HAL
open science

Size-based disciplines for job scheduling in data-intensive scalable computing systems

Mario Pastorelli

To cite this version:

Mario Pastorelli. Size-based disciplines for job scheduling in data-intensive scalable computing systems. Distributed, Parallel, and Cluster Computing [cs.DC]. Télécom ParisTech, 2014. English. NNT : 2014ENST0048 . tel-01415094

HAL Id: tel-01415094

https://pastel.hal.science/tel-01415094

Submitted on 12 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2014-ENST-0048

EDITE - ED 130

Doctorat ParisTech
T H È S E
pour obtenir le grade de docteur délivré par

TELECOM ParisTech
Spécialité « INFORMATIQUE et RESEAUX »

présentée et soutenue publiquement par

Mario PASTORELLI

le 18 julliet 2014

Size-Based Disciplines for Job Scheduling in Data-Intensive Scalable Computing Systems

Directeur de thèse: **Pietro MICHIARDI**

T H È S E

Jury

M. Ernst BIERSACK, Département Réseaux et Sécurité, EURECOM	Président
M. Guillaume URVOY-KELLER, Université Nice Sophia Antipolis (UNS)	Rapporteur
M. Giovanni CHIOLA, University of Genoa (DIBRIS)	Rapporteur
M. Patrick BROWN, Orange Labs, Sophia Antipolis	Examinateur

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

Abstract

The past decade have seen the rise of *data-intensive scalable computing* (DISC) systems, such as Hadoop, and the consequent demand for scheduling policies to manage their resources, so that they can provide quick response times as well as fairness. Schedulers for DISC systems are usually focused on the fairness, without optimizing the response times. The best practices to overcome this problem include a manual and ad-hoc control of the scheduling policy, which is error-prone and difficult to adapt to changes.
In this thesis we focus on size-based scheduling for DISC systems. The main contribution of this work is the *Hadoop Fair Sojourn Protocol* (HFSP) scheduler, a size-based preemptive scheduler with aging; it provides fairness and achieves reduced response times thanks to its size-based nature. In DISC systems, job sizes are not known a-priori: therefore, HFSP includes a job size estimation module, which computes approximated job sizes and refines these estimations as jobs progress.
We show that the impact of estimation errors on the size-based policies is not significant, under conditions which are verified in a system such as Hadoop. Because of this, and by virtue of being designed around the idea of working with estimated sizes, HFSP is largely tolerant to job size estimation errors. Our experimental results show that, in a real Hadoop deployment and with realistic workloads, HFSP performs better than the built-in scheduling policies, achieving both fairness and small mean response time. Moreover, HFSP maintains its good performance even when the cluster is heavily loaded, by focusing the resources to few selected jobs with the smallest size.
HFSP is a preemptive policy: preemption in a DISC system can be implemented with different techniques. Approaches currently available in Hadoop have shortcomings that impact on the system performance. Therefore, we have implemented a new preemption technique, called *suspension*, that exploits the operating system primitives to implement preemption in a way that guarantees low latency without penalizing low-priority jobs.

Acknowledgements

In the past years, many great people have been part of my life and inspired and helped my work. Without their support this thesis would not have been possible. I owe all of them a debt of gratitude and I would like to use the following lines to express my thankfulness.
My first thanks go to my advisor, Prof. Pietro Michiardi, for his patience, support and guidance. He has been a wise and careful mentor and helped me with his immense enthusiasm. However, reducing the role of Prof. Michiardi to a mentor would not be fair: indeed, in the past years he has been a good friend and helped me becoming a better person both at the professional and the human level. I hope to work again with him in the future and continue to learn from him as I did during my period at Eurecom as a PhD student.
Moreover, he made me part of a team of fantastic people where I have grown as a researcher and engineer. I also want thank the great BigFoot team: thank you Xiaolan Sha, Duy-Hung Phan, Daniele Venzano and Nguyen Duc Trung for the amazing experience of the past years. I have been part of something bigger and more important then myself and this encouraged me to improve myself and to try to help the group. Special thanks to Matteo Dell'Amico, who gave me the possibility to do an internship at Eurecom and then start my PhD. None of this would be possible without him. His determination, curiosity and knowledge always inspired me.
I then would like to thank Prof. Damiano Carra for being my second mentor and for helping me defining problems and find possible research directions out of them. In particular, he made me a better researcher by let me realize that being a researcher means primarily to be able to understand the problem you are facing and how to approach it.
My sincere thank to my friend and former colleague Antonio Barbuzzi for his invaluable friendship and support. Friends like him are rare material in our world.
Thanks to my friends Giuseppe Reina and Marco Milanesio for their friendship and for helping me find diamonds even in the adversities.
I would like to express my gratitude to my parents, Marina Beglia and Germano Pastorelli, because they made an important part of what I am today. Their advices, encouragements and love made me working harder and better.
Another thank goes to my brother, Paolo, my very first friend. I hope to be as inspiring to him as he has been to me.

Finally, I would like to express my gratitude to my fiancée Veronica Girolimetti, for her immense support and love. As for any other part of my life, I would like to dedicate this thesis to her, the centre of my life.

Table of Contents

Table of Contents 7

1 Introduction 15
1.1 Motivations . . . 16
1.2 Challenges . . . 17
1.3 Contributions and Thesis Organization . . . 18
1.3.1 Size-Based Scheduling with Estimated Sizes . . . 19
1.3.2 The Hadoop Fair Sojourn Protocol . . . 20
1.3.3 OS-Assisted Task Preemption . . . 21
1.3.4 Conclusion and Perspectives . . . 21

2 Background and Related Work 23
2.1 Background . . . 23
2.1.1 Performance Metrics . . . 23
2.1.2 Scheduling Policies . . . 24
2.1.3 MapReduce . . . 27
2.2 Related Work . . . 32
2.2.1 Size-Based Scheduling Policies with Inexact Job Sizes . . . 32
2.2.2 Scheduling for DISC Systems . . . 33
2.2.3 Job Size Estimation in MapReduce . . . 35
2.2.4 Preemption for MapReduce systems . . . 36

3 Size-based scheduling with estimated sizes 37
3.1 Scheduling Based on Estimated Sizes . . . 37
3.1.1 SRPTE and FSPE . . . 38
3.1.2 FSPE+PS . . . 39
3.2 Evaluation Methodology . . . 41
3.2.1 Scheduling Policies Under Evaluation . . . 41
3.2.2 Parameter Settings . . . 42
3.2.3 Simulator Implementation Details . . . 43
3.3 Experimental Results . . . 44
3.3.1 Synthetic Workloads . . . 44

3.3.2 Real Workloads . . . 50
3.4 Summary . . . 52

4 The Hadoop Fair Sojourn Protocol 55
4.1 Jobs . . . 55
4.2 The Estimation Module . . . 56
4.2.1 Tiny Jobs . . . 56
4.2.2 Initial Size . . . 56
4.2.3 Final Size . . . 57
4.3 The Aging Module . . . 58
4.3.1 Virtual Cluster . . . 58
4.3.2 Aging Speed . . . 59
4.3.3 Estimated Size Update . . . 59
4.3.4 Failures . . . 60
4.3.5 Manual Priority and QoS . . . 60
4.4 The Scheduling Policy . . . 60
4.4.1 Job Submission . . . 60
4.4.2 Priority To The Training Stage . . . 60
4.4.3 Virtual Time Update . . . 62
4.4.4 Data locality . . . 62
4.4.5 Scheduling Algorithm . . . 62
4.5 Impact of Estimation Errors . . . 63
4.6 Task Preemption . . . 65
4.6.1 Task Selection . . . 65
4.6.2 When Preemption Occurs . . . 66
4.7 Summary . . . 66

