 at a system load of 0.7.

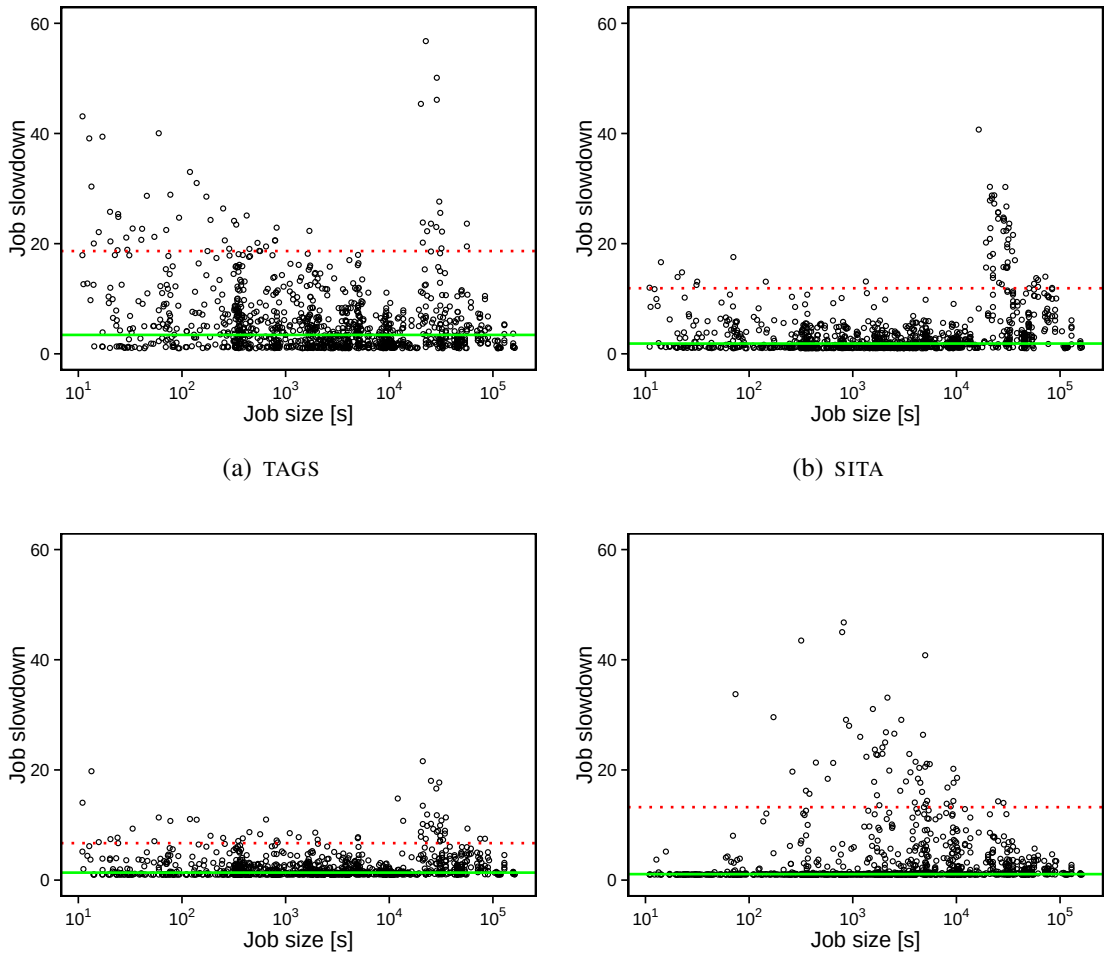
in Figure 5.5 for the partition sizes considered here). Interestingly, the fraction of work that gets completed in partition 1 is 35% for both policies with these optimal cutoffs.

Further, we investigate the setting of the time limit of queue 1 for the FBQ policy. The slowdown statistics in Figure 5.8 show two important things. First, we see from Figure 5.8(a) that under a system load of 0.7, FBQ is very insensitive to the queue time limit, in contrast to both TAGS and SITA. However, in Figure 5.8(b) we show that going to 0.9 system load, FBQ becomes more sensitive. We analyze in more detail the performance of all policies under heavy traffic in Section 5.5.3. Secondly, Figure 5.8 shows that FBQ is by far the best performing policy with respect to both the median slowdown and the slowdown variability across the whole range of queue time limits considered. The time limit we consider to be the best for FBQ is 12,000 seconds.

5.5.2 Load Unbalancing

In previous studies of the TAGS and SITA policies for distributed server systems, it has been shown that choosing the queue time limits (or cutoffs) to balance the expected load across partitions can lead to suboptimal median slowdown [28, 55]. Counterintuitively, load unbalancing optimizes fairness when the workload has a heavy-tailed distribution. The intuition behind this strategy is that a large majority of the jobs get to run at a reduced load, thus reducing the median slowdown.

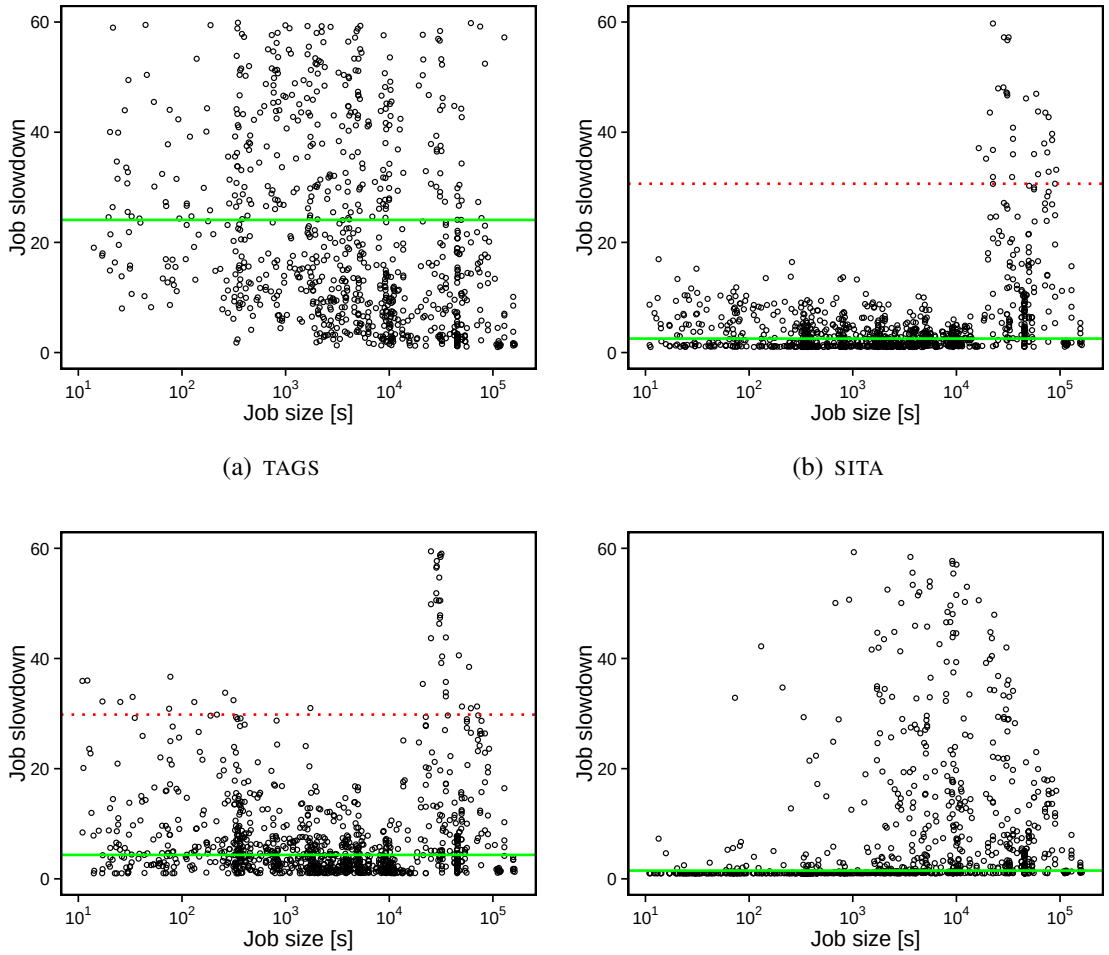
We investigate now whether load unbalancing has the same effect in our MapReduce use case. Figure 5.9 shows the utilizations of partitions 1 and 2 for different capacities of partition 1 with the corresponding optimal queue time limits as we have determined in



(a) TAGS (b) SITA

(c) FBQ (d) COMP

Figure 5.10: Scatter plots of the job slowdown versus the job size for all policies under system loads of 0.7. The horizontal lines show the median and the 95^{th} percentile of the job slowdown distribution.



(a) TAGS (b) SITA

(c) FBQ (d) COMP

Figure 5.11: Scatter plots of the job slowdown versus the job size for all policies under system loads of 0.9. The horizontal lines show the median and the 95^{th} percentile of the job slowdown distribution. The horizontal lines that show the 95^{th} percentile of TAGS and COMP are higher than 60.

Section 5.5.1. Indeed, we observe for TAGS and SITA that for any capacity of partition 1 in the range 10-50%, partition 1 is assigned significantly less load than partition 2.

Nevertheless, it seems that when we try to achieve both low median job slowdown and low job slowdown variability, only SITA is comparable to FBQ, while TAGS seems to be hitting a wall. The former two policies have very close slowdown variability when they operate at their optimal time limits – 18,000 seconds for SITA (see Figure 5.7(b)) and 12,000 seconds for FBQ (see Figure 5.8(a)). In contrast, in Figure 5.9 we show that TAGS has a very high utilization in partition 2 at the optimal partition size of 50% and the optimal time limit of 14,000 seconds. This can be explained by what is the major difference between TAGS and SITA. Whereas SITA runs each job till completion in its "own" partition (wrong job size predictions can be made), TAGS moves jobs across multiple partitions through preemption until they reach the proper queues where they can complete. So SITA provides better isolation for short jobs than TAGS does. As a consequence, with TAGS the slowdowns of short jobs may be significantly higher than with SITA, as very large jobs with high levels of parallelism may monopolize the entire partition 1, no matter what its capacity is set to. Thus, TAGS with its optimal time limit of 14,000 seconds may achieve a lower job slowdown variability than SITA with its optimal time limit of 18,000 seconds, but it does so at the expense of a significantly higher median job slowdown.

5.5.3 Heavy-Traffic Performance

One major criticism of the TAGS and SITA policies in distributed server systems has been the strong dependence of their performance on the system load [55, 102]. As the load increases, unbalancing the load across the two servers (partitions) is difficult to achieve without overloading the second server (partition). In Figures 5.10 and 5.11 we show scatter plots of the job slowdown versus the job size for all policies with the optimal parameters. We observe that under system load of 0.9, the performance of TAGS is very poor, with the median job slowdown being twice as large as with FIFO. Further, with both TAGS and COMP the job slowdown at the 95^{th} percentile is higher than 60. Despite having a relatively lower median job slowdown than TAGS and twice as low as FIFO, SITA hurts significantly the top 5% large jobs in our workload.

Although one would expect this issue to be removed with FBQ, it turns out this is not necessarily true for MapReduce. We observe that even for FBQ, balancing between the two optimization goals, i.e., the median job slowdown and the job slowdown variability, is difficult, especially when the system is under heavy-traffic. To explain this, we have to look at the internal structure of MapReduce. As we have explained in our model, when jobs are being preempted, they are allowed to finish their running tasks no matter their given priority at that time. This has a negative effect on both the system utilization and

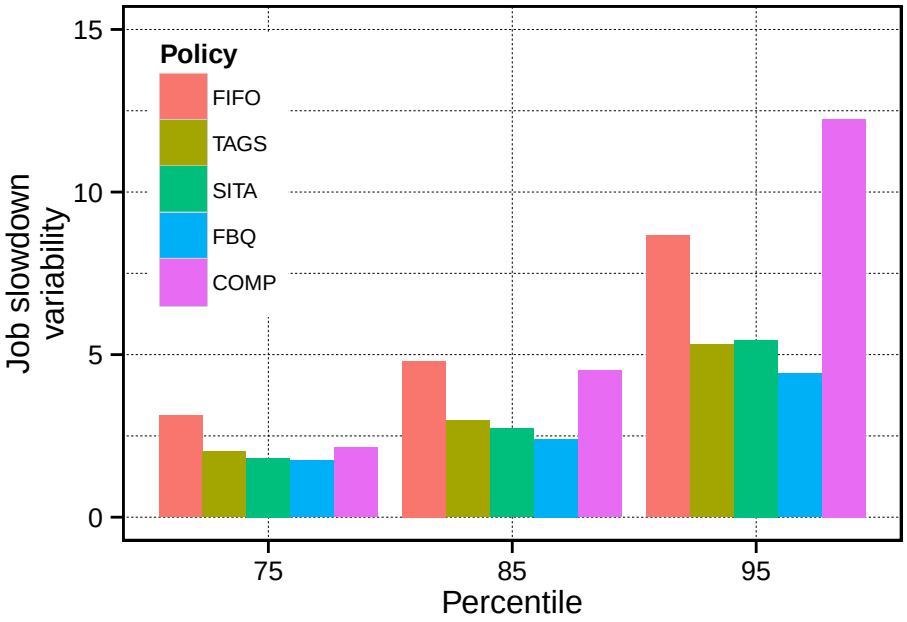


Figure 5.12: The job slowdown variability at different percentiles.

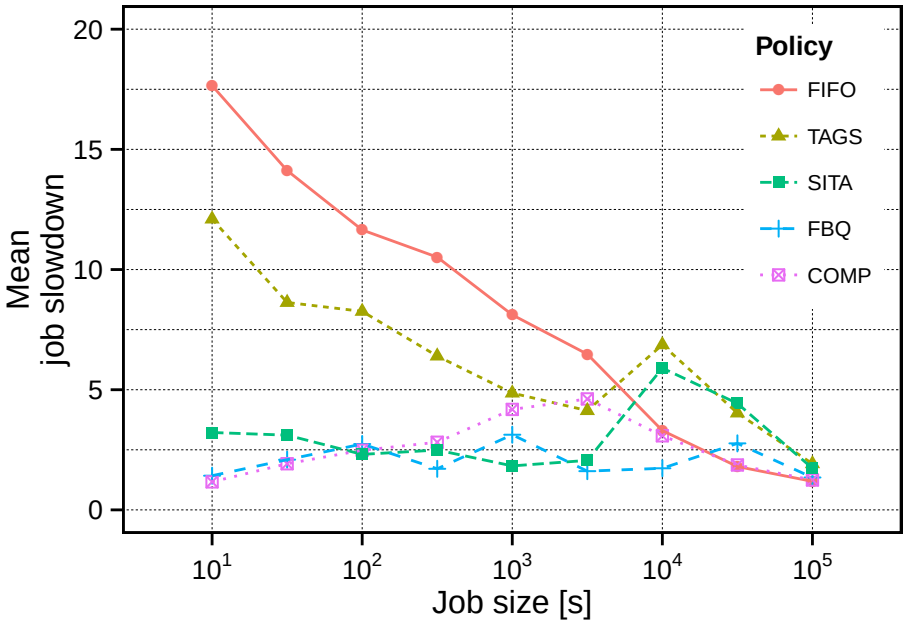


Figure 5.13: The mean job slowdown per job-size subrange of the complete range of job sizes at a system load of 0.7 (horizontal axis in log scale).

the slowdowns of large jobs, which may have reduce tasks unnecessarily occupying slots, while their map tasks are waiting for higher-priority queues to become empty.

5.5.4 Fairness Analysis

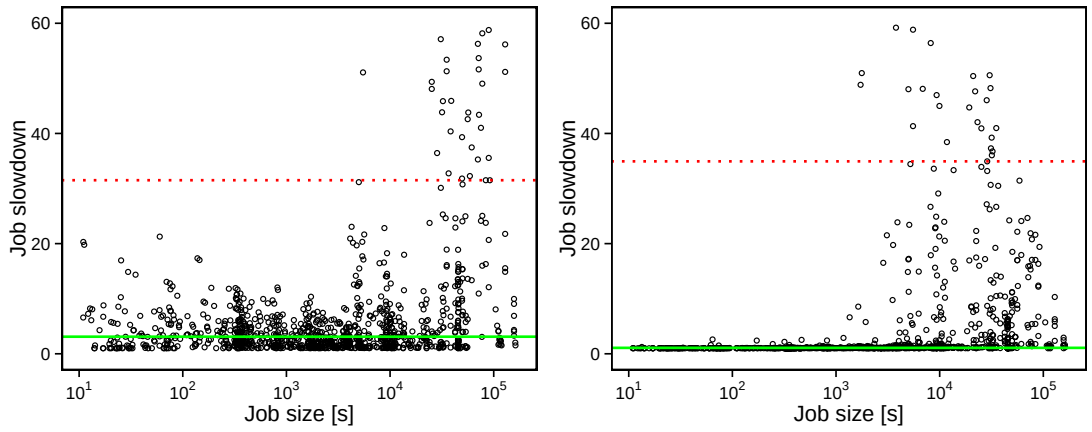
Recent mathematical analysis has shown that scheduling policies that are biased against long jobs by giving priority to relatively small jobs, are not only optimal with respect to mean response time or mean slowdown, but are also fair [16, 58, 59]. In addition, as MapReduce was originally created for jobs processing terabytes of data, scheduling disciplines that allow short jobs to preempt large ones have never been used in practice for MapReduce workloads for fear of hurting the very large jobs. In this section we will present summary evidence to what extent our policies help in the two dimensions of job slowdown variability: reduced job slowdown variability (the ratio of the 95^{th} and 50^{th} percentiles of the job slowdown distribution) and even slowdowns across the complete range of job sizes.

In Figure 5.12 we show the job slowdown variability at different percentiles (as defined in Section 5.2) under a system load of 0.7. COMP provides gains over FIFO only up to the 95^{th} percentile. It is obvious that here FBQ is superior to TAGS, SITA, and that all these policies achieve a significant improvement over FIFO. FBQ consistently improves the slowdown variability at all percentiles by a factor of 2 over FIFO. Although TAGS and SITA are very close to FBQ, they seem to shift the slowdown variability to the second partition, where the largest jobs with thousands of tasks experience slowdowns because of being confined to smaller partitions. In fact, it is not the truly large jobs that suffer, but those jobs that are not sufficiently small to be completed in partition 1 and therefore end up in partition 2.

We next compare how the mean job slowdown varies across different job sizes with each policy. In Figure 5.13, we show the mean slowdowns of jobs in logarithmically spaced subranges of the complete range of job sizes (so, the first job-size range covers sizes between 10 and $10\sqrt{10}$, the second between $10\sqrt{10}$ and 100, etc.) at a utilization of 0.7. The mean slowdowns with FBQ, SITA, and COMP are considerably lower and less variable than with TAGS and FIFO, with FBQ being somewhat more stable. Thus, in particular FBQ manages to achieve a very even job slowdown across the complete job size range, which is exactly what we wanted.

5.5.5 More than Two Queues

When configured with $K = 2$ queues, both FBQ and SITA consistently improve the job slowdown at the 95^{th} percentile by a factor of 2 over FIFO for system loads between 0.7 and 0.9. However, under system loads of 0.9 or higher, a relatively small fraction of large



(a) FBQ (b) COMP

Figure 5.14: Scatter plots of the job slowdown versus the job size for FBQ and COMP with $K = 4$ queues under a system load of 0.9. The horizontal lines show the median and the 95^{th} percentile of the job slowdown distribution.

jobs (less than 5%) experience considerably larger slowdowns than the median. Therefore, we want to assess whether having more queues may reduce even more the job slowdown variability. As for SITA, an additional queue comes with the burden of partitioning and assigning appropriate capacities to those partitions, we compare only FBQ and COMP.

We will first explain the way we set the time limits of the queues for FBQ. Recall from Figure 5.11(c) that the job slowdowns offered by FBQ in optimal setting for $K = 2$ have relatively different values across three ranges of job sizes. The job slowdowns are somewhat stable for job sizes between 4,000 seconds and 10,000 seconds, but are very variable for job sizes outside that range. Thus, we set the time limits of queue 1 and queue 2 to 4,000 seconds and 10,000 seconds, respectively. As for queue 3, we set the time limit to 36,000 seconds, which is the size of the job that suffers the highest job slowdown in the optimal setting for $K = 2$. Even though there may be other better values, setting the time limits is very critical for FBQ with multiple queues at system loads of 0.9 or higher.

In Figure 5.14 we show scatter plots of the job slowdown versus the job size for FBQ and COMP when the number of queues K is set to 4. Two points stand out. First, FBQ with $K = 4$ offers considerable gains when the system is under heavy load. The median slowdown improves by 30% and no job of relatively small size (below 4,000 seconds) is over the 95^{th} percentile of the job slowdown distribution. Secondly, in contrast with FBQ, COMP has lower median job slowdown and more variable job slowdowns for large job sizes.

5.6 Related Work

A large body of both theoretical and experimental work exists on the evaluation of task assignment policies that are biasing towards jobs with small sizes. However, despite the superior performance of size-based policies datacenter schedulers have not yet incorporated them for fear of unfairly hurting very large jobs.

It is well-known that the Shortest-Remaining-Processing-Time (SRPT) discipline is the best policy with respect to mean sojourn time given any arrival sequence and job size distributions [100, 110]. Another policy with comparable performance to SRPT is the Shortest-Processing-Time-Product (SPTP), which serves jobs according to the minimum product of initial size and remaining service time. In an M/G/1 setting, it has been shown that SPTP is optimal with respect to mean slowdown [65].

In contrast to the general premise that SRPT may treat unfairly large jobs which may starve given an adversarial arrival sequence, recent work has shown that such concerns are not realistic in the average case. More specifically, it can be proved that for an M/G/1 model and workloads which are heavy-tailed, all jobs experience a lower response time under SRPT when compared to the more popular processor-sharing (PS) discipline [16, 59]. SRPT achieves similar performance even for general job size distributions under moderate loads, and is not prohibitive even under higher loads. As a follow-up to the previous result, it has been shown that in a more practical setting of a web server, SRPT-based scheduling is to be preferred over the de-facto FAIR scheduler which fairly allocates fractions of resources to handle incoming requests [58]. However, SRPT is rarely used in current schedulers because of the practical consideration of possibly unknown job sizes. Luckily, the former drawback of SRPT discipline can be alleviated in several ways. First, many computer workloads exhibit strong correlations between the job input size and the processing time (or size) of the job. This phenomenon has been noticed in the case of web server requests, and is valid for MapReduce workloads as well. Secondly, when job sizes cannot be anticipated in any way, SRPT may be approximated in an M/G/1 model by employing multi-level scheduling with a large (infinite) number of feedback queues [99].

One representative policy for the problem of task assignment for server farms is Size-Interval-Task-Interval-Assignment or SITA [56], which isolates all jobs with sizes within a predefined size interval to a single server. SITA was specifically designed to reduce variability at each queue by dividing the job size distribution so that each queue handles a less variable portion of the original distribution. Although it may seem that the optimal cutoff points in the distributions should be chosen in such a way that the load is balanced across different servers (of a server farm), counterintuitively, it has been proved that SITA is more effective at reducing the mean slowdown when exactly the opposite is done [60]. Thus, SITA unbalances the load across the hosts and allows smaller tasks to run at lighter-loaded servers. It has been shown that with SITA, the more heavy-tailed

the workload is, the better is the load unbalanced across hosts, thus greatly improving the mean slowdown [28]. Further, as the job size variability increases, SITA is by far superior to the Least-Work-Left (LWL) policy, which sends jobs to the server with the least remaining work. Despite the general belief of being the better policy, it seems SITA may in fact be inferior to the previous greedy policy, for particular job size distributions [57]. Derived from the same fundamental idea of load unbalancing, TAGS achieves comparable performance to SITA in the more challenging case of unknown job sizes [55]. Closest to our work, there exists a thorough simulation-based analysis of SITA with respect to fairness under supercomputing workloads [102].

5.7 Conclusion

In this chapter we have presented four multi-queue size-based scheduling policies for data-intensive workloads derived from previous solutions to this problem for sequential or rigid jobs in single-server and distributed-server systems. The basic mechanisms employed by our policies are partitioning of resources of the datacenter, and system feedback by means of preemption in a work-conserving way. Hence, jobs with different ranges are isolated in separate queues or partitions, either by means of preemption from one queue to another (the TAGS and FBQ policies), or through some form of prediction (the SITA and COMP policies).

We have analyzed these policies with an extensive set of realistic simulations of MapReduce workloads and we have showed close to ideal improvement for the vast majority of short jobs even in unfavorable load conditions, while only a relatively small fraction of large jobs suffer (less than 5%). In particular, we have found that TAGS operates well for capacities of partition 1 between 20-50%, and that SITA offers considerable better slowdown performance when the size of partition 1 is small. FBQ is comparable to SITA, but at both lower median slowdown and slowdown variability, and unlike SITA, it is very insensitive to the queue time limit. Under heavy load, TAGS and COMP are by far the worst performing policies, while FBQ and SITA hurt significantly only the top 5% largest jobs in our workload. However, FBQ with 4 queues achieves a 30% improvement in median slowdown over the 2-queue setting.

We have found that FBQ consistently improves the slowdown variability over FIFO by a factor of 2 under system loads between 0.7 and 0.9. Thus, FBQ may be the best policy in practice, as it not only comes with the advantage of requiring the setting of fewer parameters than both TAGS and SITA, but also offers very even job slowdowns across the complete range of job sizes.

Finally, we have deployed our policies on a multicluster system and we have showed that the relative error between the simulations and the real-world experiments is less than 1% for both the median job slowdown and the job slowdown at the 95^{th} percentile.

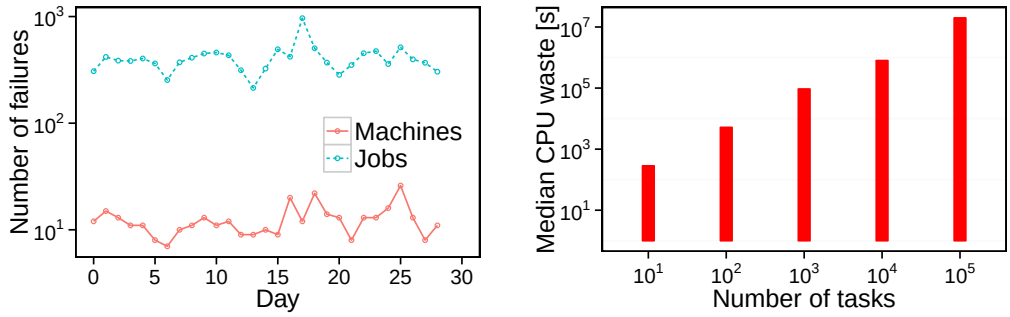
Chapter 6

Checkpointing In-Memory Data Analytics Applications

6.1 Introduction

The performance of large-scale data analytics frameworks such as Hadoop and Spark has received major interest [43, 44, 92, 139] from both academia and industry over the past decade. Surprisingly, this research assumes an ideal execution environment, which is in sharp contrast with the resilience-oriented design goals of these systems. In turn, these goals are motivated by the high rates of failures experienced by large-scale systems operating in clusters [71, 101] and datacenters [98, 132]. A key feature influencing the adoption of data analytics frameworks is their *fault-tolerant* execution model, in which a master node keeps track of the tasks that were running on machines that failed and restarts them from scratch on other machines. However, we face a fundamental limitation when the amount of work lost due to failure and re-execution is excessive because we need to allocate extra resources for recomputing work which was previously done. Frameworks such as Spark provide an API for checkpointing, but leave the decision of which data to checkpoint to the user. In this chapter we design PANDA, a cluster scheduler that performs automatic checkpointing and so improves the resilience of in-memory data analytics frameworks.

Failures in large-scale clusters are inevitable. The likelihood of having hardware crashes during the first year of a typical 10,000-machine cluster is very high according to several reports from the Google infrastructure team [29]. In particular, the system administrators expect about 1,000 individual machine failures and thousands of disk failures. In order to put into perspective the impact of failures on production workloads, we analyze failure reports from a Google cluster of 12,000 machines running half billion jobs over a month [98]. In Figure 6.1(a) we show the rate of machine and job failures in this Google cluster. Despite the relatively small number of machine failures (13 machines every hour),



(a) Failure rate

(b) CPU waste

Figure 6.1: The average number of job and machine failures per hour (a) and the median CPU waste per job size range (b) in the Google trace. The vertical axes are in log-scale.

we observe a huge number of jobs (400 jobs every hour) that either fail, get killed by the system, or are simply abandoned by users. We expect this large number of failures to result into large amounts of wasted work. In Figure 6.1(b) we show the median job waste, that is the amount of work completed but lost due to failures for the complete range of job sizes (number of tasks). Indeed, the amount of wasted work increases linearly with the job size. The Google infrastructure is only one of a long series of multicluster systems experiencing problems in their infancy and in the long term. For example, the grid computing community has uncovered high failure rates [33], and in particular the flagship project CERN LCG had high failure rates years after going into production, with more than 25% unsuccessful jobs across all sites [21].

As today's clusters have large amounts of free memory [77, 125], frameworks such as Spark advocate in-memory data processing, as opposed to previous on-disk approaches such as Hadoop. Unfortunately, as has been abundantly reported by the community, manipulating large datasets with Spark is challenging, and we have identified three causes of frequent failures in Spark that necessitate jobs to be restarted from scratch. First, the job runtime is very sensitive to the way the framework allocates the available memory during its execution. As a result, it may have variable performance across different applications depending on how much memory they are allowed to use for storage and for job execution [139]. A second cause is that several built-in operators (e.g., groupBy, join) require that all values for one key fit in the memory. This constraint is in sharp contrast with the design of the framework which only supports coarse-grained memory allocation (per worker). Finally, memory-hungry tasks that produce a large number of persistent objects that stay in memory during the task runtime result in expensive garbage collection [83].

Using checkpointing to improve fault tolerance has a long history in computer systems [111, 117, 143]. In particular, the most commonly used method for checkpointing high performance computing applications is coordinated checkpointing, where an appli-

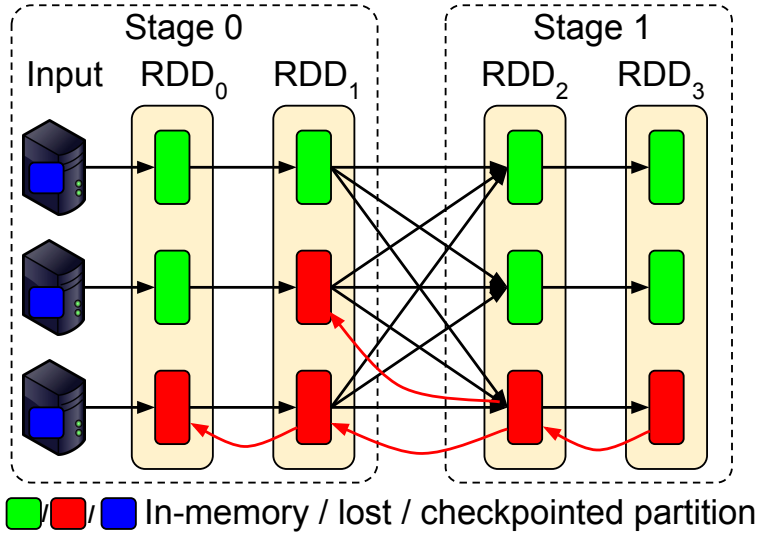
cation *periodically* stops execution and writes its current state to an external stable storage system. As setting the optimal checkpointing interval has been acknowledged as a challenging problem [35], existing solutions require the failure rates and the checkpointing cost to be known upfront, and to be constant over time. These assumptions are unrealistic for data analytics frameworks, which typically run computations in multiple interdependent stages each of which generates an *intermediate dataset* that is used as input by other stages. According to several reports from production clusters [24], the sizes of the intermediate datasets may vary significantly across stages of a single job, and as a result they cannot be anticipated.

Checkpointing a task has resource implications which are important to consider. While a task may be quickly recovered from a checkpoint, occupying an extra slot to perform the checkpoint may increase the job runtime due to the high cost of reliably saving the task's output. To remedy this, we propose PANDA, a checkpointing system that carefully balances the opportunity cost of persisting a task's output to an external storage system and the time required to recompute when the task is lost. This opportunity cost is driven by the evidence of *unpredictable intermediate data sizes* and *outlier tasks* of jobs in production traces from Google and Facebook [98, 115], which form the basis of our checkpointing policies. Firstly, we propose the *greedy* policy that greedily selects tasks for checkpointing until a predefined budget is exceeded. Secondly, our *size-based* policy considers the longest tasks of a job because those tasks are more likely to delay the job completion if they are lost. Finally, we design the *resource-aware* policy that checkpoints tasks only if their recomputation cost is likely to exceed the cost of checkpointing it.

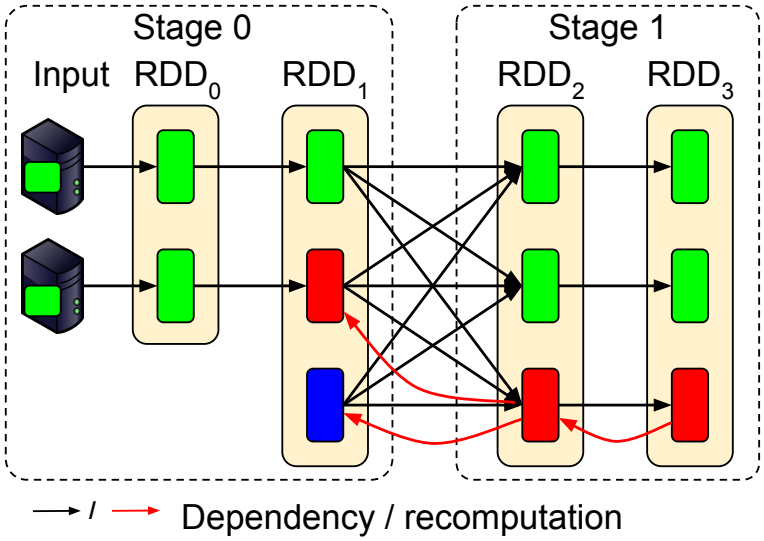
In this chapter we make the following contributions:

1. We design PANDA, a fine-grained checkpointing system that checkpoints tasks at stage boundaries by persisting their output data to stable storage. We reduce the checkpointing problem to a task selection problem and we incorporate into PANDA three policies designed from first principle analysis of traces from production clusters. These policies take into account the size of task output data, the distribution of task runtimes, or both (Sections 6.3 and 6.4).