5 System Evaluation 67
5.1 The BigFoot Platform . . . 67
5.2 Experimental Setup . . . 69
5.3 Macro Benchmarks . . . 71
5.3.1 Response Time . . . 71
5.3.2 Slowdown . . . 73
5.4 Micro Benchmarks . . . 75
5.5 Summary . . . 80

6 OS-Assisted Task Preemption 81
6.1 OS-assisted Task Preemption . . . 81
6.1.1 Suspension and Paging in the OS . . . 81
6.1.2 Thrashing . . . 82
6.1.3 Implementation Details . . . 83
6.2 Experimental Evaluation . . . 84

6.2.1 Experimental Setup 84
6.2.2 Performance Metrics 85
6.2.3 Results 85
6.2.4 Impact of Memory Footprint. 87
6.3 Discussion 88
6.3.1 Scheduling and Eviction Strategies 88
6.3.2 Implications on Task Implementation 89
6.4 Summary 90

7 Conclusion **91**
7.1 Future Work 92

A French Summary **105**
A.1 Motivations 109
A.2 Challenges 111
A.3 Contributions et Organisation de thèse 113
A.3.1 Planification de Taille-base avec taille estimée 113
A.3.2 The Hadoop Fair Sojourn Protocol 115
A.3.3 OS-Assisted Task Preemption 117
A.3.4 Conclusion and Perspectives 118
A.4 Travaux Futurs 119

List of Figures

2.1 FIFO and PS examples . . . 25
2.2 LAS and SRPT examples . . . 26
2.3 FSP for two jobs . . . 27
2.4 MapReduce Job Workflow . . . 30

3.1 Examples for job under- and over-estimation. . . . 38
3.2 Mean sojourn time against PS. . . . 45
3.3 Impact of shape. . . . 45
3.4 Impact of error on heavy-tailed workloads, sorted by growing skew. . . . 47
3.5 Pareto job size distributions, sorted by growing skew. . . . 48
3.6 Impact of load and timeshape. . . . 48
3.7 Mean conditional slowdown. . . . 49
3.8 Per-job slowdown: full CDF (top) and zoom on the 10% more critical cases (bottom). . . . 49
3.9 CCDF for the real workloads. . . . 50
3.10 MST of the Facebook workload. . . . 51
3.11 MST of the IRCache workload. . . . 52

4.1 Example of continuous updates of job sizes. . . . 58
4.2 An example of virtual time and job size progress in HFSP. . . . 59
4.3 Job's lifetime in HFSP . . . 61
4.4 Illustrative examples of the impact of estimation errors on HFSP. . . . 64

5.1 Aggregate *mean* response times for all workloads. . . . 72
5.2 ECDF of the response times for the PROD workload. . . . 72
5.3 ECDF of the response times for the DEV workload, excluding jobs from bin 1. . . . 73
5.4 ECDF of the response times for the TEST workload, grouped per bin. . . . 73
5.5 ECDF of the per-job slowdown, for all workloads. . . . 74
5.6 Time-series of the cluster load, for an individual experiment with the PROD workload. . . . 75
5.7 ECDF of task run times, for the Map and Reduce phases of all jobs across all workloads for HFSP. . . . 76

5.8 ECDF of the normalized task run time, for the MAP and REDUCE phases of all jobs across all workloads for HFSP. 76
5.9 ECDF of estimation errors, for the MAP and REDUCE phases of all jobs across all workloads for HFSP. 76
5.10 Most important percentiles and outliers of estimation errors, grouped by bin, for the phases of jobs across all workloads for HFSP. 77
5.11 ECDF of job response times, for the TEST workload. Comparison between the KILL and WAIT preemption primitives. 79
5.12 ECDF of the per-job slowdown, for the TEST workload. Comparison between the KILL and WAIT preemption primitives. 79

6.1 Task execution schedules. 84
6.2 Baseline experiments: a comparison of the three preemption primitives with light-weight tasks. 86
6.3 Worst-case experiments: a comparison of the three preemption primitives with memory-hungry tasks. 87
6.4 Overheads when varying memory usage. 88

List of Tables

3.1 Simulation parameters. 42

5.1 Summary of the workloads used in our experiments. 71

Chapter 1

Introduction

Scheduling is one of the most studied problems in many fields, and in particular in computer science. From an abstract perspective, the scheduling problem can be seen as follows: given a set of resources and a set of jobs, where a job needs a subset of the resources to progress, how can they be assigned to the jobs such that all the jobs will eventually complete while optimizing some metric? Instances of this problem are common for operating systems, databases, networking, cloud computing systems and many others. Scheduling plays a fundamental role since the performance of a system greatly varies based on the adopted policy. The impact of a scheduling policy is usually measured through a performance index: the most commonly used metrics are, for instance, the mean *response time* (also referred to as *sojourn time*), which is the time a job stayed in the system until it is completely served, or the mean *waiting time*, i.e. the time spent in the waiting line before receiving service, or the *fairness*, which indicates how fairly a job is treated. Some metrics are sometimes difficult to optimize at the same time, therefore a scheduling policy may need to face a trade-off. For instance, the *Processor Sharing* (PS) scheduling policy, which equally divides the resources among the jobs currently in the system, provides fairness, but it brings to high mean response times.

Scheduling policies can be classified in four families, based on whether a policy is *size-based* or *blind to size*, or if a policy is *preemptive* or *non-preemptive* – we consider work-conserving policies, meaning that, if there are jobs to be served, the server is always busy working on a job, and no work that is done is lost. A size-based scheduling policy is a policy that takes scheduling decisions by considering the size of a job. For instance, the *Shortest Remaining Processing Time* (SRPT) policy is size-based because it schedules the job with the smallest remaining processing time first. The *First-In-First-Out* (FIFO) scheduler is, instead, blind to the size since it schedules jobs based on their arrival time. A preemptive scheduling policy is a policy that can suspend a job in service before it completes, *i.e.* it can remove resources previously granted to a job. Preemptive schedulers are known in literature to offer better performance, since a large job in service may block the system, while preemption may suspend the execution of such a large job in

case of arrival of a small job. For example, PS is a preemptive scheduler while FIFO is not.
Among the four families, the scheduling policies that are size-based *and* preemptive are known to provide the best performance: the SRPT policy, for instance, is optimal in terms of mean response time [78], while the *Fair Sojourn Protocol* (FSP) [75] guarantees fairness while providing a mean response time similar to SRPT.

In this thesis we focus on scheduling policies for *Data-Intensive Scalable Computing* (DISC) systems. Such systems are composed by hundreds or thousands of servers, and jobs can use either all the resources or only a fraction of them. Given the widespread adoption of these systems, and their growing relevance, it is interesting to investigate the impact of the scheduling disciplines on the performances of such distributed and parallel systems.

1.1 Motivations

Size-based scheduling policies are known in literature for their superior performance compared to policies that are blind to size. Despite this, size-based policies have little adoption in real systems because of the problems that arise when such policies are converted from theoretical policies to practical schedulers. As a result, policies that are blind to size are the de-facto standard for DISC systems.
Schedulers for DISC systems are complex: indeed, scheduling multiple jobs, where each job is composed by tasks that can be run in parallel, in a distributed environment is a challenging task. Current DISC systems, such as Hadoop [44], Spark [86] and Naiad [122], use schedulers that are based on two different basic strategies: PS and FIFO. In production systems, the system administrators usually deploy some variations of such policies: for instance, it may introduce different priority classes that may favor interactive jobs with respect to batch processing jobs, and use FIFO within each priority class. These approaches require that the system administrator *configures manually* the scheduler (*e.g.*, how to handle different priorities) based on the workload and the system setting. This process requires a vast knowledge of both the workload and the system and tends to be error prone, difficult to both validate and debug and cannot be adapted easily to workload and system changes. Moreover, in a DISC system where resources are distributed among many machines, the manual configuration is even more critical and involves many parameters to be fine tuned and regularly checked.

We believe that, due to the lack of research work on how to use size-based schedulers in realistic environments, current systems are truly missing the opportunity of using better

scheduling policies to avoid drawbacks of blind to size policies and to drastically reduce the problems related to manual configurations.

As we are going to show in this thesis, not only size-based scheduling policies for DISC systems are a viable solution, but they also perform very well in many production level scenarios. The main motivation of this thesis is to show that size-based scheduling is to show this by designing and implementing the *Hadoop Fair Sojourn Protocol* (HFSP), a size-based scheduling policy for a real and complex system such as Hadoop, and by providing an extensive study of the scheduler architecture and its performance necessary to understand how and why it works.

Thesis Statement: *A preemptive size-based scheduling policy for DISC systems can be both efficient and fair, and it can improve the performance with respect to the current state-of-the-art scheduling policies.*

1.2 Challenges

Implementing a size-based scheduling policy for DISC systems raises many challenges; in the following, we summarize the most important ones.

Job sizes are unknown: Size-based schedulers, despite their superior performance, are very rarely deployed in practice. A key reason is that, in real systems, job size is almost never known *a priori*. When designing a size-based scheduler, therefore, the first problem is how to obtain a job size so that it can be used by the scheduler to sort jobs. We assume that the information about the job size is not given (*e.g.*, provided by the user): the scheduler must find a way to *determine the size of a job once the job has arrived, i.e., while the job is in the queue and other jobs are running*. Since the measurement of the job size may be imprecise, we need to face another problem: the estimation errors.

Estimation errors: In a real system it is not possible to determine the exact size of a job. Instead, it is often possible to provide *estimations* of job sizes. This means that the scheduler needs to *cope with erroneous sizes*.
Perhaps surprisingly, even considering the simple case of a single server, very few works in the literature tackled the problem of size-based scheduling with inaccurate job size information. Moreover, these few works give somewhat pessimistic results, suggesting that size-based scheduling is effective only when the error of the estimated size is small. Nevertheless, such studies cover a limited family of workloads, and their answers are not exhaustive.
When designing a size-based scheduler, therefore, we need to understand the impact of the size estimate errors on the overall performance, *i.e.*, if the scheduler is able to make

sufficiently good decisions despite imprecise information on the job size. To the best of our knowledge, no study tackled the design of size-based scheduling techniques that are *explicitly designed with the goal of coping with errors* in job size information, not even in the single server case. For this reason, we first need to understand the impact of errors in the single server case, and then extend the lesson learned from this basic configuration to the more complex DISC systems.

Job preemption: the scheduling policies that have the best performance in term of fairness and mean sojourn time are preemptive. Indeed, many works show that a preemptive scheduling policy is often better than its not preemptive counter-part. This can be intuitively understood considering the case when a new small job arrives while the system is serving a large job: without preemption, the small job needs to wait until the large job is completely served.
In many real systems, such as Hadoop, job preemption is often absent or partially implemented. When there is a preemption primitive implemented, it usually has drawbacks and limitations preventing it from being effectively used because its drawbacks are worse then its advantages.
When designing a size-based scheduler, therefore, we need to understand how to provide efficient primitives for preemption, and how to implement such primitives in current system in a seamless way.

Jobs starvation: Size-based scheduling policies like SRPT can cause starvation when a job is kept from getting the resources by continuously arriving smaller jobs. Some works addressed this problem in the case of single server queue by proposing policies that restore fairness; nevertheless, such policies have never been implemented in real systems (not even single server systems).

1.3 Contributions and Thesis Organization

The main contribution of this thesis is the *Hadoop Fair Sojourn Protocol*, a size-based scheduling policy for Hadoop that is both fair and efficient. Because HFSP relies on job size estimation and because size estimation is so important for size-based scheduling in general, the first Chapter (Chapter 3) is dedicated to the study of the impact of estimated sizes on existing scheduling policies in some simple scenarios. With this work as background, we then proceed defining HFSP (Chapter 4) and experimentally evaluating it (Chapter 5). In the HFSP evaluation Chapter we also provide a study of HFSP with the preemption primitives currently available in Hadoop, and we suggest that a new preemption primitive should be implemented to overcome the Hadoop preemption primitive problems. The next Chapter (Chapter 6) is then dedicated to a new preemptive primitive that solves such problems.

The rest of this section is dedicated to an overview of our contributions.

1.3.1 Size-Based Scheduling with Estimated Sizes

Before starting with the design of the HFSP protocol, we need to understand if the imprecise information about sizes may have a significant impact on the scheduling choices. Since this problem has not been well studied in the literature, we analyze it starting from a basic configuration: the single server case. Even if the single server case may seem simple, it provides a lot of insights that are useful to understand the behavior of the system. Chapter 3 is therefore dedicated to the study of scheduling policies that use estimated sizes. The insights gained in this Chapter will be used to drive the design of a more complex scheduler for DISC systems such as HFSP.
In the first part of the Chapter, we study the scheduling problem in presence of estimation errors and then provide an overview of the current state-of-the-art for both size-based and blind to size schedulers performance.
We then describe the impact of estimation errors on scheduling policies by defining SRPTE and FSPE, two variants of well known size-based scheduling policies for single server queues that work with estimated sizes. There are two kinds of errors that can be done when a job size is estimated: it can be *overestimated* or *underestimated* if its estimated size is, respectively, bigger or smaller than its real size. These two kinds of errors have two completely different impacts on size-based policies and require different strategies to deal with them. In particular, while jobs with overestimated size delay only themselves, jobs with underestimated size can potentially delay all the jobs in the queue. Between the two, the second kind of errors has the highest impact on the scheduling policy and can lead to very poor performance.
The next part of Chapter is dedicated to define *FSPE+PS*, a size-based scheduling policy for single server queue that is tolerant to estimation errors. Our simulative evaluation, which considers both variation in the workload composition and in the estimation error, shows two main results:

- despite their problems with jobs with underestimated sizes, both SRPTE and FSPE have excellent performance in many cases and they are good choices for an implementation of a real scheduler.
- FSPE+PS is even superior and provides better performance even in extreme cases of both workload composition and error.

The result of this Chapter is that size-based scheduling, even in presence of imprecise information, is a feasible policy that can be implemented in real systems.