2. With a set of experiments in a multicluster system, we analyze, among other aspects of PANDA, the benefit of trading off execution latency with fast recovery from failures for single, failing applications. With a set of large-scale simulations, mimicking the size and the operation of a Google cluster, we analyze the effectiveness of PANDA in reducing the average job runtime of a complete workload. (Sections 6.5 and 6.6).



(a) Recomputation



(b) Checkpointing

Figure 6.2: An example of a lineage graph with data dependencies between RDD partitions. The recomputation tree of a missing partition in unmodified Spark (a) and in Spark with checkpointing (b). All lost partitions are located on a single machine and the input dataset is replicated in stable storage.

6.2 System Model

In this section we present the main abstractions used by Spark to perform both efficient and fault-tolerant in-memory data processing (Section 6.2.1). Furthermore, we describe the scheduling mechanism employed by Spark to execute parallel jobs on a cluster with many machines (Section 6.2.2).

6.2.1 Lineage Graphs

We explain the RDD data abstraction used by Spark [146] to persist large datasets in the memory of multiple cluster machines and we discuss the notion of *lineage graph*, a fault-tolerant data structure that guards the framework against data loss when machine failures are expected.

Data analytics frameworks such as Spark [146] leverage the distributed memory of the cluster machines with a new abstraction called *resilient distributed datasets* (RDDs), which provides efficient data processing across a broad range of applications (SQL queries, graph processing, machine learning, and streaming). An RDD is a collection of *data partitions* distributed across a set of cluster machines. Users have access to a rich set of transformations (e.g., map, filter, join) to create RDDs from either data in stable storage (e.g., HDFS, S3) or other RDDs. Typically, such transformations are *coarse-grained* because they apply the same operation in parallel to each partition of the RDD.

RDDs may not be materialized in-memory at all times. Instead, Spark maintains the sequence of transformations needed to compute each RDD in a data structure called the *lineage graph*. In other words, the lineage graph is a directed acyclic graph (DAG) where a vertex represents an RDD partition and an incoming edge represents the transformation used to compute the RDD. Furthermore, Spark distinguishes two main types of data dependencies between RDDs: (1) the *narrow* dependency, in which each partition of the parent RDD is used by at most one partition of the child RDD (e.g., map, filter), and (2) the *wide* dependency, in which multiple child partitions may depend on the same parent partition (e.g., join, groupBy).

As RDDs are typically persisted in volatile memory without replicas, a machine failure causes the loss of all partitions that are located on it. Spark automatically recovers a missing partition by identifying in the lineage graph its *recomputation tree*, which is the minimum set of missing ancestor partitions and the dependencies among them needed to recover the partition. Thus, the *critical recomputation path* of a given partition is the sequence of partitions in its recomputation tree that determine the minimum time needed to recover the partition. In the worst case, the critical recomputation path may go back as far as the origin of the input data. Then, Spark applies for each missing partition the sequence of transformations in its recomputation tree according to the precedence

constraints among them. As different partitions of the same RDD may have different recomputation trees, the recovery of a complete RDD typically results in recomputing a sub-DAG of the initial lineage graph.

To avoid long critical recomputation paths, Spark allows its users to *cut-off* the lineage graph through a *checkpointing* operation that reliably saves a complete RDD to stable storage. Checkpointing an RDD in Spark is similar to how Hadoop spills shuffle data to disk, thus trading off execution latency with fast recovery from failures. Figure 6.2 shows an example of a lineage graph for a simple Spark computation, with both narrow and wide dependencies between RDDs. The figure depicts the recovery of a missing partition by recomputing all its ancestors (a) and by reading an existing checkpoint (b).

Spark exposes a basic interface for checkpointing complete RDDs, but it is the user's decision to select which RDDs to checkpoint. As the intermediate RDD sizes are not known upfront, selecting RDDs statically, prior to the execution of an application, is difficult. Spark checkpoints a given RDD by creating a parallel job with tasks that save the RDD partitions from memory to stable storage. However, when the memory is fully utilized, Spark evicts RDD partitions using a *least-recently-used* (LRU) policy. This way of checkpointing RDDs is inefficient because it may trigger recomputations if some RDD partitions are evicted from memory.

6.2.2 DAG Scheduler

We present an overview of the scheduling architecture used by Spark to (re-)allocate compute slots to jobs that consist of multiple sets of tasks with precedence constraints among them.

To compute an RDD, Spark's scheduler creates a job by translating the RDD dependencies in the lineage graph into a DAG of *processing stages*. Each stage consists of a set of *parallel tasks* that apply the same operation (transformation) to compute independently each RDD partition. In this DAG, tasks pipeline as many transformations with narrow dependencies as possible, and so we identify *stage boundaries* by transformations with wide dependencies. Such transformations typically require a *shuffle* operation, as illustrated in Figure 6.2. A shuffle operation splits the output partitions of each task in the *parent* stage into multiple shuffle files, one for each task in the *child* stage. Tasks in the child stage may only run once they have obtained all their shuffle files from the parent stage.

In order to compute an RDD, the scheduler executes tasks in successive stages on *worker* machines based on their precedence constraints (data dependencies), data locality preferences (run tasks closer to input data), or fairness considerations (per job quotas). Similarly to Dryad and MapReduce, Spark jobs are *elastic* (or *malleable*) and can run simultaneously, taking any resources (compute slots) they can get when it is their turn. The DAG scheduler in Spark schedules the tasks of a stage only after all its parent stages

Table 6.1: The workload traces from two large production clusters at Facebook [115] and Google [98].

Trace	Facebook	Google
Dates	October 2010	May 2011
Duration (days)	45	29
Framework	Hadoop	Borg
Cluster size (machines)	600	12,000
Number of jobs	25,000	668,048
Task runtimes	No	**Yes**
Data sizes	**Yes**	No
Failed machines per hour	Unknown	7 to 25

have generated their output RDDs. Scheduling tasks based on a strict queueing order such as *first-in-first-out* (FIFO) compromises locality, because the next task to schedule may not have its input data on the machines that are currently free. Spark achieves task locality through delay scheduling, in which a task waits for a limited amount of time for a free slot on a machine that has data for it.

Next, we present the main mechanisms that Spark uses to detect and to recover from worker failures. Similarly to other fault-tolerant cluster frameworks, Spark relies on time-outs and connection errors to infer worker failures. The scheduler expects heartbeats from its healthy *workers* every 10 seconds, and marks as lost a worker that has not sent any heartbeat for at least 1 minute. A dead worker not only leads to the failure of its running tasks, but also makes all previously computed work on that worker unavailable. As a consequence, tasks that fail to transfer data from a lost worker trigger fetch errors that may also serve as an early indication of a failure. Spark re-executes failed tasks as long as their stage's parents are still available. Otherwise, the scheduler resubmits tasks recursively in parent stages to compute the missing partitions.

6.3 Design Considerations

In this section we identify three techniques for checkpointing in-memory data analytics jobs (Section 6.3.1). Moreover, we investigate the main properties of workloads from Facebook and Google that we use as first principles in the design of our checkpointing policies (Section 6.3.2). Finally, we propose a scheduling and checkpointing structure for automatic checkpointing of data analytics jobs (Section 6.3.3).

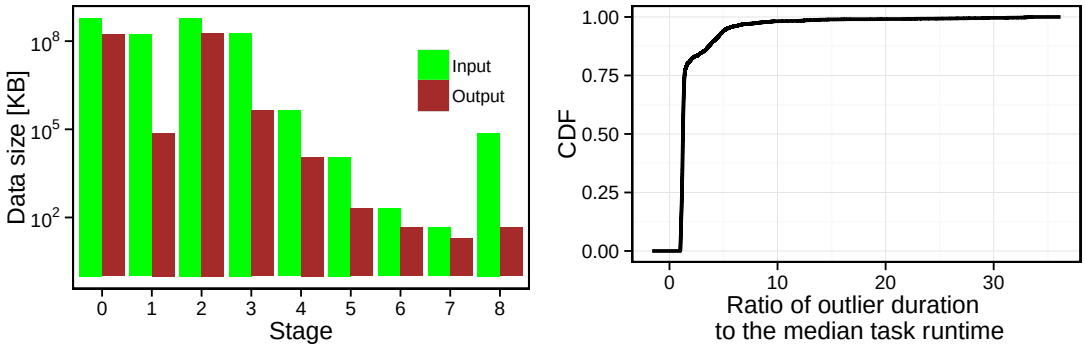
6.3.1 Checkpointing Tasks

The basic fault-tolerance mechanism used by data analytics frameworks to mitigate the impact of machine failures is to recompute lost data by repeating tasks based on their precedence constraints in the lineage graph. Obviously, this approach may be time-consuming for applications with large lineage graphs. Checkpointing the running tasks of a job to stable storage allows the job to only partially recompute data generated since the last checkpoint. However, checkpointing introduces an overhead proportional to the size of the data persisted to stable storage.

We identify different ways of checkpointing data analytics jobs. One way of of doing so is to employ traditional checkpointing mechanisms available in operating systems that suspend the execution of running tasks and store their states for later resumption. In this method, checkpointing jobs is performed *at any point*, as opposed to the later two approaches. However, this process may degrade performance considerably and may trigger frequent machine reboots [67]. Tasks of in-memory data analytics jobs are allocated large heap sizes of multiple GBs, and so checkpointing their states is relatively slow. In addition, recovery from a checkpoint stored on another machine triggers additional network traffic, which may hurt the performance of other jobs in the cluster.

Another approach is to checkpoint tasks at *safe points* from where the remaining work can be executed without requiring any context from the current execution. At a higher level, tasks in data analytics jobs pipeline a sequence of narrow transformations between successive RDD partitions. Furthermore, tasks split each RDD partition they process into a sequence of *non-overlapping subsets* each of which may have multiple records that share the same *key*. Thus, a natural way to checkpoint tasks for many transformations (e.g., map, reduce, join) is at key boundaries, when all the processing for a key is complete. This approach has been previously proposed in Amoeba, a system that aims at achieving true elasticity by trimming the durations of long task through checkpointing. However, because tracking such safe points in data analytics frameworks is notoriously difficult, as they typically require a global view of intermediate data, Amoeba originally supported only a small number of transformations in MapReduce frameworks and has not evolved since [5].

Finally, we can checkpoint tasks at *stage boundaries* by persisting their *output data* to stable storage. A stage boundary for a task is the point from where the output data is split into multiple shuffle files each of which aggregates input data for a single reducer. Shuffle files are written to the buffer cache, thus allowing the operating system to flush them to disk when the buffer capacity is exceeded. Because checkpointing shuffle files requires complex synchronization between multiple tasks that write sequentially to the same shuffle file, we perform checkpointing on the output data before splitting it into shuffle files. We choose this way of checkpointing because it integrates well with the lineage-based mechanism adopted by current frameworks. We need to recompute a task when either the



(a) Data sizes

(b) Task runtimes

Figure 6.3: The variability of the intermediate data sizes (a, vertical axis in log scale) and the prevalence of outliers (b) in the BTWorld application.

machine on which it runs fails, or when (part of) the output it produced was located on a machine that fails and is still needed. Checkpointing tasks at stage boundaries helps only in the latter case. After checkpointing the output of a task completely, we no longer need to know how to compute or recover its input, and so we can cut-off its lineage graph.

6.3.2 Task Properties

Although there is a rich body of work that studies the characteristics of datacenter workloads [24, 97], not many public traces exist. The largest traces available are from the Hadoop production cluster at Facebook [115] and from Google's Borg resource manager [98]. Table 6.1 shows the relevant details of these traces. An investigation of these traces reveals that datacenter workloads are largely dominated by the presence of outlier tasks, and that the sizes of the intermediate data of jobs may be very variable. Although both traces are relatively old, it is unlikely that these task properties have changed since their collection. In this section, we check their validity by analyzing the BTWorld application [61], which we use to process *monitoring data* from the BitTorrent global network; this application is later described in Section 6.5.

Unlike a job's input size, which is known upfront, intermediate data sizes cannot be anticipated. Complex applications such as BTWorld consist of many processing stages, out of which only a few require the complete input, while the others run on intermediate data. Figure 6.3(a) shows that there is no strong correlation between the input and output data sizes, and that the output sizes range from a few KB to hundreds of GB. We compute the stage selectivity, defined as the ratio of the output size and the input size, for each job in the Facebook trace. We find that the stage selectivities may span several orders of magnitude: a small fraction of the stages perform data transformations (selectivity of

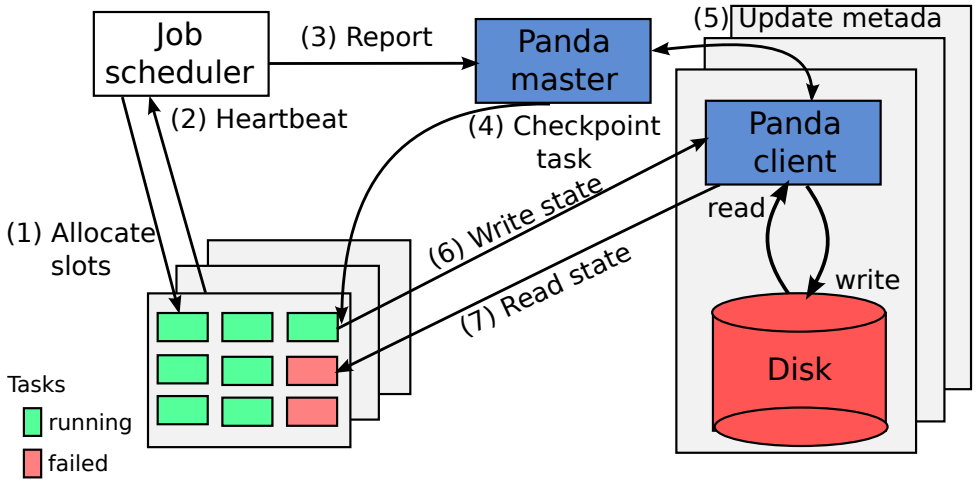


Figure 6.4: The system architecture for implementing the PANDA checkpointing mechanism in data analytics frameworks.

1), while the large majority are either data compressions (selectivity less than 1) or data expansions (selectivity higher than 1).

In data analytics workloads, tasks may have inflated runtimes due to poor placement decisions (resource contention) or imbalance in the task workload (input data skew). Indeed, Figure 6.3(b) shows that 70% of the task outliers in the BTWorld application have a uniform probability of being delayed between 1.5x and 3x the median task runtime. The distribution is heavy-tailed, with top 5% of the outliers running 10x longer than the median. Similarly, the tasks in the Google cluster are also very variable and fit well a heavy-tailed distribution (Pareto with shape parameter 1.3).

We use the large variability of the intermediate data sizes in the design of a *greedy* checkpointing policy, which employs a specified budget to avoid excessive checkpointing. Similarly, the prevalence of outliers forms the basis of a *size-based* checkpointing policy, which seeks to checkpoint the long running tasks in a job. Finally, we use both properties in a *resource-aware* checkpointing policy, which checkpoints tasks only when the cumulative cost of recomputing them is larger than the cost of checkpointing. In Section 6.4 we present the design of our policies starting from first principles, with all the features needed to perform well in a datacenter.

6.3.3 Checkpointing Architecture

We present the main design elements and the operation of PANDA, an adaptive checkpointing system for in-memory data analytics jobs which integrates well with the architecture of current framework schedulers.

Figure 6.4 shows the architecture of a typical data analytics framework, with a cluster-wide *job scheduler* and a fault-tolerant *distributed filesystem* which coordinate the execution of tasks on a set of cluster machines with co-located processors and storage volumes (illustrated for simplicity as separate entities). The job scheduler handles the allocation of compute slots to numerous parallel tasks of a data analytics job with user-defined constraints (step 1) and waits for periodic heartbeats to keep track of the state of the running tasks (step 2). The distributed filesystem (e.g., HDFS in our deployment) employs a three-way replication policy for fault-tolerance and allows our system to *reliably persist* a data analytics job by saving its input, intermediate, or output datasets.

PANDA's architecture consists of a *checkpoint master* and a set of *clients* located at each cluster machine. The PANDA master is periodically updated by the job scheduler with progress reports of the running tasks (step 3). A progress report incorporates for each task the following properties: the amount of input/output data size read/written so far and the current task runtimes. The master's main role is to decide *when* to checkpoint running tasks and *which* among the running tasks of a job to checkpoint (step 4). To do so, PANDA employs one of the policies presented in Section 6.4.

The checkpoint master receives updates from clients with the location of each checkpoint in the reliable storage system and maintains a global mapping between every checkpointed partition and the dataset it belongs to (step 5). The PANDA clients access the distributed filesystem for saving and/or fetching partitions on behalf of the job (steps 6 and 7). Thus, before a task starts running, it first uses the PANDA client to retrieve from the checkpoint master the location of its checkpoint. The PANDA client fetches the checkpoint from the distributed filesystem so that the task gracefully resumes its execution from that point onwards. If the task was not previously checkpointed, it executes its work completely.

6.4 Checkpointing Policies

We will now address the question of which subsets of tasks to checkpoint in order to improve the job performance under failures while keeping the overhead of checkpointing low. The policies we propose for this purpose may use the size of the task output data (GREEDY), the distribution of the task runtimes (SIZE), or both (AWARE). Furthermore, we use an adaptation of the widely known periodic checkpointing approach (PERIODIC) to data analytics frameworks that periodically checkpoints all completing tasks. In Table 6.2 we state the main differences between our policies.

6.4.1 Greedy Checkpointing

Our GREEDY policy seeks to limit the checkpointing cost in every stage of a job in terms of the amount of data persisted to disk to a specified *budget*. Intuitively, we want to reduce

Table 6.2: PANDA's policy framework for checkpointing in-memory data analytics jobs in datacenters.

Policy	Data size	Task runtime	Description
GREEDY	yes	no	fraction of the input data
SIZE	no	yes	longest tasks in the job
AWARE	yes	yes	checkpoint vs. recompute
PERIODIC	yes	no	every τ seconds

the number of recomputations after a failure in a *best-effort* way by selecting in each stage as many tasks for checkpointing as the budget allows. The GREEDY policy sets the checkpointing budget of a stage to some fraction of the size of the total input data transferred to it from the tasks of its parent stages. This fraction may depend on the selectivities of the tasks of the stage—if the latter are low, the fraction can be small. For example, for the BTWorld workflow with a median task selectivity of 0.1, it can be set to 10%.

The GREEDY policy is invoked for every stage of a job once all its parent stages have generated their output RDDs. It will then start checkpointing *any* completing task as long as it does not exceed the stage's budget. Tasks that are in the process of checkpointing when the budget is exceeded are allowed to complete their checkpoints.

6.4.2 Size-based Checkpointing

Our SIZE policy aims to reduce the amount of work lost after a failure by checkpointing *straggler* tasks that run (much) slower than other tasks of the job. The main intuition behind the SIZE policy is to avoid recomputing time-consuming tasks that prevent pending tasks of the job from starting.

Straggler tasks in data analytics frameworks may be due to large variations in the *code* executed and the *size of the data* processed by tasks. Across all stages of the BTWorld workflow, the coefficient of variation in task runtimes is 3.4. Although the code is the same for all tasks in each stage, it differs significantly across stages (e.g., map and reduce). Furthermore, the amount of data processed by tasks in the same stage may vary significantly due to limitations in partitioning the data evenly.

The SIZE policy now works in the following way. In order to differentiate straggler tasks, SIZE builds up from scratch for every running job a history with the durations of its finished tasks. Thus, at any point in time during the execution of a job, SIZE has an estimation of its median task runtime, which becomes more accurate as the job completes a larger fraction of its tasks. SIZE checkpoints only those tasks it considers stragglers, that is, tasks whose durations are at least some number of times (called the *task multiplier*) as high as the current estimation of the median task runtime.

6.4.3 Resource-aware Checkpointing

The AWARE policy aims to checkpoint a task only if the estimated benefit of doing so outweighs the cost of checkpointing it. We explain below how the AWARE policy estimates both the recomputation and the checkpointing cost of a task, which is done after it has completed.

Prior to the execution of a job, AWARE sets the probability of failure by dividing the number of machines that experienced failures during a predefined time interval (e.g., a day) by the cluster size. AWARE derives these data from the operation logs of the cluster that contain all machine failing events.

A machine failure may cause data loss, which may require recomputing a task if there are pending stages that need its output in order to run their tasks. However, the recomputation of a task may cascade into its parent stages if its inputs are no longer available and need to be recomputed in turn. We define the DAG level of a task as the length of the longest path in the lineage graph that needs to be recomputed to recover the task from a failure.

AWARE estimates the recomputation cost of a task as the product of the probability that the machine on which it ran fails and its *recovery time*, which is the actual cost of recomputing it, including the recursive recomputations if its recomputation cascades into its parent stages. When the input files of a lost task are still available, either in the memory of other machines or as checkpoints in stable storage, the recovery time is equal to the task runtime. If multiple input files of a task to be recomputed are lost, we assume that they can be recomputed in parallel, and we add the maximum recomputation cost among the lost tasks in its parent stages to its recovery time. We do this recursively as also input files of tasks in parent stages may be lost in turn.

The checkpointing cost of a task is a function of the amount of data that needs to be persisted, the write throughput achieved by the local disks the task is replicated on, and the contention on the stable storage caused by other tasks that are checkpointed at the same time. While the former two may be anticipated, the latter is highly variable and difficult to model accurately. In particular, checkpointing a task along with other tasks that require replicating large amounts of data to stable storage may inflate the checkpointing cost. In order to solve this problem, we propose the following method to approximate the checkpointing cost of a task in a given stage. When a stage starts, we artificially set the cost of checkpointing its tasks 0, thus making AWARE checkpoint the first few waves of tasks in order to build up a partial distribution of task checkpointing times. Then we let AWARE set the checkpointing cost of a task to the 95^{th} percentile of this distribution, which is all the time adapted with the checkpointing times of the checkpointed tasks in the stage.

The AWARE policy now works in the following way. It is invoked whenever a stage becomes eligible for scheduling its tasks, and then, using the job's lineage graph, it estimates

the recomputation costs of its tasks. We amortize the checkpointing cost of a task by its DAG level, so that tasks with long recomputation paths are more likely to be checkpointed. AWARE checkpoints only those tasks whose potential resource savings are strictly positive, that is, tasks whose recomputation costs exceed the amortized checkpointing cost.

6.5 Experimental Setup

We evaluate the checkpointing policies described in Section 6.4 through experiments on the DAS multicluster system. In this section, we present the cluster configuration and the data analytics benchmarks that we use to assess the performance of PANDA.

6.5.1 Cluster Setup

We have implemented PANDA in Spark and we evaluate our checkpointing policies on the fifth generation of the Dutch wide-area computer system DAS [119]. In our experiments we use DAS machines that have dual 8-core compute nodes, 64 GB memory, and two 4 TB disks, connected within the cluster through 64 Gbit/s FDR InfiniBand network. To provision machines for our experiments we use the SLURM scheduling system [109] deployed on DAS. We perform experiments with two cluster configurations for long and short jobs with allocations of 20 and 5 machines, respectively.

We co-locate PANDA with an HDFS instance that we use to store the input datasets and the checkpoints performed by our policies. We setup the HDFS instance with a standard three-way replication scheme and a fixed data block size of 128 MB. We assume the HDFS instance runs without failures so that both the input datasets and the checkpoints are always available for PANDA.

We want to analyze the performance of typical data analytics applications under different patterns of *compute node failures*. Therefore, we consider Spark *worker failures*, which may cause loss of work already done that is stored in the local memory of the workers. We assume that new worker machines may be provisioned immediately to replace the lost workers, so that the size of our cluster remains constant during the execution of the application.

We clear the operating system buffer cache on all machines before each experiment, so that the input data is loaded from disk. To emulate a production environment with long-running processes, we warm up the JVM in all our experiments by running a full trial of the complete benchmark. For the experiments we show in Section 6.6, we report the mean over three executions.

Table 6.3: The cluster configurations for our applications.

Application	Benchmark	Nodes	Dataset	Input [GB]	Runtime [s]
BTWorld	real-world	20	BitTorrent	600	1587
PPPQ	standard	20	TPC-H	600	1461
NMSQ	standard	20	TPC-H	600	656
PageRank	real-world	5	Random	1	128
KMeans	real-world	5	Random	10	103

6.5.2 Applications

In our evaluation we use a diverse set of applications ranging from real-world workflows to standard benchmarks that are representative for data analytics frameworks. Table 6.3 presents the configuration we use in our experiments for each job. We analyze the performance of PANDA with both long-running jobs that have durations in the order of tens of minutes (e.g., BTWorld, PPPQ, and NMSQ) and short interactive jobs that take minutes to complete (e.g., PageRank and KMeans). We describe these jobs in turn.

BTWorld. The BTWorld [137] application has observed since 2009 the evolution of the global-scale peer-to-peer system BitTorrent, where files are broken into hashed pieces and individually shared by users, whether they have completely downloaded the file or not. To help users connect to each other, BitTorrent uses trackers, which are centralized servers that give upon request lists of peers sharing a particular file. BTWorld sends queries to public trackers of the BitTorrent system and so it collects statistics about the aggregated status of users. These statistics include for each *swarm* in the tracker (users who share the same torrent file) the number of *leechers* (users who own some but not all pieces of the file), the number of *seeders* (users who own all pieces of the file), and the total number of *downloads* since the creation of the torrent.

We have designed a MapReduce-based workflow [61] to answer several questions of interest to peer-to-peer analysts in order to understand the evolution over time of the BitTorrent system. The complete BTWorld workflow seeks to understand the evolution of each individual tracker we monitor, to determine the most popular trackers (over fixed time intervals and over the entire monitoring period), and to identify the number of new swarms created over time. In our experiments with PANDA, BTWorld takes an input dataset of 600 GB. As we show in Figure 6.5(a), the lineage graph of BTWorld consists of a long chain of stages and a single join.

TPC-H. The TPC-H benchmark [123] consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the dataset have been chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implementation. This benchmark illustrates decision support systems that ana-

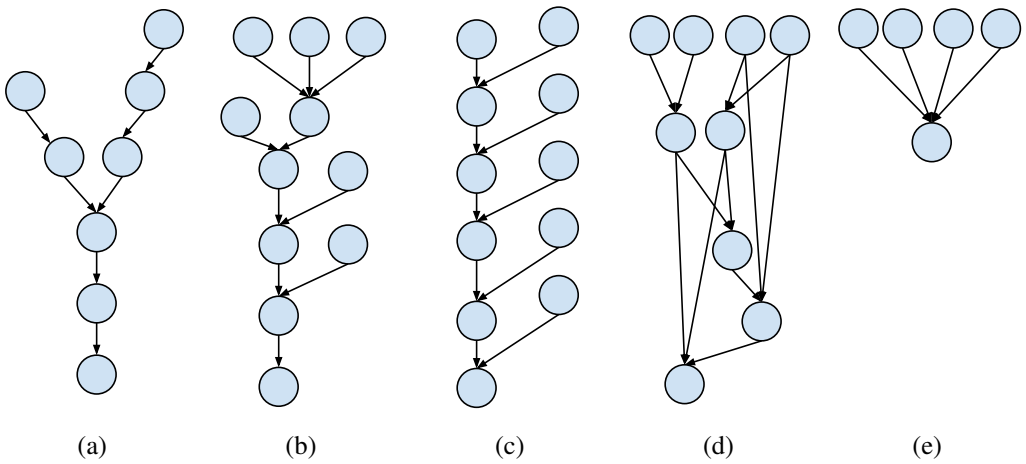


Figure 6.5: The data flow of BTWorld (a), PPPQ (b), NMSQ (c), PageRank (d), and KMeans (e) as a DAG of stages: the nodes and the edges represent the stages and the wide dependencies among them.

lyze large volumes of data, execute queries with high degrees of complexity, and give answers to critical business questions. The benchmark randomly generates eight relational tables with a schema that represents a typical data warehouse dealing with sales, customers, and suppliers. For a detailed description of the benchmark we refer to its standard specification [123].

We use two queries from this benchmark: the Potential Part Promotion Query (PPPQ) and the National Market Share Query (NMSQ). PPPQ seeks candidates for a promotional offer by selecting suppliers in a particular nation that have an excess of a given product – more than 50% of the products shipped in a given year for a given nation. NMSQ determines how the market share of a given nation within a given region has changed over two years for a given product. In our experiments we execute both queries with an input dataset of 600 GB. Figures 6.5(b) and 6.5(c) show the lineage graphs of both TPC-H queries that combine in almost every stage results from two or three parent stages.

PageRank. PageRank is the original graph-processing application used by the Google search engine to rank documents. PageRank runs multiple iterations over the same dataset and updates the rank of a document by adding the contributions of documents that link to it. On each iteration, a document sends its contribution of r_i/n_i to its neighbors, where r_i and n_i denote its rank and number of neighbors, respectively. Let c_{ij} denote the contribution received by a document i from its neighbor j. After receiving the contributions from its neighbors, a document i updates its rank to $r_i = (\alpha/N) + (1 - \alpha) \sum c_{ij}$, where N is the total number of documents and α a tuning parameter.

We use the optimized PageRank implementation from the `graphx` library of Spark with a 1 GB input dataset. We generate a random input graph with 50,000 vertices that has a log-normal out-degree distribution with parameters μ and σ set to 4 and 1.3, re-

Table 6.4: An overview of the experiments performed to evaluate PANDA.

Experiment	Jobs	Policies	Baselines	Failure pattern	Section
Parameters	BTWorld PPPQ	all	none	none	6.6.1
Overhead	all	all	Spark	none	6.6.2
Machine failures	BTWorld PPPQ NMSQ	all	Spark	space-correlated	6.6.3
Lineage length	PageRank KMeans	AWARE	Spark	single failure	6.6.4
Simulations	BTWorld PPPQ PageRank	AWARE	Spark	space-correlated	6.6.5

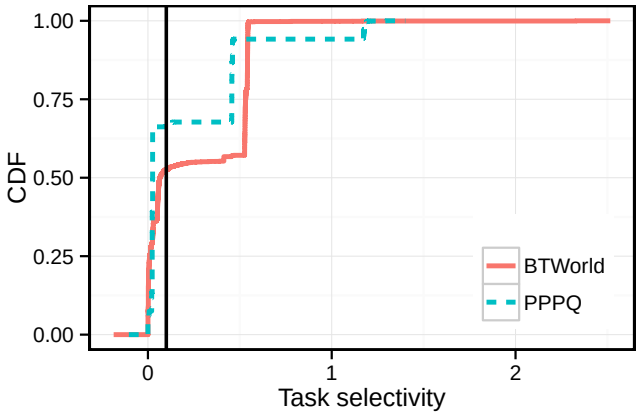
spectively. In Figure 6.5(d) we show the lineage graph for a single iteration of PageRank. An interesting property of this application is that its lineage becomes longer with the number of iterations.

KMeans. Clustering aims at grouping subsets of entities with one another based on some notion of similarity. KMeans is one of the most commonly used clustering algorithms that clusters multidimensional data points into a predefined number of clusters. KMeans uses an iterative algorithm that alternates between two main steps. Given an initial set of means, each data point is assigned to the cluster whose mean yields the least *within-cluster sum of squares* (WCSS). In the update step, the new means that become the *centroids* of the data points in the new clusters are computed.

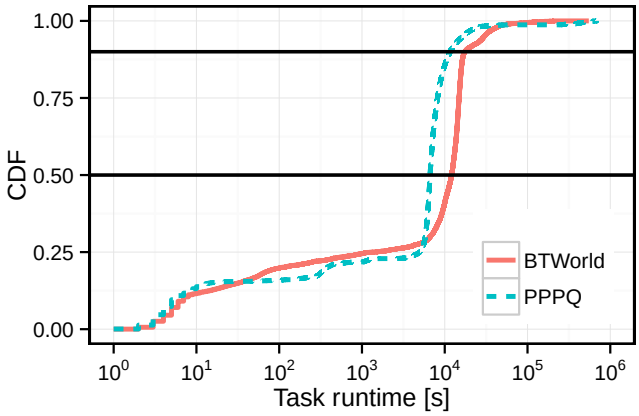
We use the optimized implementation from the `mllib` library of Spark with a 10 GB dataset that consists of 10 millions data points sampled from a 50-dimensional Gaussian distribution. Figure 6.5(e) shows that KMeans with four iterations has a relatively simple lineage graph, with a single shuffle operation that combines results from multiple stages that have narrow dependencies to the input dataset.

6.6 Experimental Evaluation

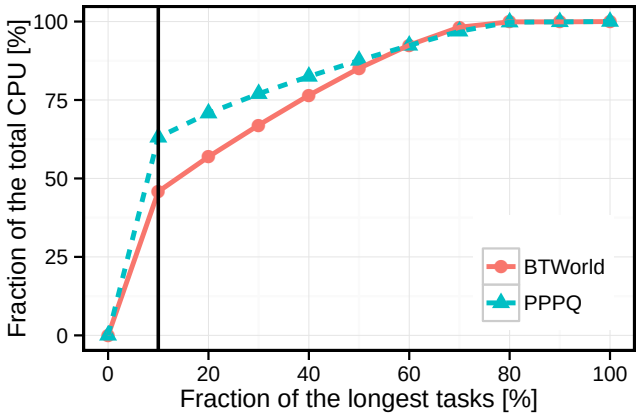
In this section we present the results of five sets of experiments that each address a separate aspect of the performance of PANDA. Table 6.4 presents an overview of these experiments. We investigate the setting of the parameters in GREEDY, SIZE, and AWARE (Section 6.6.1). We measure the checkpointing overhead to determine how far we are from the default Spark implementation without checkpointing (Section 6.6.2). Thereafter, we evaluate the performance of our policies under various patterns of space-correlated fail-



(a) Task selectivities



(b) Task runtimes



(c) Processing time

Figure 6.6: The distributions of the task selectivity (a) and the task runtime (b) for BTWorld and PPPQ, and the fraction of the total CPU time versus the fraction of the longest tasks in BTWorld and PPPQ (c). The vertical lines represent the task selectivity of 0.1 (a) and the longest 10% tasks (c), and the horizontal lines represent the median and the 90^{th} percentile of the task runtimes (b).

ures (Section 6.6.3). Moreover, we assess the impact of the length of the lineage graph on the performance of PANDA when failures are expected (Section 6.6.4). Finally, we perform simulations to evaluate the benefit of checkpointing at larger scale (Section 6.6.5).