1.3.2 The Hadoop Fair Sojourn Protocol

In Hadoop, a job is composed by tasks, and tasks can be run in parallel. As we are going to show in Chapter 4, most of the times tasks of the same job have very similar sizes. A few tasks can be run in the system and then, based on their performance, it is possible to deduce the estimated size of a job with small error.
We implemented the Hadoop Fair Sojourn Protocol scheduler, a scheduling policy for DISC systems based on the FSPE policy defined for a single server queue. While the main contribution of this Chapter is a fully fledged scheduler for a distributed system, the architectural choices that lead to this scheduler are not less important. Indeed, the adaption of scheduling policies defined for the single server case (such as FSPE) to a real system like Hadoop raises many challenged that must be addressed.
HFSP consists of two main components, which are described in detail in Chapter 4: the *estimation module* and the *aging module*.
The estimation module is the component that estimates job sizes. It first provides a very rough estimation when a job is submitted and then upgrades the size to a more precise one based on the performance of a subset of the tasks. This strategy of estimating sizes is designed around DISC systems, in which a job is composed by tasks, and leads to very good estimations in the end.
The aging module is the component that avoids job starvation by applying what is called "aging" to a job. HFSP doesn't take decisions only based on the size but also based on how much time the job stays in the queue. In this way, even a relatively big job will eventually obtain resources, which solves the starvation problem. To age jobs, HFSP simulates *Max-Min Processor Sharing* in a virtual cluster with the same characteristics as the real one.
HFSP is a preemptive scheduler. Hadoop provides two possible options that can be used for job preemption. The first one is to stop running the tasks belonging to the job to be preempted by killing them – we call this strategy Kill. The second option is to wait for each task to complete and, once the resources become available on a task-by task basis, the system assignes such resources to the preempting job – we call this strategy Wait[1]. The final part of Chapter 4 is dedicated to the analysis of the advantages and drawbacks of these two approaches.

The following Chapter, Chapter 5, is dedicated to an experimental evaluation of HFSP. Evaluating the real implementation of a scheduling policy for a system such as Hadoop is a very complex task. We decided to validate it in an experimental way to be able to compare HFSP to the most used schedulers for Hadoop, which are the Fair – an implementa-

[1]Even if this approach may not seem a job preemption, we should consider that a job may require more tasks than the system can provide, therefore, when a task completes, the system reassigns the resources to the running job so that it can proceed; with the Wait primitive, instead, the system assigns the resources to the preempting job.

tion of the PS discipline – and the FIFO schedulers. The main reason is that we want to put emphasis on the fact that HFSP is a real scheduler that outperforms industrial-ready schedulers. Our results show that HFSP is always better than both schedulers for the two metrics observed – fairness and mean response (or sojourn) time. We also observed that the sequential nature of HFSP leads to smaller cluster load and better "absorption" of bursts of job submissions. The cluster load and burst tolerance make HFSP capable to deal with smaller cluster than the counterparts for the same workload, leading to less cluster costs and better resource utilization.
The next part of the evaluation Chapter is dedicated to estimation errors. The results confirm that errors done by estimating the job size with our estimation module is small enough to be able to justify the use of size based schedulers with imperfect job size information.
The final part of this Chapter shows an experimental evaluation of HFSP with Kill preemption mechanism enabled – the default behaviour of HFSP is to use Wait – with very interesting results. Compared to Wait, Kill is a good way to achieve better fairness for all jobs and smaller response times for small and medium jobs, but it has an impact on the largest jobs.

1.3.3 OS-Assisted Task Preemption

Chapter 6 is dedicated to the design of a new task preemption primitive alternative to Wait and Kill (the primitives available for Hadoop), which we named *task suspension*. Our solution works at the Operating Systems (OS) level: in fact, tasks are just OS processes, therefore we can control running jobs by using the primitives to Suspend and Resume. This approach is completely transparent to users and exploits the system functionalities. Our preemption mechanism can be used in production even with stateful tasks, *i.e.*, tasks that have a state and need that state to continue the computation.
After the definition of our mechanism, we compare the results obtained by using our Suspend, the Kill and the Wait primitives. Our results show that in almost every case, except for corner cases when the job is just started or has almost finished, our Suspend primitive always performs better than the other two primitives.

1.3.4 Conclusion and Perspectives

The last Chapter of the thesis summarizes the main results we obtained. The design of a scheduler for a complex system such as Hadoop raises many issues, and we address many of them in our work. Nevertheless, the system itself is evolving, and its widespread adoption introduces different functionalities that are making the system more and more complex. In the last part of the Chapter we provide a set of possible future directions that consider such evolving complex system.

Chapter 2

Background and Related Work

The first part of this Chapter presents the background on scheduling, starting from scheduling policies for single server systems and then delving into the scheduling for data-intensive scalable computing systems. The second part of the Chapter is dedicated to the related work.

2.1 Background

In this section we provide a background on performance metrics for schedulers (Section 2.1.1), on scheduling policies theory (Section 2.1.2) and MapReduce (Section 2.1.3), which are the base ground of our work. The literature on both fields, in particular for scheduling policies, is vast. We will focus on the work that is used as a foundation for what is done in this thesis. In particular we will discuss mainly the scheduling policies for *batch* jobs, which are mostly processing jobs that, once they are started, do not require user interaction to progress.

2.1.1 Performance Metrics

In this thesis we are going to focus on two metrics: the *mean response time* (or *sojourn time*) and *fairness*. The response time is the time that passes between the moment a job is submitted and when it completes; such a metric is widely used in the scheduling literature.
The definition of fairness is more elusive: in his survey on the topic, Wierman affirms that *"fairness is an amorphous concept that is nearly impossible to define in a universal way"* [79]. A common approach is to consider *slowdown*, *i.e.* the ratio between a job's sojourn time and its size, according to the intuition that the waiting time for a job should be somewhat proportional to its size. In this work we focus on the per-job slowdown, which allows us to analyze if a job is treated unfairly [81], *i.e.*, it has a very large slowdown compared to the other jobs.

2.1.2 Scheduling Policies

Scheduling policies have been the subject of many studies in the past decades, from both theoretical and experimental perspectives. In this Section we describe the main scheduling policies available in literature as background for both Chapter 3 and Chapter 4. The scheduling policies presented here consider a simple setting with a single server; it is also assumed that jobs can be preempted at any time without *context-switching* costs, *e.g.* cost to switch from one job to another, and the single server is a continuous resource that can be split arbitrarily. Because our work is mainly focused on a real implementation of scheduling policies, we are going to relax those assumptions in Chapter 4.