6.6.1 Setting the Parameters

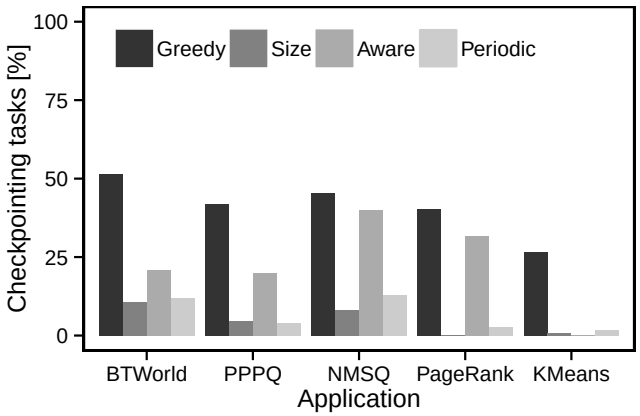
All our policies have parameters needed in order to operate in a real environment. In particular, GREEDY and SIZE use the checkpointing budget and the task multiplier, respectively. In contrast with these policies that both set workload-specific parameters, AWARE sets the probability of failure that quantifies the reliability of the machines, and so it is independent of the workload properties. In this section we seek to find good values of these parameters.

One way of setting the parameters for GREEDY and SIZE is to evaluate the performance of each policy for a range of parameter values with various failure patterns. However, this method is time-consuming, and may in practice have to be repeated often. To remove the burden of performing sensitivity analysis for each policy, we propose two simple rules of thumb based on the history of job executions in production clusters and in our DAS multicluster system. We show in Sections 6.6.2 through 6.6.5 that our policies perform well with these rules.

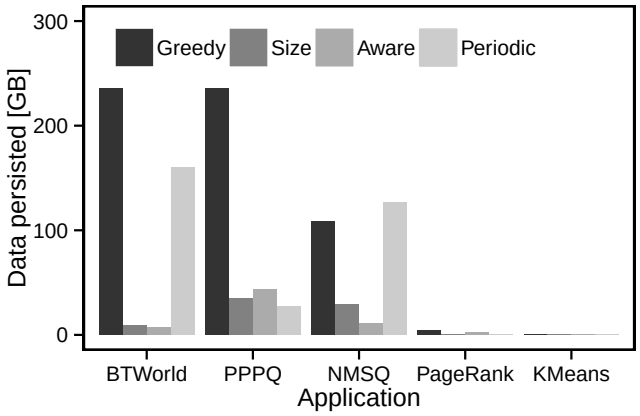
The GREEDY *policy sets the checkpointing budget to the median selectivity of the tasks across all jobs that we use in our experiments.* The checkpointing budget limits the amount of data that is replicated to HDFS in each stage. We expect GREEDY to have a large overhead when setting the checkpointing budget to a large value. In Section 6.3.2 we have shown that in the Facebook production cluster, a large majority of tasks have relatively low selectivities. Figure 6.6(a) shows the distributions of the task selectivity in BTWorld and PPPQ. Because the median task selectivity is below 0.1 for both jobs, we set the checkpointing budget in our experiments with GREEDY to 10%.

The SIZE *policy sets the task multiplier to the ratio of the* 90^{th} *percentile and the median of the runtimes of the tasks across all jobs that we use in our experiments.* The task multiplier aims at identifying the longest tasks in a job, and so setting a small value may result in checkpointing a large fraction of tasks. As we have shown in Section 6.3.2, this is unlikely to happen for data analytics jobs because they typically run tasks that have heavy-tailed durations. Figure 6.6(c) shows that only 10% of tasks in BTWorld and PPPQ account for roughly 50% of the total processing time of the job. As Figure 6.6(b) shows, the ratio of the 90^{th} percentile and the median of the distribution of task runtimes for both BTWorld and PPPQ is 1.5. Thus, we set the task multiplier in our experiments with SIZE to 1.5.

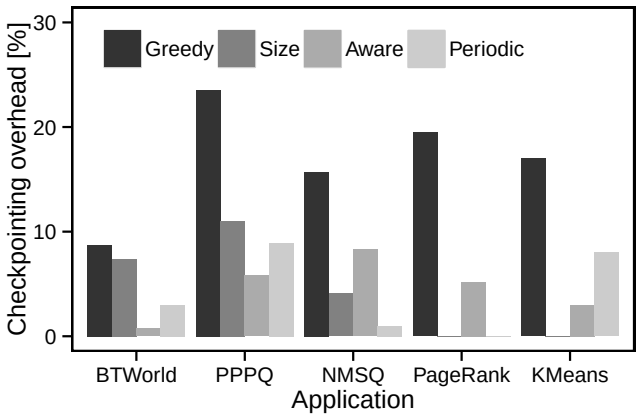
Unlike the previous two policies, which both require an analysis of task properties, the AWARE policy only needs as parameter the likelihood of being hit by a failure. We want AWARE to checkpoint more tasks as it operates on less reliable machines and vice versa.



(a) Checkpointing tasks



(b) Checkpointing storage



(c) Checkpointing overhead

Figure 6.7: The fraction of checkpointing tasks (a), the amount of data persisted to HDFS (b), and the checkpointing overhead (c) with all our policies. The baseline is unmodified Spark.

In order to highlight the checkpointing overhead and the performance of AWARE in unfavorable conditions, we assume that all machines allocated to execute our jobs experienced failures. Thus, we set the probability of failure in our experiments with AWARE to 1.

Finally, in order to show the improvements provided by our policies relative to the traditional way of checkpointing, in our experiments with the PERIODIC policy we set the optimal checkpointing interval based on Young's approximation for each application. To do so, our version of the PERIODIC policy requires an estimation of the checkpointing cost and the mean time to failure. Thus, we assume that we know prior to the execution of each job both its checkpointing cost and the failure time.

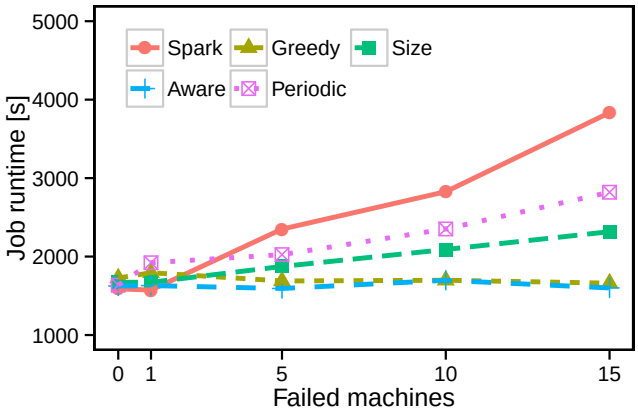
6.6.2 The Impact of the Checkpointing Overhead

Spark has been widely adopted because it leverages memory-locality, and so it achieves significant speedup relative to Hadoop. Because checkpointing typically trades-off performance for reliability, we want to evaluate how far the performance of PANDA is from the performance of unmodified Spark when it runs on reliable machines. In this section we evaluate the overhead due to checkpointing tasks in PANDA relative to the performance of unmodified Spark without failures and without checkpointing.

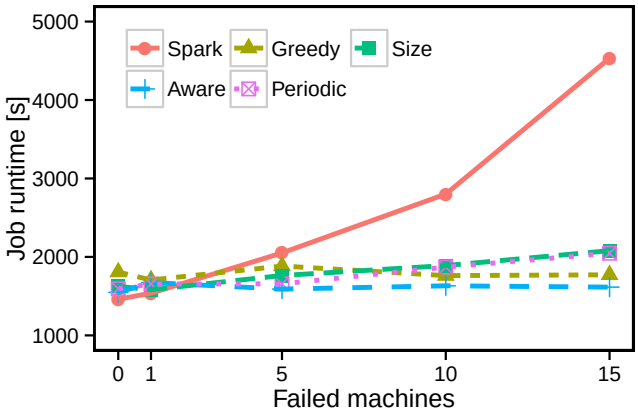
Unlike previous approaches to checkpointing that typically save periodically the intermediate state of an application, PANDA reduces the checkpointing problem to a task selection problem. Therefore, we first want to assess how selective our policies are in picking their checkpointing tasks. To this end, in Figure 6.7(a) we show the number of tasks that are checkpointed by each policy for all applications. We find that GREEDY is rather aggressive in checkpointing tasks, while SIZE is the most conservative policy. In particular, we observe that with the GREEDY policy, PANDA checkpoints between 26-51% of the running tasks in our applications. Further, because the SIZE policy targets only the outliers, it checkpoints at most 10% of the running tasks for all applications. Similarly to SIZE, our adaptation of the PERIODIC policy checkpoints relatively small fractions of tasks for all applications.

Because AWARE balances the recomputation and the checkpointing costs for each task, the number of checkpointing tasks is variable across different jobs. We observe that for jobs that have relatively small intermediate datasets such as NMSQ and PageRank, AWARE checkpoints roughly 40% of the tasks. However, for BTWorld and PPPQ, which both generate large amounts of intermediate data, AWARE is more conservative in checkpointing and so it selects roughly 20% of the tasks.

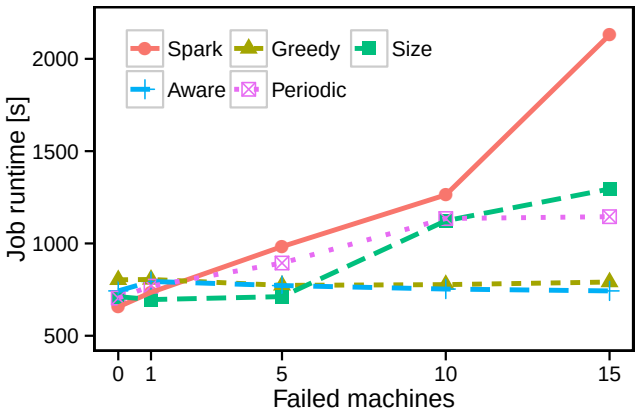
PANDA assumes the presence of an HDFS instance to persist its checkpoints that is permanently available. Running many large applications in a cluster may lead to significant amount of storage space used by PANDA. In Figure 6.7(b) we show the amount of data persisted by our policies in each application. We find that GREEDY checkpoints signifi-



(a) BTWorld



(b) PPPQ



(c) NMSQ

Figure 6.8: The job runtime versus the number of failures with all our policies for BTWorld, PPPQ, and NMSQ. The baseline is Spark with its default lineage-based recomputation.

cantly more data than both SIZE and AWARE for all applications. In particular, GREEDY replicates 30 times as much data as SIZE and AWARE with BTWorld. To perform the checkpoints of all four applications, GREEDY requires a storage space of 1.8 TB (including replicas), whereas SIZE and AWARE require 212 GB and 190 GB, respectively. We also find that despite being rather conservative in selecting its checkpointing tasks, the PERIODIC policy requires a storage space of 1 TB.

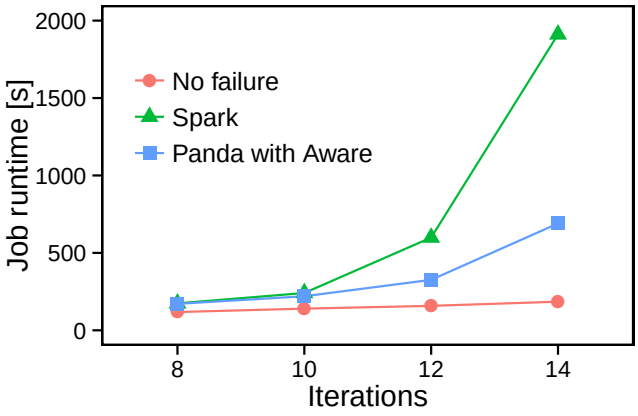
Finally, we assess the checkpointing overhead of our policies as a percentage increase in the job runtime relatively to a vanilla version of Apache Spark (see Table 6.4). Figure 6.7(c) shows the checkpointing overhead for all four applications. GREEDY suffers significant performance degradation and its checkpointing overhead may be as high as 20%. However, both SIZE and AWARE incur less than 10% overhead, and so they are very close to the performance of Spark without checkpointing. This result can be explained by what is the main difference between our policies. Whereas GREEDY checkpoints tasks in a best-effort way, both SIZE and AWARE employ more conservative ways of selecting checkpointing tasks based on outliers or cost-benefit analysis. Further, because PERIODIC performs its checkpoints at fixed intervals during the application runtime, the contention on the HDFS is relatively low at all times. As a consequence, although PERIODIC checkpoints significantly more data than both SIZE and AWARE, they all have similar overheads.

We conclude that both SIZE and AWARE deliver very good performance and they are close to the performance of Spark without checkpointing. These policies are very selective when picking checkpointing tasks, they use relatively small storage space to persist the output data of tasks, and they incur a checkpointing overhead that is usually below 10%.

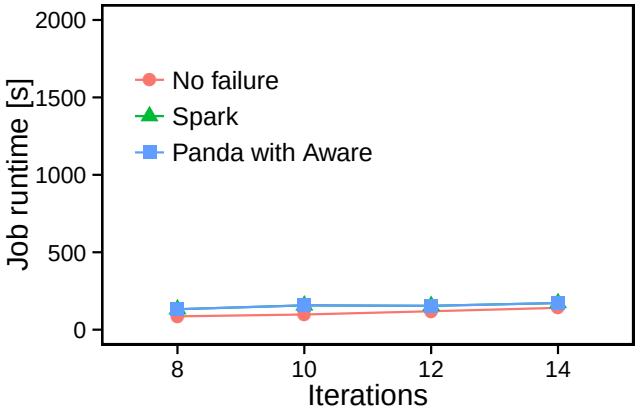
6.6.3 The Impact of the Machine Failures

Space-correlated failures, defined as groups of machine failures that occur at the same time across the datacenter, have been frequently reported in large-scale systems such as grids and clusters [101], and more recently in datacenters [98, 104]. Therefore, in this section, we evaluate the performance of PANDA under space-correlated failures. To this end, we report in Figure 6.8 the job runtime with and without our checkpointing policies for a range of concurrent failures that occur in the last processing stage of our BTWorld, PPPQ, and NMSQ applications. As a hint of reading this figure, the values at 0 machine failures represent the job runtimes with our policies and with Spark when the job completes without experiencing failures.

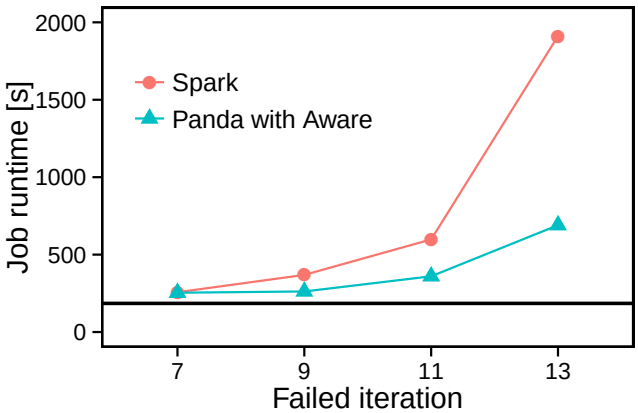
Without checkpointing, the recomputation time due to failures causes a significant performance degradation for the entire range of concurrent failures. We observe that the job runtime in unmodified Spark increases linearly with the number of concurrent failures for all applications. For example, when only 25% of the cluster size is lost due



(a) PageRank



(b) KMeans



(c) PageRank

Figure 6.9: The job runtime when a single machine fails before the job completes for PageRank and KMeans with different numbers of iterations (a), (b), and the job runtime when a single machine fails in a given iteration for PageRank with 14 iterations (c). The horizontal line represents the job runtime without failures in unmodified Spark for PageRank with 14 iterations.

to failures, the job runtime increases by 48% for BTWorld, and by 40% for the two TPC-H queries. For the stress test we consider with 15 out of 20 machines that fail, Spark delivers very poor performance with all applications completing between 2.5 and 3 times as slow as when no failures occur.

Figure 6.8 also shows that all our checkpointing policies deliver very good performance for the complete range of failures. Both GREEDY and AWARE provide constant runtimes irrespective of the number of failures for all applications. The reason for this result is that they cut-off the lineage graph at key stages, thus avoiding recomputing previously completed work. We also observe that AWARE performs slightly better than GREEDY because it introduces a lower checkpointing overhead, as we have shown in Figure 6.7. SIZE also reduces the impact of failures, but the job runtime still increases linearly with the number of failures. However, SIZE performs at its best and gets very close to the performance of GREEDY and AWARE for jobs such as PPPQ that have outlier tasks with very long durations. In particular, we have shown in Figure 6.6(c) that in PPPQ only 10% of the tasks account for more than 60% of the total processing time. Further, we find that PERIODIC has poor performance for BTWorld, but its performance is very close to the performance of SIZE for the two TPC-H queries.