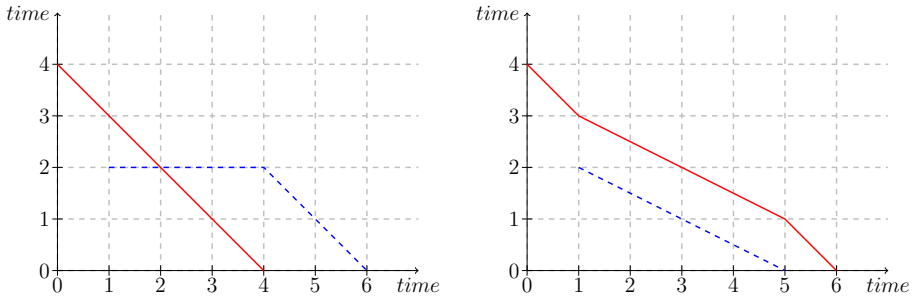
First-In-First-Out

The *First In-First-Out* scheduler (FIFO), also known in literature as *First-Come-First-Served* (FCFS), is a scheduling policy that always schedules the job that has arrived first. The performance of FIFO is highly affected by the workload composition, *i.e.* the jobs that must be scheduled. If the workload is mainly composed by jobs with similar sizes, then FIFO is a good choice. If the workload has jobs with different sizes, then smaller jobs are likely to queue until the larger jobs submitted before them complete. For instance, in a batch system, jobs may have sizes with different order of magnitudes (*e.g.*, from seconds to hours [26]). If a small job (few seconds to run it) comes after a large job (hours to run it), then the former will have to wait for the latter to finish. Despite the problems of FIFO, it is still used a lot in production because of its simple implementation. Figure 2.1a shows an example of FIFO scheduling: given one job j_1 with arrival time 0 and size 4[1] and another job j_2 with arrival time 1 and size 2, FIFO will first schedule j_1 and then, when j_1 completes, j_2. The Figure shows the remaining size of a job in the y-axis and the time progression in the x-axis.
FIFO strengths: when jobs sizes are very similar, FIFO is fair, meaning that all jobs finish before or similarly to PS, and efficient in term of mean response time.
FIFO weaknesses: when jobs sizes are very different among them, FIFO is not fair and performs poorly in term of mean response time.

Processor Sharing

Processor Sharing (PS) is a scheduling policy that splits the resources equally among all the jobs in the queue. In PS, all the jobs in queue receive the same fraction of resources (*e.g.* CPU, bandwidth...): if there are n jobs, then each job receives $1/n$ of the server capacity. PS achieves fairness but at the cost of increasing the processing times of the jobs. Indeed, if a job with size s receives on average $1/n$th of the resources, then it will take $s \cdot n$ times to complete.
Figure 2.1b shows an example of PS: given two jobs with arrival time respectively $t_1 = 0$

[1]Job sizes are normalized with respect to the service rate.



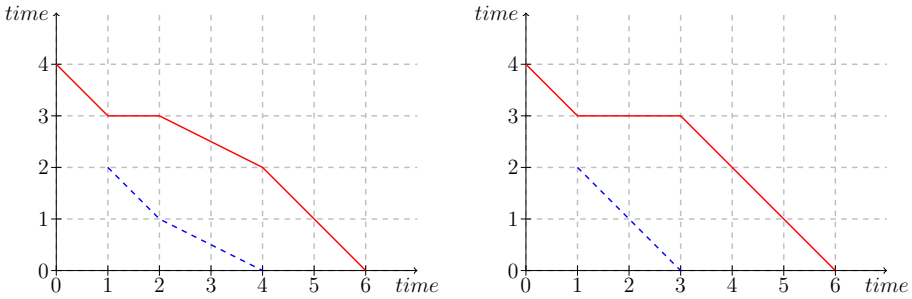
(a) FIFO scheduling of two jobs (b) PS scheduling of two jobs

Figure 2.1: FIFO and PS examples

and $t_2 = 1$, the scheduler will schedule half of the resources to each one and wait them to complete.
Because in the real system the assumption of continuous resources is relaxed and thus resources are discrete, PS is often implemented by using a *Round Robin* strategy: each job receives one "slot" of the resources until all jobs have their resources or all the resources are occupied. An example of Round Robin scheduling implementation is the *Max-Min Fair Scheduler*, which uses round robin to assign resources to each job starting from the one that needs fewer resources. We are going to use an implementation of the Max-Min Fair Scheduler for HFSP in Chapter 4.
PS strengths: fairness among jobs despite workloads composition.
PS weaknesses: jobs response times are increased based on how many jobs are in queue.

Least Attained Service

Least Attained Service (LAS) is a scheduling policy that prioritizes the job which used the system less. When two jobs have the same server usage time, LAS does Processor Sharing among them. LAS works particularly well for skewed workloads [74] because long jobs are preempted by other jobs and the more a job used the server the lower priority it will have.
Figure 2.2a shows an example of LAS scheduling: given one job j_1 with arrival time 0 and size 4 and another job j_2 with arrival time 1 and size 2, LAS will schedule j_1 between its submission and the submission of j_2, then schedule the server to j_2 and when j_2 reaches the same amount of service time of j_1 it will proceed using a PS policy.
LAS strengths: when jobs sizes are very different among them, LAS is fair, meaning that all jobs finish before or similarly to PS, and efficient in term of mean response time.
LAS weaknesses: when jobs sizes are very similar, LAS is not fair and performs poorly in term of mean response time.



(a) LAS scheduling for two jobs (b) SRPT scheduling for two jobs

Figure 2.2: LAS and SRPT examples

Shortest Remaining Processing Time

Shortest Remaining Processing Time (SRPT) is a size-based scheduling that serves the job with the smallest remaining size first. SRPT guarantees the minimum mean response time [78].
Figure 2.2b shows an example of SRPT scheduling: given a job j_1 with arrival time 0 and size 4 and another job j_2 with arrival time 1 and size 2, SRPT will schedule j_1 until j_2 arrives, then will give priority to j_2 because its size 2 is smaller than the current remaining size of j_1, which is 3. When j_2 completes, SRPT grants the server to j_1 that completes at 6. The optimal mean response time is $\frac{6+2}{2} = 4$.
While SRPT is well known in scheduling theory to be the most efficient scheduler in term of mean response time, it has two main problems that prevent its utilization inside a cluster. First of all, jobs can starve. In particular, a job will never have granted the server if the jobs queue always has a smaller job inside.
SRPT strengths: optimal mean response times.
SRPT weaknesses: jobs can starve.

Fair Sojourn Protocol

The *Fair Sojourn Protocol* (FSP) scheduling [38] is a size-based scheduling policy based on SRPT and PS that is both fair and leads to small mean response times. The key idea of FSP is to observe the jobs completion times in PS, through a simulation, and then schedule one job at a time, by selecting the job that completes first.
Figure 2.3 shows an example of FSP policy: in the example done for SRPT, with job j_1 with arrival time 0 and size 4 and job j_2 with arrival time 1 and size 2, FSP has the same behaviour of SRPT and, consequently, leads to the optimal mean response time. The difference is that while SRPT takes decisions based on the size of a job, FSP uses the finish time of jobs in PS, that can be seen in Figure 2.1b, to sort jobs.
FSP strenghts: fair, meaning that all jobs finish before or similarly to PS, and nearly optimal response time.

FSP weaknesses: requires a simulation to be able to run. While Friedman *et al.* [38] demonstrated that the implementation on single-server queue systems is straightforward, a simulative approach for a multi-server queue system can be heavy for the scheduler machine.

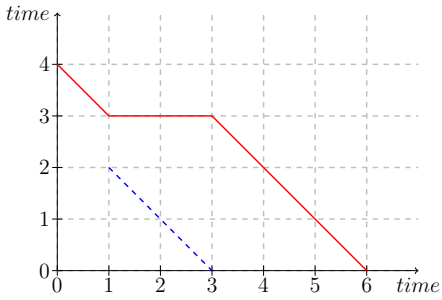


Figure 2.3: FSP for two jobs

2.1.3 MapReduce

MapReduce, popularized by Google [1], is both a data-intensive scalable computing system (DISC) and a programming model for DISC systems. MapReduce has been designed around some best-practices for DISC systems:

- **Commodity Hardware**: prefer a large number of commodity hardware to a small number of high-end computers.
- **Failures are the norm, not the exception**: design the system with failures in mind. If a failure occurs, the system must automatically recover and continue to work.
- **Move computation, not the data**: data is expensive to move, move computation where the data is instead.
- **Process data sequentially**: when large amounts of data are read from a hard disk, the fastest way to read it is reading sequentially from the start to the end. While reading data, processing that can be done locally (*e.g.* filtering and/or projections) is performed.
- **Hide what can be automatized**: MapReduce is a complex distributed framework but many parts of it can be hidden to the user. Let the user write only the important part of its program and automatize everything else.

MapReduce Distributed Architecture

The MapReduce cluster is composed by many machines that have different roles. There exist essentially two entities: the server, called JobTracker, which manages the other machines, and the clients, called TaskTracker, which are the machines that perform the work. When a TaskTracker wants to join the cluster, it contacts the JobTracker. After it is added to the cluster, the TaskTracker periodically contacts the JobTracker with messages that contain its status. By checking the differences between the old state and the new state of the TaskTracker, the JobTracker assigns work to the TaskTracker. If the JobTracker doesn't receive any message from a TaskTracker for a certain amount of time, it puts the TaskTracker in a blacklist and eventually removes the TaskTracker from the clients list.
A MapReduce Job is composed by two phases called respectively Map and Reduce. The Reduce phase can start only when the Map phase is completed and the output of the Map phase has been completely transfered to the machines in which the Reduce phase will run. Each phase is composed by Tasks, where each Task is a single unit of work that a TaskTracker can perform. Tasks can run in parallel and they do not communicate with the other Tasks (shared-nothing architecture). When a Map task completes its computation, it releases its resources and then sets up a HTTP server from which every Reduce task can pull its data. This data transfer is called the Shuffle phase and it is part of the Reduce phase[2]. When a TaskTracker contacts the JobTracker with some free resources, the JobTracker assigns some Tasks to the free resources. When the Tasks are completed, the TaskTracker contacts the JobTracker and the JobTracker updates the state of the Job. When all the Tasks of a Job phase are completed, then that job phase is complete.

MapReduce Programming Model

The MapReduce Programming Model has been designed to make as simple as possible to define jobs. Basically, a MapReduce Job is defined by a Map and a Reduce function:

$$map : (k_1, v_1) \rightarrow [(k_2, v_2)]$$
$$reduce : (k_2, [v_2]) \rightarrow [(k_3, v_3)]$$

The Map function takes in input a key-value pair and outputs a list of key-value pairs of a different type. The Reduce function takes in input a pair composed by a key and a list of values and produces a list of key-value pairs. The name of the two functions are inspired by the functions available in *Lisp* [1] to operate over lists.
A classical example of MapReduce program is WordCount, described in Algorithm 1. In WordCount, the Map function takes in input the line as value and the line offset in

[2] A Hadoop configuration parameter allows starting the Shuffle phase before the completion of the Map phase. This is dangerous, however, since Reduce tasks risk remaining idle while waiting for late Map tasks stragglers. Best practices advice a conservative choice for this parameter [2].

the file as key and "emits" each word of the line with a counter set to 1. The REDUCE function takes in input the counters for a single word, sum the counters together and "emits" a single tuple composed by the word as key and the sum as value. An optional function, called COMBINE, can aggregate results locally in order to minimize the traffic between MAP and REDUCE tasks. For the word-count example, the COMBINE function had the very same implementation of REDUCE.

```
fun map(offset,line):
    foreach term ∈ line:
        emit(term, 1)

fun reduce(term,counts):
    sum ← 0
    foreach count ∈ counts:
        sum ← sum + count
    emit(term, sum)
```

Algorithm 1: WordCount in MapReduce

The description of the two functions is not enough to define the job. The developer must also provide an input from which the lines of text must be read and an output to which the word and counter pairs are written.
Once the developer supplies these informations, MapReduce creates the job and executes it. The procedure of creating the job converts this simple job specification into a real job. The real job will have many components that are not defined by the developer directly, but that are required for the job to run.
First of all, a MapReduce job needs a way to read the input, for which we will use the name *input reader*. For a textual file, this component reads line by line the given file. The MAP function is applied to every line generated by the input reader.
The tuples created for a single REDUCE are sorted based on the key. During the SHUFFLE phase, the machine that must run the REDUCE function contacts all the machines that run the MAP function to fetch its data. The fetched data is then sorted based on the key. The REDUCE machine will apply the REDUCE function to each pair and the result of the REDUCE function will be written by an *output writer*. When the MAP phase reaches a progress threshold, the JOBTRACKER instructs a number of machines manually set by the user to run the REDUCE function. This threshold is usually set by the user.
Figure 2.4 shows an example of a MapReduce job.

Scheduling Policies for MapReduce

The *Task Scheduler* is the scheduler that assigns TASKTRACKER resources to tasks of one or more jobs in the queue. In Hadoop, the Task Scheduler is called every time a TASKTRACKER has some free resources and contacts the JOBTRACKER. The JOBTRACKER asks

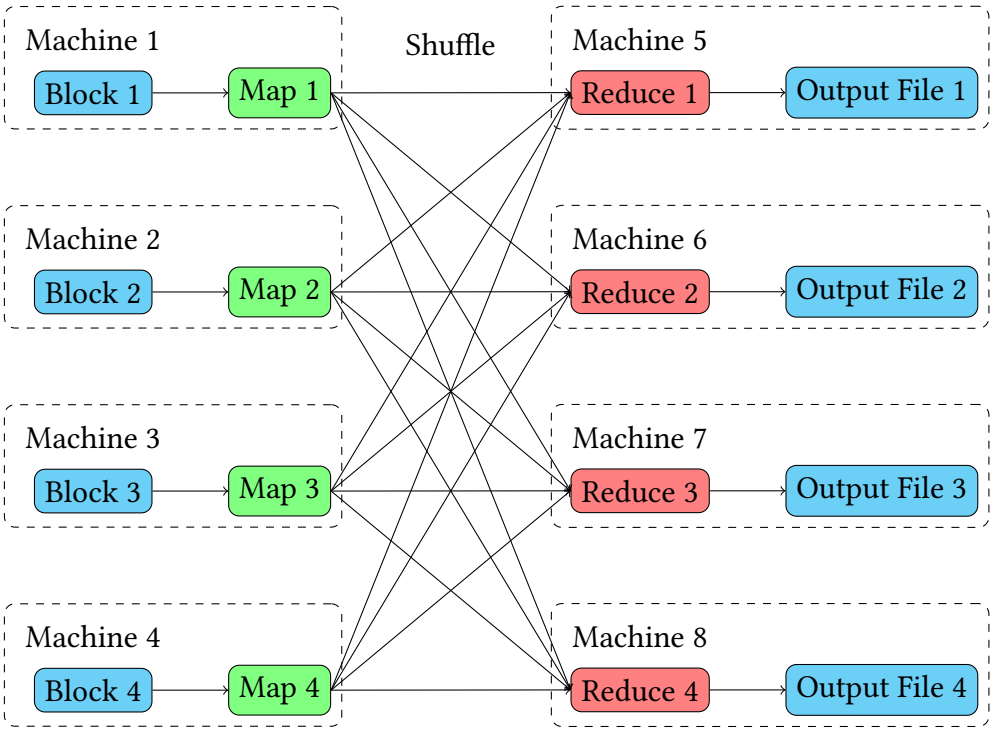


Figure 2.4: MapReduce Job Workflow

what tasks must be assigned for the given TaskTracker and the scheduler responds with a list of Tasks.