We conclude that our policies outperform unmodified Spark for any number of space-correlated failures. While both GREEDY and AWARE provide constant job runtimes for the complete range of machine failures, AWARE is our best policy because it introduces a much lower overhead than GREEDY.

6.6.4 The Impact of the Lineage Length

Although Spark may use the job lineage graph to recover lost RDD partitions after a failure, such recovery may be time-consuming for jobs such as PageRank that have relatively long lineage chains with many wide dependencies (see its DAG in Figure 6.5(d)). Conversely, applications such as KMeans, in which narrow dependencies prevail, may be recovered relatively fast from data in stable storage (see its DAG in Figure 6.5(e)). In this section we seek to highlight the impact of the lineage graph structure on the job runtime with and without checkpointing in the presence of a single machine failure that occurs at different moments during the job execution.

Figures 6.9(a) and 6.9(b) show the differences in performance of PANDA with the AWARE policy on the job runtime when a single machine fails before the job completes for different numbers of iterations of PageRank and KMeans. We observe that PageRank completes relatively fast for Spark without failures for the complete range of iterations from 8 to 14. However, the performance of PageRank degrades significantly even when a single failure perturbs the last iteration of the job. In particular, the job runtime is 11 times as large as the fail free execution in Spark for PageRank with 14 iterations. Because

PageRank requires many shuffle operations, a machine failure may result in the loss of some fraction of data from each parent RDD, thus requiring a long chain of recomputations. Figure 6.9(a) also shows that PANDA with the AWARE policy performs very well and bounds the recomputation time for any number of iterations. Not only does PANDA complete the job four times as fast as the recomputation-based approach in Spark, its performance is also very close to the performance of Spark without failures.

Unlike PageRank, which suffers significantly from failures, we show in Figure 6.9(b) that KMeans is rather insensitive to faulty machines and that Spark is less than 5% off the fail-free execution for any number of iterations. The reason for this result can be explained by what is the main difference between the lineage graphs of PageRank and KMeans. As we have shown in Section 6.5.2, the length of the PageRank lineage graph is proportional to the number of iterations of the job, and so the amount of recomputations triggered by a single failure grows significantly for jobs with many iterations. In contrast with PageRank, KMeans has a much simpler lineage graph, with many narrow dependencies followed by a shuffle, and so its length remains constant irrespective of the number of iterations. As a consequence, checkpointing KMeans is not worthwhile, because these narrow dependencies may be quickly recovered from stable storage. However, because our AWARE policy avoids checkpointing stages that are one hop away from the input dataset, we observe that its operation falls back to default Spark in the case of KMeans.

Finally, Figure 6.9(c) shows the results of experiments in which a single machine fails at different moments during the job execution for PageRank with 14 iterations. In general, we observe that Spark performs well when the failure occurs in the early stages of the job. However, it delivers poor performance when the time of failure is closer to the job completion time. We find that PANDA is effective in reducing the recovery time and outperforms Spark irrespective of the time of failure.

We conclude that applications such as PageRank that have many wide dependencies are good candidates for checkpointing, while machine learning applications such as KMeans may be recovered with relatively small recomputation cost. Furthermore, we have shown that PANDA performs very well for lineage graphs in which the recomputation cost is excessive irrespective of the time of failure.

6.6.5 The Impact of the Failure Pattern

So far, we have evaluated different aspects of the operation of PANDA with single applications that experience space-correlated failures at a certain time during their execution. We have shown that our policies deliver very good performance with relatively small checkpointing overhead. However, it is not clear whether the improvements hold for a long-running system when multiple jobs receive service in the cluster only a fraction of

Table 6.5: The distribution of job types in our simulated workload.

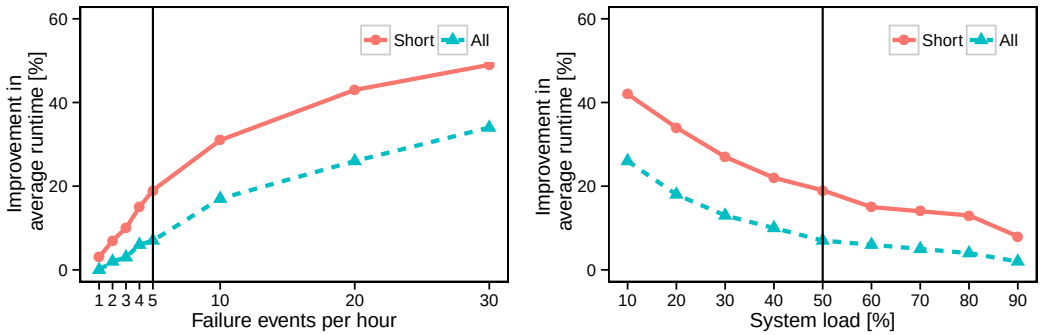
Application	Total nodes	Failed nodes	Scale	Runtime [s]	Jobs [%]
BTWorld	20	5	1.0	1587	16
	20	5	5.0	7935	5
	20	5	10.0	15870	5
PPPQ	20	5	1.0	1461	16
	20	5	5.0	7305	5
	20	5	10.0	14610	5
PageRank (14 iterations)	5	1	1.0	185	16
	5	1	5.0	925	16
	5	1	10.0	1850	16

which experience failures. Thus, we want to evaluate the improvement in the average job runtime achieved by AWARE for a complete workload relative to unmodified Spark.

We have built our own simulator in order to evaluate the impact of the frequency of failures on the overall improvement of PANDA with the AWARE policy. We simulate the execution of a 3-day workload on a 10,000-machine cluster (similar to the size of the Google cluster discussed in Section 6.1). We perform simulations at a higher-level than the earlier single-application experiments, and so we use the overall job durations (with or without failures) from experiments rather than simulating the execution of separate tasks.

In our event-based simulator, jobs are submitted according to a Poisson process and they are serviced by a FIFO scheduler. The scheduler allocates to each job a fixed number of machines which are released only after the job completes. Although the Google trace consists of mostly short jobs in the order of minutes, the longest jobs may take hours or even days to complete. To generate a similar realistic workload, we scale up the durations of our jobs by different scaling factors as shown in Table 6.5. Table 6.5 also shows the distribution of the job types in our workload. In particular, 80% of the jobs complete within 30 minutes (short jobs), whereas the durations of the remaining jobs exceed 2 h (long jobs).

In order to reuse the results from the experiments, we create the following failure pattern. We assume that failures occur according to a Poisson process that may hit every job at most once. Each failure in our simulation is a space-correlated failure event that triggers the failure of 5 machines. Table 6.5 shows for every job in our workload the number of failed machines when it is hit by a failure event. In particular, a failure event may hit a single BTWorld query, a single PPPQ query, or 5 different PageRank jobs. To simulate the execution of a job that is hit by a failure we replace the job runtime given in Table 6.5 by the job runtime shown in Figure 6.8 or 6.9 multiplied by the job's scaling factor. Our



(a) Failure rate

(b) System load

Figure 6.10: The improvement achieved by PANDA with AWARE for short jobs and for the complete workload under a system load of 50% for different values of the failure rate (a), and for a failure rate of 5 under different values of the system load (b).

assumption here is that both the checkpointing overheads and the improvements in the job runtime achieved by AWARE hold irrespective of the scaling factor of the job. Similarly to the experiments we performed on the DAS, we report averages over three simulations.

Figure 6.10(a) shows the results of the simulations for different values of the failure rate under a system load of 50%, which is the average utilization of the Google cluster. PANDA provides significant gains when machines are more likely to fail but may not be worthwhile when failures occur rarely. Intuitively, jobs are more likely to be hit by failures when the failure rate is high. In particular, the fraction of failed jobs increases from 0.7% to 24% when the failure rate increases from 1 to 30 failure events per hour. We find that PANDA with AWARE reduces the average job runtime with 34% relative to the execution in unmodified Spark when the failure rate is 30 per hour. However, PANDA stops being beneficial when the cluster experiences less than one failure event per hour. Furthermore, Figure 6.10(a) shows that the improvement for short jobs is significantly higher than the overall improvement for our workload. For example, for a failure rate of 5, which is equivalent to the maximum failure rate in the Google cluster (see Table 6.1), short jobs improve by 19% on average, whereas the overall improvement is only 7%.

Finally, in Figure 6.10(b) we show the results of the simulations for different values of the system load when 5 failure events are expected every hour. We find that PANDA provides significant improvements over the complete range of system loads, but becomes less beneficial under high loads. The intuition of this result is that failed jobs account for larger fractions of the total number of jobs when the system load is low. In particular, the fraction of failed jobs decreases from 18% to 2.5% when the system load increases from 10% to 90%.

6.7 Related Work

In this section we present different methods of achieving resilience in high-performance computing (HPC) systems and data analytics frameworks.

BlobCR [89] seeks to efficiently capture and roll-back the state of scientific HPC applications in public clouds. Recent work [35] analyzed practical methods for optimizing the checkpointing interval using real-world failure logs. Multi-level checkpointing [32] aims at reducing the overhead of checkpointing in large-scale platforms by setting different levels of checkpoints each of which has its own overhead and recovery capability. An adaptive checkpointing scheme with work migration [142] has been developed to minimize the cost of running applications on resources from spot markets. Similar techniques that aim to reduce the checkpointing overhead of the naive periodic checkpointing policy exploit the temporal locality in failures [122].

Closest to our work, TR-Spark [141] and Flint [105] propose checkpointing policies for data analytics applications that run on transient resources which are typically unstable, but not necessarily due to faults. In particular, TR-Spark employs cycle-scavenging to leverage such transient resources which are kept idle as a resource buffer by cloud providers and may be revoked due to load spikes. TR-Spark takes a statistical approach to prioritize tasks that have a high probability of being completed before the resources where they run are revoked and to checkpoint data blocks that are likely to be lost before they are processed by the next processing stages. To do so, TR-Spark requires both the distribution of the task runtimes and the distribution of the inter-arrival failure time for each resource allocated to the framework. Similarly, Flint provisions instances available on the spot market which have relatively low prices and may be revoked due to price spikes. Flint supports RDD-level checkpointing using an adaptation of the periodic checkpointing policy to data analytics applications. In contrast, PANDA targets a datacenter environment where applications may suffer from outright node failures and proposes a more comprehensive set of task-level checkpointing policies that take into account not only the lineage structure of the applications, but also workload properties such as the task runtimes and the intermediate data sizes.

6.8 Conclusion

The wide adoption of in-memory data analytics frameworks is motivated by their ability to process large datasets efficiently while sharing data across computations at memory speed. However, failures in datacenters may cause long recomputations that degrade the performance of jobs executed by such frameworks. In this chapter we have presented PANDA, a checkpointing system for improving the resilience of in-memory data analytics frameworks that reduces the checkpointing problem to a task selection problem. We

have designed three checkpointing policies starting from first principles, using the size of the task output data (GREEDY), the distribution of the task runtimes (SIZE), or both (AWARE). The GREEDY policy employs a best-effort strategy by selecting as many tasks for checkpointing as a predefined budget allows. The SIZE policy checkpoints only straggler tasks that run much slower than other tasks of the job. The AWARE policy checkpoints a task only if the cost of recomputing it exceeds the time needed to persist its output to stable storage.

With a set of experiments on a multicluster system, we have analyzed and compared these policies when applied to single failing applications. We have found that our policies outperform both unmodified Spark and the standard periodic checkpointing approach. We have also analyzed the performance of PANDA with the AWARE policy by means of simulations using a complete workload and the failure rates from a production cluster at Google. We have found that PANDA is beneficial for a long-running system and can significantly reduce the average job runtime relative to unmodified Spark. In particular, the SIZE policy delivers good performance when the failure rate (fraction of failed machines per day) is relatively low (less than 6%). Although both GREEDY and AWARE turn out to provide significant improvements for a large range of failure rates (more than 6%), AWARE is our best policy because it introduces a much lower overhead than GREEDY. However, when the datacenter is prone to failures of complete racks, the rather aggressive checkpointing strategy of the GREEDY policy may be worthwhile.

Chapter 7

Conclusion

In this thesis we have sought to optimize the performance of data analytics frameworks. To this end, we have focused on different aspects of the datacenter scheduling architecture, from allocation of resources to multiple frameworks, to scheduling jobs within those frameworks, to handling compute node failures. We have addressed these challenges of the datacenter scheduling problem by combining fundamental and experimental research.

We have consistently built our solutions by starting from first principles derived from analytic results in basic theoretical models and by designing appropriate adaptations for the evolving usage trends. To address each challenge we have proposed policies that are presumably elegant, widely applicable, and highly implementable. We have shown that our solutions lead to significant improvements in the performance of modern data analytics frameworks. In order to do so, we have taken an experimental approach by analyzing the performance of prototype systems on the DAS multicluster system using real-world workloads. The rich set of experiments we have conducted on the DAS provide valuable insights into the performance attained by our solutions. To better understand the experimental results and to bridge the gap between the analysis of those experiments and prior theoretical work, we also performed large-scale simulations of scheduling policies.

This chapter presents the main contributions that offer answers to the research questions outlined in the introduction and suggests several future research directions.

7.1 Lessons Learned

We present the main conclusions which contribute to our answers to the three challenges in datacenters of *multi-tenancy* (Chapters 2 and 3), *workload heterogeneity* (Chapters 4 and 5), and *failures* (Chapter 6) that we have identified in Chapter 1. These conclusions provide insights into the performance of data analytics frameworks in datacenters.

1. We have shown with a comprehensive set of experiments in Chapters 2 and 3 that dynamic (re-)allocation of resources is the key to achieve performance isolation of frameworks in datacenters.

2. The two-level scheduling architecture we have presented in Chapter 2 is a simple yet effective way of allocating resources to multiple frameworks in datacenters that enables four types of isolation with respect to data, version, failures, and performance. Moreover, growing and shrinking single frameworks leads to significant improvements relatively to static deployments.

3. With the weighting mechanism we have designed in Chapter 3 it is possible to provide balanced service levels across multiple frameworks. In order to differentiate frameworks, we have used feedback of the three stages of a system: the input, the usage, and the output. We have further shown that feedback on the input to the system is the best way of balancing the service levels of multiple frameworks.

4. Current framework schedulers usually lead to job slowdowns of small jobs that are at least an order of magnitude larger than those of long jobs. With the multi-queueing models we have explored in Chapters 4 and 5 it is possible to reduce the job slowdown variability in frameworks.

5. We have shown in Chapter 4 with typical data analytics workloads that confining jobs to smaller partitions so that no large job monopolizes the framework outperforms state-of-the-art schedulers with respect to both job slowdown variability and median job slowdown.

6. We have bridged the gap between analytical and experimental results by simulating in Chapter 5 a class of size-based scheduling policies in single and distributed server systems with appropriate adaptations to data analytics frameworks. We have found that multi-queueing models may be an extremely powerful tool for optimizing the performance in datacenters. Furthermore, we have demonstrated that feedback queueing is more effective in reducing the job slowdown variability in data analytics frameworks than load unbalancing.

7. As we have shown in Chapter 6, lineage graphs employed by in-memory data analytics frameworks provide fault-tolerance but may be time-consuming for large applications. Checkpointing the application cuts off the lineage graph and reduces the recovery path when the framework experiences failures. Furthermore, fine-grained checkpointing by carefully trading off the costs of persisting and recomputing a task results in better performance than the static approach of checkpointing complete datasets.

7.2 Future Directions

Much work remains on optimizing the performance of data analytics frameworks, resulting out of both theoretical and practical limitations. We will present some of the remaining open questions.

1. In Chapters 2 and 3, with our experiments with the scheduling system for frameworks, we have assumed for simplicity only MapReduce frameworks. A natural extension to our evaluation would be to explore the performance of the resource balancing mechanism with diverse frameworks. As different types of frameworks may have conflicting optimization goals, we believe this work will be interesting, and it will pose additional challenges to address.

2. In Chapters 2 and 3 we have studied the problem of allocating the resources of a single datacenter across multiple data analytics frameworks. Although many large companies own tens of datacenters, current data analytics frameworks do not satisfy the low latency requirements of running queries on geographically distributed datasets. Therefore, it would be interesting to design policies for scheduling data analytics workloads across multiple datacenters.

3. In Chapters 4 and 5 we have assumed datacenters with homogeneous resources, jobs with loose placement preferences that may run anywhere, and fairness as the most important optimization goal. In practice, datacenters can be rather heterogeneous, can execute jobs that have hard locality constraints, and can require fast service times. Therefore, we identify as an important future work to explore which are the appropriate policies that account for all these factors.

4. In Chapters 4 and 5 we have designed policies for centralized schedulers that launch all jobs through a single node. As datacenter sizes grow and the demand for low-latency interactive data processing increases, decentralized schedulers are increasingly prominent. Thus, an interesting future direction is to propose policies for reducing the job slowdown variability in frameworks by scheduling jobs from a set of machines that operate autonomously.

5. In Chapter 6 we have studied the problem of checkpointing in-memory data analytics applications in order to improve their resilience after failures. Current job schedulers typically assume that each job runs a fixed set of tasks, thus ignoring their need to perform checkpoints or to recompute missing partitions after failures. Designing a job scheduler that jointly optimizes the resource allocation and fault-tolerance mechanisms would be an interesting future work.