The default scheduling policy of Hadoop is FIFO, that grants resources to the job arrived first. In a system like MapReduce in which the size of a job can vary by different orders of magnitude, FIFO leads to poor performance for skewed workloads (Chapter 3).

The alternative is the Fair Scheduler [19], a scheduling policy inspired by Processor Sharing. The key idea of the Fair Scheduler is to split resources among jobs so that each job receives its fair share. This, of course, guarantees fairness but has a big impact on the performance of jobs. Basically, jobs progress slowly based on the number of jobs that are in the queue.

One key point of MapReduce is to move computation to the data instead of the data to the computation in order to limit network usage. While this strategy works very well, the fact that many jobs are scheduled in the same cluster can lead to occasions when tasks cannot be assigned to a machine with the input block required by the task. In this case, the Hadoop scheduler can choose to assign the task to another machine and then transfer through the network the block to that machine. This procedure is expensive and for this reason Zaharia *et al.* developed a modification to the Fair Scheduler called Delay Scheduling [33]. There are three levels of *data locality* for a task assigned to a TaskTracker: its input can be local to the machine where it has been assigned, it can be on a machine in the same rack of the machine where it has been assigned or elsewhere. The Delay Scheduling policy delays the assignment of tasks to machines

without the block for a certain amount of time. This threshold can be set manually and raises significantly the number of tasks that work with local blocks. The experimental results by Zaharia *et al.* show a significant improvement in the performance of the Fair Scheduler.

MapReduce Workload Composition

The evaluation of real scheduling policy implementations is a difficult task. In this thesis, this evaluation is done experimentally: a real MapReduce cluster has been deployed and real jobs are run (Chapter 5). Such an approach makes sense only when the cluster configuration and the workload composition are based on what it is used in real deployment of MapReduce. A lot of works have focused on studying real workloads and clusters.
Chen *et al.* [26] present a study of MapReduce workloads done across five Cloudera [119] customers and two years, 2009 and 2010, at Facebook [120]. The traces show that those workloads and clusters are very different. The clusters are obviously tailored on the company size. For the workloads, most jobs have input and output in the MB and GB range while there are some jobs with small ($<= 1KB$) or huge ($>= 1TB$) size. The aggregate result is that more than 92% of jobs manage $< 10GB$ of data. Workloads composition have no common pattern and the authors conclude that it is vital to tailor schedulers for each specific cluster. The output of their work is *SWIM* [17], a workload replay tool that can be used to test new techniques against realistic workloads.
While Chen *et al.* work focuses on industrial deployment, Ren *et al.* [87] analyze three different clusters used for research. The first result of their analysis regards how MapReduce jobs are written: most jobs are written with the standard API while many others use Hadoop Streaming, a way to analyze data with tools external to Hadoop, and part are written with Pig [105], a high-level language for Hadoop. Another interesting result is the distribution of job structures, where a structure is a classification of a job, *e.g.*, the number of iterative jobs in the three clusters. The second aspect of this work is the distribution of small and big jobs. The result is that most of the jobs are small, they process less than $10GB$ of data and they run for less that 8 minutes; there exist few outliers that process less than $100GB$ of data and run for more than 5 hours. Chen *et al.* conclude that small jobs can be problematic for clusters which do not have a scheduler designed or configured to support such jobs.
Appuswamy *et al.* offer a view of a Hadoop cluster usage at Microsoft [121]. The study, which covers 174.000 jobs submitted in 2011, shows that 80% of the jobs have an input smaller than $1TB$ and the median is $14GB$. A very interesting aspect of this work is that it offers an overview of the relationship between the input size and the total CPU time in a real cluster. From the data provided, we can say that almost every job is IO-intensive with few exceptions.

Preemption

Preemption is an important concept in scheduling in general, and there are several use cases in a system such as Hadoop that can benefit from such a primitive. In a system like Hadoop, where jobs are composed by tasks, there are two kinds of preemption: job preemption and task preemption. Task preemption means that a task that is running is stopped while job preemption means that a job that has just received resources won't receive any resource the next time the scheduler is called.
Job schedulers, like the Hadoop FAIR and Capacity schedulers, can use task preemption to warrant fairness [100]: if a job starves due to long-running tasks of another job, the latter may be preempted. In deadline scheduling [101], preemption can be used to make sure that jobs that are close to the deadline are run as soon as possible. Size-based schedulers [73, 72] in general attribute priorities to jobs according to a virtual or real size, and preemption can guarantee that higher-priority jobs are allowed to run earlier.
Currently, two job preemption strategies are available for Hadoop. One technique is to wait for tasks that should be preempted to complete: we call this Wait strategy. Another approach is to kill tasks, using the Kill primitive. Clearly, the first policy has the shortcoming of introducing large latencies for high-priority tasks, while the second one wastes work done by killed tasks. We refer to the work by Cheng *et al.* [102] for an approach that strives to mitigate the impact of the Kill strategy by adopting an appropriate eviction policy (*i.e.*, choosing which tasks to kill).
In Chapter 6, we present a proposal for a Suspend preemption primitive that avoids the weaknesses of Wait and Kill.

2.2 Related Work

We present the works related to this thesis.
Section 2.2.1 regards scheduling for single server queues with inexact sizes and it touches works related to the contribution we introduce in Chapter 3. Section 2.2.2 focuses on scheduling for DISC systems and Section 2.2.3 on job size estimation on DISC systems. They both touch topics related to HFSP, described in Chapter 4.
Finally, Section 2.2.4 discusses job preemption and is related to Chapter 6.

2.2.1 Size-Based Scheduling Policies with Inexact Job Sizes

Perhaps surprisingly, not much work considers the effect of inexact job size information on size-based scheduling policies. Lu *et al.* [70] have been the first to consider this problem, showing that size-based scheduling is useful only when job size evaluations are reasonably good (high correlation, greater than 0.75, between the real job size and its estimate). Their evaluation focuses on a single heavy-tailed job size distribution, and does not explain the causes of the observed results. We show the effect of different job size

distributions (heavy-tailed, memoryless and light-tailed), and we show how to modify the size-based scheduling policies to make them robust to job estimation errors.
Wierman and Nuyens [71] provide analytical results for a class of size-based policies, but consider an impractical assumption: results depend on a bound on estimation error. In the common case where most estimations are close to the real value but there are outliers, bounds need to be set according to outliers, leading to pessimistic predictions on performance. In our work, instead, we do not impose any bound on the error.
To the best of our knowledge, these are the only works targeting job size estimation errors for the single server queue (usually referred to as M/G/1 queue).