Bibliography

[1] C. L. Abad, M. Yuan, C. X. Cai, Y. Lu, N. Roberts, and R. H. Campbell. Generating request streams on big data using clustered renewal processes. *Performance Evaluation*, 70(10):704–719, 2013.

[2] Amazon EMR. `http://aws.amazon.com/elasticmapreduce`.

[3] Amazon Web Services. `https://aws.amazon.com/products/`.

[4] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *ACM European Conference on Computer Systems (EuroSys)*, pages 287–300, 2011.

[5] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-tenant Data-intensive Compute Clusters. In *ACM Symposium on Cloud Computing (SoCC)*, page 24, 2012.

[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 185–198, 2013.

[7] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: coordinated memory caching for parallel jobs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 20–20, 2012.

[8] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 289–302, 2014.

[9] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In

USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 265–278, 2010.

[10] Apache Hadoop. `http://hadoop.apache.org`.

[11] Apache Hadoop Capacity Scheduler. `https://goo.gl/Soa56J`.

[12] Apache Hadoop Fair Scheduler. `https://goo.gl/xT0Ekm`.

[13] M. Asay. Why the world's largest Hadoop installation may soon become the norm. `https://goo.gl/DDLPgc`.

[14] B. Avi-Itzhak and H. Levy. On Measuring Fairness in Queues. *Advances in Applied Probability*, 36(3):919–936, 2004.

[15] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *IEEE Computer*, 49(5):54–63, 2016.

[16] N. Bansal and M. Harchol-Balter. Analysis of SRPT Scheduling: Investigating Unfairness. In *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 279–290, 2001.

[17] L. A. Barroso, J. Clidaras, and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[18] G. Brumfiel et al. Down the Petabyte Highway. *Nature*, 469(20):282–283, 2011.

[19] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *VLDB Endowment*, 3(1-2):285–296, 2010.

[20] J. Buisson, O. Sonmez, M. Hashim, W. Lammers, and D. Epema. Scheduling Malleable Applications in Multicluster Systems. In *IEEE Conference on Cluster Computing (CLUSTER)*, pages 372–381, 2007.

[21] CERN LHC Computing Grid. `http://wlcg-public.web.cern.ch/`.

[22] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis. In *ACM European Conference on Computer Systems (EuroSys)*, pages 43–56, 2012.

[23] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *VLDB Endowment*, 5(12):1802–1813, 2012.

[24] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance using Workload Suites. In *IEEE Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 390–399, 2011.

[25] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. N. Tantawi, and C. Krintz. See Spot Run: Using Spot Instances for MapReduce Workflows. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 7–7, 2010.

[26] W. Cirne and F. Berman. Adaptive Selection of Partition Size for Supercomputer Requests. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 187–207, 2000.

[27] E. G. Coffman Jr, R. R. Muntz, and H. Trotter. Waiting Time Distributions for Processor-Sharing Systems. *Journal of the ACM (JACM)*, 17(1):123–130, 1970.

[28] M. E. Crovella, M. Harchol-Balter, and C. D. Murta. Task Assignment in a Distributed System: Improving Performance by Unbalancing Load. Technical report, Boston University Computer Science Department, 1997.

[29] J. Dean. Designs, Lessons and Advice from Building Large Distributed Systems. *Keynote from LADIS*, page 1, 2009.

[30] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 10–10, 2004.

[31] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM (CACM)*, 51(1):107–113, 2008.

[32] S. Di, Y. Robert, F. Vivien, and F. Cappello. Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(1), 2017.

[33] C. L. Dumitrescu, I. Raicu, and I. Foster. Experiences in RunningWworkloads over Grid3. In *International Conference on Grid and Cooperative Computing*, pages 274–286, 2005.

[34] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative MapReduce. In *ACM Symposium on High Performance Distributed Computing (HPDC)*, pages 810–818, 2010.

[35] N. El-Sayed and B. Schroeder. Checkpoint/Restart in Practice: When 'Simple is Better'. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 84–92, 2014.

[36] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit. On Advantages of Grid Computing for Parallel Job Scheduling. In *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 39–39, 2002.

[37] Facebook to Expand Prineville Data Center. `http://goo.gl/fJAoU`.

[38] Fawkes Resource Manager. `http://www.ds.ewi.tudelft.nl/ghit/projects/fawkes/`.

[39] L. Fei, B. Ghit, A. Iosup, and D. Epema. KOALA-C: A Task Allocator for Integrated Multicluster and Multicloud Environments. In *IEEE Conference on Cluster Computing (CLUSTER)*, pages 57–65, 2014.

[40] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 1–34, 1997.

[41] E. J. Friedman and S. G. Henderson. Fairness and Efficiency in Web Server Protocols. In *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 229–237, 2003.

[42] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[43] B. Ghit and D. Epema. Reducing Job Slowdown Variability for Data-Intensive Workloads. In *IEEE Symposium Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 61–70, 2015.

[44] B. Ghit and D. Epema. Tyrex: Size-based Resource Allocation in MapReduce Frameworks. In *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 11–20, 2016.

[45] B. Ghit, N. Yigitbasi, and D. Epema. Resource Management for Dynamic MapReduce Clusters in Multicluster Systems. In *Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS) in conjunction with ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1252–1259, 2012.

[46] B. Ghit, N. Yigitbasi, A. Iosup, and D. Epema. Balanced Resource Allocations Across Multiple Dynamic MapReduce Clusters. In *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 329–341, 2014.

[47] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 24–24, 2011.

[48] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints. In *ACM European Conference on Computer Systems (EuroSys)*, pages 365–378, 2013.

[49] I. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks. In *ACM European Conference on Computer Systems (EuroSys)*, pages 57–70, 2012.

[50] Google Datacenter Locations. `https://goo.gl/Vgdgbf`.

[51] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, pages 51–62, 2009.

[52] A. G. Greenberg and N. Madras. How Fair is Fair Queuing. *Journal of the ACM (JACM)*, 39(3):568–598, 1992.

[53] B. Guenter, N. Jain, and C. Williams. Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 1332–1340, 2011.

[54] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *IEEE Parallel and Distributed Processing Symposium (IPDPS)*, pages 395–404, 2014.

[55] M. Harchol-Balter. Task Assignment with Unknown Duration. *Journal of the ACM (JACM)*, 49(2):260–288, 2002.

[56] M. Harchol-Balter, M. E. Crovella, and C. D. Murta. On Choosing a Task Assignment Policy for a Distributed Server System. *Journal of Parallel and Distributed Computing (JPDC)*, 59(2):204–228, 1999.

[57] M. Harchol-Balter, A. Scheller-Wolf, and A. R. Young. Surprising Results on Task Assignment in Server Farms with High-variability Workloads. In *ACM Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE)*, pages 287–298, 2009.

[58] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based Scheduling to Improve Web Performance. *ACM Transactions on Computer Systems (TOCS)*, 21(2):207–233, 2003.

[59] M. Harchol-Balter, K. Sigman, and A. Wierman. Asymptotic Convergence of Scheduling Policies with Respect to Slowdown. *Performance Evaluation*, 49(1):241–256, 2002.

[60] M. Harchol-Balter and R. Vesilo. To Balance or Unbalance Load in Size-Interval Task Allocation. *Probability in the Engineering and Informational Sciences*, 24(2):219, 2010.

[61] T. Hegeman, B. Ghiţ, M. Capotă, J. Hidders, D. Epema, and A. Iosup. The BTWorld Use Case for Big Data Analytics: Description, MapReduce Logical Workflow, and Empirical Evaluation. In *IEEE Conference on Big Data*, pages 622–630, 2013.

[62] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-Intensive Analytics. In *ACM Symposium on Cloud Computing (SoCC)*, pages 18:1–18:14, 2011.

[63] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 295–308, 2011.

[64] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *New Frontiers in Information and Software as Services*, pages 209–228, 2011.

[65] E. Hyytiä, S. Aalto, and A. Penttinen. Minimizing Slowdown in Heterogeneous Size-aware Dispatching Systems. In *ACM Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE)*, pages 29–40, 2012.

[66] A. Iosup, N. Yigitbasi, and D. Epema. On the Performance Variability of Production Cloud Services. In *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 104–113, 2011.

[67] M. Isard. Autopilot: Automatic Data Center Management. *ACM SIGOPS Operating Systems Review*, 41(2), 2007.

[68] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

[69] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 261–276, 2009.

[70] M. Isard and Y. Yu. Distributed Data-Parallel Computing using a High-Level Programming Language. In *ACM Conference on Management of Data (SIGMOD)*, pages 987–994, 2009.

[71] B. Javadi, D. Kondo, A. Iosup, and D. Epema. The Failure Trace Archive: Enabling the Comparison of Failure Measurements and Models of Distributed Systems. *Journal of Parallel and Distributed Computing (JPDC)*, 73(8):1208–1223, 2013.

[72] P. R. Jelenkovic, X. Kang, and J. Tan. Adaptive and Scalable Comparison Scheduling. In *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 215–226, 2007.

[73] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–103, 2010.

[74] Koala MapReduce Runner. `http://www.ds.ewi.tudelft.nl/koala/using-koala/mrrunner`.

[75] S. Krishnan, M. Tatineni, and C. Baru. myHadoop-Hadoop-on-Demand on Traditional HPC Resources. *San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego*, 2011.

[76] J. Leverich and C. Kozyrakis. On the Energy (in) Efficiency of Hadoop Clusters. *ACM SIGOPS Operating Systems Review*, 44(1):61–65, 2010.

[77] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM Symposium on Cloud Computing (SoCC)*, pages 1–15, 2014.

[78] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar. CAM: A Topology Aware Minimum Cost Flow Based Resource Manager for MapReduce Applications in the Cloud. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 211–222, 2012.

[79] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic Environments. In *ACM Symposium on High-Performance Distributed Computing (HPDC)*, pages 95–106, 2010.

[80] M. Lin, A. Wierman, L. L. Andrew, and E. Thereska. Dynamic Right-Sizing for Power-Proportional Data Centers. *IEEE/ACM Transactions on Networking (TON)*, 21(5):1378–1391, 2013.

[81] M. Lin, L. Zhang, A. Wierman, and J. Tan. Joint Optimization of Overlapping Phases in MapReduce. *Performance Evaluation*, 70(10):720–735, 2013.

[82] H. Liu. Cutting MapReduce Cost with Spot Market. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[83] Z. Liu and T. E. Ng. Leaky Buffer: A Novel Abstraction for Relieving Memory Pressure from Cluster Data Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(1), 2017.

[84] Miller, Rich. Google Uses About 900,000 Servers. `https://goo.gl/edqOjR`.

[85] H. Mohamed and D. Epema. KOALA: A Co-Allocating Grid Scheduler. *Concurrency and Computation: Practice and Experience*, 20(16):1851–1876, 2008.

[86] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, 2013.

[87] A. Murthy. Mumak: Map-Reduce Simulator. `https://goo.gl/ajy4XB`.

[88] T. Nguyen and M. Vojnovic. Weighted Proportional Allocation. In *ACM Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE)*, pages 173–184, 2011.

[89] B. Nicolae and F. Cappello. BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds Using Virtual Disk Image Snapshots. In *ACM Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 34, 2011.

[90] K. Normandeau. What is Driving the US Market? https://goo.gl/ASfNR6.

[91] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *ACM Conference on Management of Data (SIGMOD)*, pages 1099–1110, 2008.

[92] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making Sense of Performance in Data Analytics Frameworks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–307, 2015.

[93] Panda Checkpointing System. http://www.ds.ewi.tudelft.nl/ghit/projects/panda.

[94] M. Pastorelli, A. Barbuzzi, D. Carra, M. Dell'Amico, and P. Michiardi. HFSP: Size-based Scheduling for Hadoop. In *IEEE Conference on Big Data*, pages 51–59, 2013.

[95] D. A. Patterson. The Data Center Is The Computer. *Communications of the ACM (CACM)*, 51(1):105–105, 2008.

[96] D. Raz, H. Levy, and B. Avi-Itzhak. A Resource-allocation Queueing Fairness Measure. In *ACM Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE)*, pages 130–141, 2004.

[97] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *ACM Symposium on Cloud Computing (SoCC)*, page 7, 2012.

[98] C. Reiss, J. Wilkes, and J. Hellerstein. Google Cluster Data. https://goo.gl/0N5ZaU.

[99] L. Schrage. The Queue M/G/1 with Feedback to Lower Priority Queues. *Management Science*, 13(7):466–474, 1967.

[100] L. E. Schrage and L. W. Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research*, 14(4):670–684, 1966.

[101] B. Schroeder and G. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[102] B. Schroeder and M. Harchol-Balter. Evaluation of Task Assignment Policies for Supercomputing Servers: The Case for Load Unbalancing and Fairness. *Cluster Computing*, 7(2):151–161, 2004.

[103] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *IEEE Conference on Data Engineering (ICDE)*, pages 60–60, 2006.

[104] M. Sedaghat, E. Wadbro, J. Wilkes, S. De Luna, O. Seleznjev, and E. Elmroth. DieHard: Reliable Scheduling to Survive Correlated Failures in Cloud Data Centers. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 52–59, 2016.

[105] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *ACM European Conference on Computer Systems (EuroSys)*, page 6, 2016.

[106] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 349–362, 2012.

[107] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[108] A. Singh et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, pages 183–197, 2015.

[109] Slurm Manager. `https://slurm.schedmd.com/quickstart.html`.

[110] D. R. Smith. Technical note—a new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1978.

[111] M. Snir et al. Addressing Failures in Exascale Computing. *International Journal of High Performance Computing Applications*, pages 129–173, 2014.

[112] O. Sonmez, B. Grundeken, H. Mohamed, A. Iosup, and D. Epema. Scheduling Strategies for Cycle Scavenging in Multicluster Grid Systems. In *IEEE/ACM Symposium on Cluster Computing and the Grid (CCGrid)*, pages 12–19, 2009.

[113] O. Sonmez, N. Yigitbasi, S. Abrishami, A. Iosup, and D. Epema. Performance Analysis of Dynamic Workflow Scheduling in Multicluster Grids. In *ACM Symposium on High Performance Distributed Computing (HPDC)*, pages 49–60, 2010.

[114] Apache Spark. `http://spark.apache.org`.

[115] SWIM Workload. `https://github.com/SWIMProjectUCB/SWIM`.

[116] J. Tan, X. Meng, and L. Zhang. Delay tails in mapreduce scheduling. In *ACM Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE)*, pages 5–16, 2012.

[117] A. N. Tantawi and M. Ruschitzka. Performance Analysis of Checkpointing Strategies. *ACM Transactions on Computer Systems (TOCS)*, 2(2):123–144, 1984.

[118] The Distributed ASCI Supercomputer (DAS4). `http://cs.vu.nl/das4/`.

[119] The Distributed ASCI Supercomputer (DAS5). `http://cs.vu.nl/das5/`.

[120] The Economist. Data, data everywhere. `https://goo.gl/g63h2J`.

[121] The Large Synoptic Survey Telescope. `https://www.lsst.org/`.

[122] D. Tiwari, S. Gupta, and S. S. Vazhkudai. Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, pages 25–36, 2014.

[123] Transaction Processing Performance Council. TPC Benchmark-H Standard Specification. `http://www.tpc.org`.

[124] Tyrex Scheduler. `http://www.ds.ewi.tudelft.nl/ghit/projects/tyrex`.

[125] A. Uta, A.-M. Oprescu, and T. Kielmann. Towards Resource Disaggregation—Memory Scavenging for Scientific Workloads. In *IEEE Conference on Cluster Computing (CLUSTER)*, pages 100–109, 2016.

[126] P. Vagata and K. Wilfong. Scaling the Facebook data warehouse to 300 PB. `https://goo.gl/cMpqJM`.

[127] V. K. Vavilapalli et al. Apache Hadoop Yarn: Yet Another Resource Negotiator. In *ACM Symposium on Cloud Computing (SoCC)*, page 5, 2013.

[128] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The Power of Choice in Data-Aware Cluster Scheduling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316, 2014.

[129] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *ACM International Conference on Autonomic Computing (ICAC)*, pages 235–244, 2011.

[130] A. Verma, L. Cherkasova, and R. H. Campbell. Play It Again, SimMR! In *IEEE Conference on Cluster Computing (CLUSTER)*, pages 253–261, 2011.

[131] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-Scale Cluster Management at Google with Borg. In *ACM European Conference on Computer Systems (EuroSys)*, page 18, 2015.

[132] K. V. Vishwanath and N. Nagappan. Characterizing Cloud Computing Hardware Reliability. In *ACM Symposium on Cloud Computing (SoCC*, pages 193–204, 2010.

[133] K. Wang, M. Lin, F. Ciucu, A. Wierman, and C. Lin. Characterizing the Impact of the Workload on the Value of Dynamic Resizing in Data Centers. In *ACM Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE)*, pages 405–406, 2012.

[134] T. White. *Hadoop: The Definitive Guide*. Yahoo Press, 2010.

[135] A. Wierman and M. Harchol-Balter. Classifying Scheduling Policies with Respect to Unfairness in an M/GI/1. In *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 238–249, 2003.

[136] A. Wierman, M. Harchol-Balter, and T. Osogami. Nearly Insensitive Bounds on SMART Scheduling. In *ACM SIGMETRICS Performance Evaluation Review*, pages 205–216, 2005.

[137] M. Wojciechowski, M. Capotă, J. Pouwelse, and A. Iosup. BTWorld: Towards Observing the Global BitTorrent File-Sharing Network. In *Workshop on Large-Scale System and Application Performance (LSAP) in conjunction with ACM Symposium on High Performance Distributed Computing (HPDC)*, pages 581–588, 2010.

[138] R. W. Wolff. *Stochastic Modeling and the Theory of Queues*. Pearson College Division, 1989.

[139] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 383–392, 2016.

[140] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In *ACM Symposium on Cloud Computing (SoCC)*, pages 1–14, 2014.

[141] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda. TR-Spark: Transient Computing for Big Data Analytics. In *ACM Symposium on Cloud Computing (SoCC)*, pages 484–496, 2016.

[142] S. Yi, A. Andrzejak, and D. Kondo. Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2012.

[143] J. W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Communications of the ACM (CACM)*, 17(9):530–531, 1974.

[144] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *ACM European conference on Computer systems (EuroSys)*, pages 265–278, 2010.

[145] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 2–2, 2012.

[146] M. Zaharia et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM (CACM)*, 59(11):56–65, 2016.

[147] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–42, 2010.

Summary

Data analytics frameworks enable users to process large datasets while hiding the complexity of scaling out their computations on large clusters of thousands of machines. Such frameworks parallelize the computations, distribute the data, and tolerate server failures by deploying their own runtime systems and distributed filesystems on subsets of the datacenter resources. Most of the computations required by data analytics applications are conceptually straight-forward and can be performed through massive parallelization of jobs into many fine-grained tasks. Providing efficient and fault-tolerant execution of these tasks in datacenters is ever more challenging and a variety of opportunities for performance optimization still exist.

In this thesis we optimize the job performance of data analytics frameworks by addressing several fundamental challenges that arise in datacenters. The first challenge is multi-tenancy: having a large number of users may require isolating their workloads across multiple frameworks. Nevertheless, achieving performance isolation is difficult, because different frameworks may deliver very unbalanced service levels to their users. Second, users have become very demanding from these frameworks, thus expecting timely results for jobs that require only limited resources. However, even with a few long jobs that consume large fractions of the datacenter resources, short jobs may be delayed significantly. Third, improving the job performance in the face of failures is harder still, as we need to allocate extra resources to recompute work which was already done.

In order to address these challenges we design, implement, and test several scheduling policies for the evolving usage trends that are derived from the analysis of basic theoretical models. We take an experimental approach and we evaluate the performance of our policies with real-world experiments in a datacenter, using representative workloads and standard benchmarks. Furthermore, we bridge the gap between those experiments and prior theoretical work by performing large-scale simulations of scheduling policies.

In Chapter 1 we identify the main challenges we need to address in order to optimize the job performance of data analytics frameworks. We present the research questions that aim to address these challenges, the research methods employed in this thesis, and an outline of the key contributions.

In Chapter 2 we assess the benefit of a two-level scheduling architecture on the performance of MapReduce frameworks. Two-level scheduling enables the datacenter resource manager to deploy on-demand frameworks for its users. Furthermore, this design choice allows each framework to execute its workload in isolation and to elastically scale its allocation when the user demand is fluctuating. We find that with the appropriate provisioning policies, growing and shrinking single MapReduce frameworks outperforms the typical static deployment.

In Chapter 3 we design a resource sharing mechanism for dynamically balancing the resource allocation across concurrent MapReduce frameworks. We propose an abstract model of the MapReduce framework in which the allocation may grow and shrink by relaxing the data locality assumptions. To balance the resource allocation across multiple frameworks, we investigate the performance of a family of weighting policies that use feedback from three stages of a system: the input, the usage, and the output. We show that feedback on the input to the system is the best way of balancing the service levels of multiple frameworks.

In Chapter 4 we investigate the problem of achieving both fair and fast service in the face of MapReduce workloads with heterogeneous resource requirements. We design a MapReduce job scheduler that aims at isolating sets of jobs with processing requirements in different ranges. To do so, our scheduler partitions the framework resources and limits the amount of processing time jobs receive in those partitions. Furthermore, we propose a statistical method to dynamically adapt the runtime limits based on properties of the workload. We find that confining jobs to smaller partitions so that no large job monopolizes the framework significantly improves the job performance when compared with state-of-the art MapReduce job schedulers.

In Chapter 5 we explore a class of size-based scheduling policies that address the problem of large disproportions between the processing requirements of large-scale data analytics jobs and short interactive queries executed in single frameworks. These policies employ multi-queueing models and combine in all possible ways two main mechanisms: load unbalancing and/or feedback queueing. We find that multi-queueing is a very effective method to achieve both fair and fast performance and that feedback queueing performs significantly better than load unbalancing for typical MapReduce workloads.

In Chapter 6 we address the problem of improving the resilience of in-memory data analytics frameworks. We design a cluster scheduler that performs automatic checkpointing by carefully balancing the opportunity cost of persisting an intermediate result to an external storage system and the time required to recompute when the result is lost. Driven by the evidence of unpredictable intermediate data sizes and outlier tasks of jobs in production traces from Google and Facebook, we incorporate in our scheduler three policies that take into account the size of task output data, the distribution of task runtimes, or both. We find that it is worth using our policies as the checkpointing cost is relatively low

and the recovery time after failures is significantly reduced when compared with current recomputation-based and periodic checkpointing mechanisms.

Finally, in Chapter 7 we present the main conclusions of this thesis and we put into perspective several future directions. In this thesis, we show how we can optimize the job performance in data analytics frameworks by addressing the three important challenges in datacenters of multi-tenancy, workload heterogeneity, and failures. To this end, we design a scheduling system for isolating the performance of time-varying workloads across multiple MapReduce frameworks. We propose several scheduling policies that achieve both fair and fast service for workloads with variable job sizes. Finally, we design an adaptive checkpointing system for in-memory data analytics frameworks that provides fast recovery time after failures.

Samenvatting

Data-analyse *frameworks* stellen gebruikers in staat om grote datasets te verwerken terwijl de complexiteit van het opschalen van hun berekeningen naar grote clusters van duizenden machines wordt verborgen. Dergelijke *frameworks* voeren de berekeningen parallel uit, verdelen de data en tolereren serverfouten door het installeren van hun eigen runtime-systeem en gedistribueerde bestandssysteem op een gedeelte van het datacenter. De meeste berekeningen in data-analysetoepassingen zijn conceptueel eenvoudig en kunnen worden uitgevoerd door massale parallellisering van opdrachten in vele kleine taken. Het efficiënt en fouttolerant uitvoering van deze taken in datacenters is een steeds grotere uitdaging, en er bestaat nog steeds een scala aan mogelijkheden voor de optimalisatie van prestaties.

In dit proefschrift optimaliseren we de prestaties van opdrachten in data-analyse *frameworks* door het aangaan van een aantal fundamentele uitdagingen die zich in datacenters voordoen. De eerste uitdaging is *multi-tenancy*: bij een groot aantal gebruikers kan het isoleren van hun werklasten in meerdere *frameworks* vereist zijn. Niettemin is het bereiken van prestatie-isolatie lastig, omdat verschillende *frameworks* zeer onevenwichtige serviceniveaus aan hun gebruikers kunnen leveren. Ten tweede zijn gebruikers zeer veeleisend geworden ten aanzien van deze *frameworks* en verwachten snel resultaten voor opdrachten die slechts beperkte middelen nodig hebben. Maar zelfs met lechts enkele grote opdrachten die grote delen van de middelen van het datacenter gebruiken, worden kleine opdrachten aanzienlijk vertraagd. Ten derde is het verbeteren van de prestatie van de opdrachten in de aanwezigheid van fouten nog moeilijker, omdat we extra middelen moeten aanwenden om werk te herhalen dat al gedaan was.

Om deze uitdagingen aan te pakken, ontwerpen, implementeren en testen we een aantal scheduling policies voor de zich ontwikkelende trends in het gebruik van datacenters die afgeleid zijn van de analyse van fundamentele theoretische modellen. Daartoe hanteren we een experimentele aanpak en evalueren we de prestaties van onze policies met daadwerkelijke experimenten in een datacenter, met behulp van representatieve werklasten, en met standaard benchmarks. Verder overbruggen we de kloof tussen die experimenten en eerder theoretisch door grootschalige simulaties van scheduling policies uit te voeren.

In Hoofdstuk 1 stellen we de belangrijkste uitdagingen vast om de prestaties van de opdrachten in data-analyse *frameworks* te optimaliseren. We presenteren de onderzoeksvragen om deze uitdagingen aan te pakken, de onderzoeksmethoden die in dit proefschrift worden gebruikt, en een overzicht van de belangrijkste bijdragen van het proefschrift.

In Hoofdstuk 2 beoordelen we het nut van een two-level scheduling architectuur op de prestaties van MapReduce *frameworks*. Two-level scheduling stelt de datacenter resourcemanager in staat om on-demand frameworks te installeren voor zijn gebruikers. Bovendien staat deze ontwerpkeuze toe dat elk *framework* zijn eigen werklast in isolatie kan uitvoeren en zijn deel van de middelen van het datacenter elastisch kan schalen als de vraag van de gebruiker fluctueert. We constateren dat met de juiste toewijzingsregels het laten groeien en krimpen van de individuele MapReduce *frameworks* beter presteert dan de typische statische installatie.

In Hoofdstuk 3 ontwerpen we een toewijzingsmechanisme om de hulpmiddelen dynamisch te balanceren over gelijktijdig aanwezige MapReduce *frameworks*. We stellen een abstract model voor het MapReduce *framework* voor waarbinnen de toewijzing kan groeien en krimpen door een versoepeling van de datalokaliteit. Om de toewijzing van de middelen over meerdere *frameworks* in evenwicht te brengen, onderzoeken we de prestaties van een familie van gewogen policies met terugkoppeling uit de drie fasen van een systeem: de invoer, het gebruik en de uitvoer. We laten zien dat terugkoppeling van de invoer in het systeem de beste manier is om de serviceniveaus van meerdere *frameworks* in evenwicht te brengen.

In Hoofdstuk 4 onderzoeken we het probleem om zowel eerlijke als snelle dienstverlening te bereiken voor MapReduce-werklasten met heterogene behoeftes aan middelen. We ontwerpen een MapReduce job scheduler die zich richt op het isoleren van groepen van opdrachten met verwerkingseisen binnen een bepaalde grenzen. Hiertoe partitioneert deze job scheduler de middelen van het *framework* en stelt een limiet aan de hoeveelheid verwerkingstijd die opdrachten in de partities ontvangen. Verder stellen we een statistische methode voor die deze limieten dynamisch aanpast op basis van de eigenschappen van de worklast. We constateren dat het beperken van opdrachten tot kleinere partities, zodat geen grote opdracht het *framework* monopoliseert, de prestaties aanzienlijk verbetert ten opzichte van de beste bestaande MapReduce job schedulers.

In Hoofdstuk 5 onderzoeken we een klasse van scheduling policies op basis van de grootte van opdrachten die het probleem aanpakt van het grote verschil tussen de verwerkingseisen van grootschalige data-analyse-opdrachten en korte interactieve queries in een enkel *framework*. Deze policies maken gebruik van multi-wachtrijmodellen en combineren op alle mogelijke manieren twee belangrijke mechanismen: het brengen van onbalans in werklast en het herhaald achteraan sluiten bij wachtrijen. We constateren dat het gebruik van meerdere wachtrijen een zeer effectieve methode is om zowel eerlijke als snelle

prestaties te bereiken, en dat herhaaldelijk achteraan sluiten aanzienlijk beter presteert dan onbalans voor typische MapReduce-werklasten.

In Hoofdstuk 6 pakken we het probleem aan van het verbeteren van de foutbestendigheid van in-geheugen data-analyse *frameworks*. Wij ontwerpen een cluster scheduler die automatisch *checkpointing* uitvoert door het zorgvuldig balanceren van de kosten om tussentijdse resultaten extern op te slaan en de tijd die nodig is voor het herberekenen als die resultaten verloren gaan. Vanwege aanwijzingen voor de onvoorspelbare afmeting van tussentijdse gegevens en uitschieters in de taken van opdrachten in productiewerklasten van Google en Facebook, nemen we in onze scheduler drie policies op die rekening houden met de omvang van de uitvoergegevens van taken, de verdeling van de verwerkingstijden van taken, of met beide. Door middel van echte experimenten en simulaties constateren we dat het de moeite waard is om deze policies te gebruiken aangezien de kosten van *checkpointing* relatief laag zijn en de hersteltijd na fouten significant wordt gereduceerd in vergelijking met de huidige mechanismen op basis van herberekening en periodieke *checkpointing*.

Tenslotte presenteren we in Hoofdstuk 7 de belangrijkste conclusies van dit proefschrift en plaatsen we verschillende toekomstige richtingen in perspectief. In dit proefschrift laten we zien hoe we de prestaties van opdrachten in data-analyse *frameworks* kunnen optimaliseren door het aanpakken van de drie belangrijke uitdagingen in datacenters: *multi-tenancy*, de heterogeniteit van werklasten, en het voorkomen fouten. Hiertoe ontwerpen we een scheduling systeem voor het isoleren van de prestaties van in de tijd variërende werklastenvan meerdere MapReduc *frameworks*. Wij stellen verschillende scheduling policies voor die zowel eerlijke als snelle dienstverlening bereiken voor werklasten met taken van variabele grootte. Ten slotte ontwerpen we een adaptief *checkpointing* systeem voor in-geheugen data-analyse *frameworks* dat een snel herstel na fouten biedt.

Curriculum Vitæ

Bogdan Ionuț Ghiț was born on the 3rd of September 1986 in Bucharest, Romania. He received a BSc degree in Computer Science from Politehnica University of Bucharest in 2009, and an MSc degree in Parallel and Distributed Computing Systems from Vrije University of Amsterdam in 2011.

Bogdan pursued his PhD in Computer Science at Delft University of Technology as a member of the Distributed Systems group. His research interests are in the area of computer systems and algorithms for large-scale data-intensive computing. Bogdan has built a solid background in mathematics that helps him design new scheduling policies for data analytics frameworks. The results of his research were published in top conferences such as ACM SIGMETRICS, ACM HPDC, IEEE MASCOTS, and IFIP Performance. His work on scheduling MapReduce frameworks received the best paper award at the MTAGS workshop in 2012. He won the IEEE SCALE Challenge at the IEEE/ACM CCGrid conference with the TU Delft team in 2014. He was selected to participate at the 3rd Heidelberg Laureate Forum in 2015.

During the summer of 2016, Bogdan was a research intern at IBM T. J. Watson Research Center in Yorktown Heights, New York, where he worked on the design and implementation of a resource allocation mechanism for data analytics applications in cloud spot markets. The results of this work led to one U.S. patent application.

List of Publications

1. Bogdan Ghiț and Dick Epema, "Better Safe than Sorry: Grappling with Failures of In-Memory Data Analytics Frameworks", *ACM International Symposium on High Performance Parallel and Distributed Computing (HPDC)*, 2017.

2. Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro V. Papadopoulos, Bogdan Ghiț, Dick Epema, and Alexandru Iosup, "An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows", *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2016.

3. Bogdan Ghiț and Dick Epema, "Tyrex: Size-based Resource Allocation in MapReduce Frameworks", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

4. Alexey Ilyushkin, Bogdan Ghiț, Dick Epema, "Scheduling Workloads of Workflows with Unknown Task Runtimes", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

5. Bogdan Ghiț and Dick Epema, "Reducing Job Slowdown Variability for Data-Intensive Workloads", *IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2015.

6. Lipu Fei, Bogdan Ghiț, Alexandru Iosup, and Dick Epema, "KOALA-C: A Task Allocator for Integrated Multicluster and Multicloud Environments", *IEEE International Conference on Cluster Computing (CLUSTER)*, 2014.

7. Bogdan Ghiț, Nezih Yiğitbaşı, Alexandru Iosup, and Dick Epema, "Balanced Resource Allocations across Multiple Dynamic MapReduce Clusters", *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

8. Bogdan Ghiț, Mihai Capotă, Tim Hegeman, Jan Hidders, Dick Epema, Alexandru Iosup, "V for vicissitude: The challenge of scaling complex big data workflows", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) SCALE Challenge*, 2014. IEEE SCALE Challenge winner.

9. Tim Hegeman, Bogdan Ghiț, Mihai Capotă, Jan Hidders, Dick Epema, Alexandru Iosup, "The BTWorld Use Case for Big Data Analytics: Description, MapReduce Logical Workflow, and Empirical Evaluation", *IEEE International Conference on Big Data*, 2013.

10. Bogdan Ghiț and Dick Epema, "Dynamic Resource Provisioning for Concurrent MapReduce Frameworks", *IFIP Performance (Extended Abstract)*, 2013.

11. Bogdan Ghiț, Alexandru Iosup, and Dick Epema, "Towards an Optimized Big Data Processing System", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) Doctoral Symposium*, 2013.

12. Bogdan Ghiț, Nezih Yiğitbaşı, and Dick Epema, "Resource Management for Dynamic MapReduce Clusters in Multicluster Systems", *Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS) in conjunction with International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012. Best Paper Award.