2.2.2 Scheduling for DISC Systems

The rising of DISC systems such as Hadoop has been followed by many research works on scheduling for those systems. In this Section we give an overview of the work related to HFSP.
Chang *et al.* [35] define a scheduler for nearly optimal total completion time. Their work is based on [36] where the analysis by Schulz *et al.* demonstrates that the problem of scheduling jobs to minimize the total completion time is NP-hard. Chang *et al.* transpose the problem in a multi-processor system and derive a 2-approximation algorithm for the problem. The provided scheduling policy achieves good performance in term of total completion time when the job sizes are known *a-priori*. A section of the paper shows that the scheduler could work with estimated job sizes, that is a prerequisite to use a size-based policy in a real-world system where the job size is unknown *a-priori*. Although the study of the error is interesting, it covers only errors up to 100% of the job size and our work shows that errors can be bigger. We believe a deeper study of the consequences of estimating job size is very important to understand the real performance of the scheduler in production. Our scheduler is designed to deal with estimation error and our work focuses also on explaining how to work with estimated sizes.
Moseley *et al.* [11] model the MapReduce scheduling problem in term of two-stage flexible flow-shop problem and then create a scheduler to minimize the total flow-time. To the best of our knowledge, this is one of the best models for MapReduce because it takes into account important details of a MapReduce-like system, *e.g.* the relationship between Map and Reduce tasks in a MapReduce job. The model does not consider the shuffle phase of a job nor the data-locality problem. While this is understandable from a theoretical point of view, our HFSP scheduler is a more practical example of scheduler for MapReduce that takes into account all the details of a real system. Another very important difference between their work and ours is that they consider job size to be known *a-priori* and they justify this by saying that size can be determined based on the cluster history. Even by taking into account the cluster history, the job size must be estimated and this estimation leads to errors on the estimated size. These errors can have an impact on the scheduling policy and, in practice, its study is mandatory to provide a

working scheduling policy for MapReduce. Our work shows that the lack of information about job size is a fundamental aspect for a scheduler of MapReduce and has an impact on the design and the performance of the scheduler.
Interestingly, Moseley *et al.* acknowledge that MapReduce does not support preemption but still use it for their results. Indeed, most of the interesting schedulers use preemption to achieve their performance. Our work on task preemption, presented on Chapter 6, addresses exactly this problem.
Sandholm *et al.* [41] provide a way to virtualize MapReduce clusters so that the performance of each cluster can be changed dynamically. While this is very interesting in many practical scenarios, for instance in Amazon's EC2 clusters, the target of this work is to help the cluster utilizer understand costs, performance and risks in such context. They also provide some strategies for the Hadoop Schedulers to prioritize jobs based on the submitter. While their and our work are both about job scheduling, the context in which this job scheduling is done is different and complementary. Indeed, it is possible to include their strategies in our scheduler.
Ghodsi *et al.* [39] propose a scheduling policy, called *Dominant Resource Fairness*, that strives to achieve fairness while considering multiple resource types, such as IO and CPU. They use max-min fairness as base scheduling policy and while their goal is to create a fair scheduler, they also consider the mean response times of jobs showing that their scheduler is more fair and efficient than the Fair Scheduler.
Kc *et al.* [101] design a scheduler for Hadoop called the *Constraint Scheduler*. Its goal is to satisfy the deadlines of the submitted jobs, where a deadline is a time constraint set by the user that submitted the job. The main difference with ours is, of course, the fact that we do not consider deadlines and they do not consider the mean response time.
Another deadline scheduler is *Automatic Resource Inference and Allocation* (ARIA), proposed by Verma *et al.* [47]. This work is highly related to our work because, to meet job deadlines, ARIA creates a job profile while the job is running containing job performance informations. This resembles our idea of estimating the job size. The job profile is used to estimate the completion time of the job and consequently to be able to schedule resources so that jobs can meet their deadlines. From their experiments it is possible to see that their predicted job sizes are similar to the real job sizes and this reinforces our idea that job size can be estimated in MapReduce without huge errors. As for the Constraint Scheduler, ARIA does not take into account the mean response time of jobs and focuses on the deadlines.
Verma *et al.* propose another kind of scheduler for minimizing the overall completion time of a MapReduce workload [48]. The idea is similar to the base idea of our scheduling policy: sort jobs in the "right" order based on their sizes. A big difference between our work and theirs is that they use past runs of the same jobs to infer the performance of future runs of those jobs. While we acknowledge that a part of jobs in a cluster are executed periodically, many studies confirm that a large part of the jobs, such as orchestration jobs, are not. Also, even periodic jobs can have a different size based on

the input and the machines on which they are running. We believe our approach to be easier to apply and adapt to a wider range of situations.
Tian *et al.* [30] define both a way to classify jobs based on their characteristics, named *MR-Predict*, and a new scheduling policy, called *Triple-Queue Scheduler* that uses the classifier to put jobs inside one of the three queues. The Triple-Queue Scheduler assigns tasks to slots with the goal of balancing CPU and IO usage on TaskTracker. The Triple-Queue Scheduler has a different goal from HFSP and the two approaches are complementary since each queue can use a size-based policy such as HFSP.
Tan *et al.* [42] propose a study of FIFO and Fair Scheduler with different kind of workload as well as a new scheduling policy for MapReduce called the *Coupling Scheduler*. The Coupling Scheduler is a modified version of the Fair Scheduler which assigns Reduce tasks gradually depending on the Map phase progress. Their results show that the Coupling Scheduler is better than the Fair Scheduler. Their strategy to enhance the Fair Scheduler does not change the blind-to-size nature of the Fair Scheduler and thus we conclude that in many cases the Coupling Scheduler will be less efficient than HFSP. Our work extensively shows that size-based disciplines have the best performance even in presence of errors. This work is complementary to HFSP and we believe it could be included in our scheduler to avoid assigning reducers too aggressively.
Flex [73] is a size-based scheduler for Hadoop which is available as a proprietary commercial solution. In Flex, "fairness" is defined as avoiding job starvation and guaranteed by allocating a part of the cluster according to Hadoop's Fair scheduler; size-based scheduling (without aging) is then performed only on the remaining set of nodes. Flex is part of IBM InfoSphere BigInsights [123] that is not open source. Consequently, even if we believe that Flex is the most similar scheduler to HFSP available for Hadoop, we could not test it against our scheduler.

Framework Schedulers

Recent works have pushed the idea of sharing cluster resources at the framework level, for example to enable MapReduce and Spark [3] "applications" to run concurrently. Monolithic schedulers such as YARN [8] and Omega [10] use a single component to allocate resources to each framework, while two-level schedulers [4, 9] have a single manager that negotiates resources with independent, framework-specific schedulers. We believe that such framework schedulers impose no conceptual barriers for size-based scheduling, but the implementation would require very careful engineering. In particular, size-based scheduling should only be limited to batch applications rather than streaming or interactive ones that require continuous progress.

2.2.3 Job Size Estimation in MapReduce

In this Section we present the work done on estimating the size of a job that are not part of the previous Section.

Various recent approaches [52, 31] propose techniques to estimate query sizes in recurring MapReduce jobs and in database queries [83]. Agarwal *et al.* [52] report that recurring jobs are around 40% of all those running in Bing's production servers. Our estimation module, on the other hand, works on-line with *any* job submitted to a Hadoop cluster, but it has been designed so that the estimator module can be easily plugged with other mechanisms, benefitting from advanced and tailored solutions. For example, the estimation error can be always evaluated *a posteriori*, and this evaluation can be used to decide if the size-based scheduling works better than policies blind to size.

2.2.4 Preemption for MapReduce systems

A recent preemption mechanism for Hadoop is Natjam [65]: unlike in our work, where we use the OS to perform job suspension and resuming, Natjam operates at the "application layer", and saves counters about task progress, which allow to resume tasks by fast-forwarding to their previous states. Since the state of the Java Virtual Machine (JVM) is lost, however, Natjam cannot be applied seamlessly to arbitrary tasks: indeed, many MapReduce programming patterns involve keeping track of a state within the task JVM [103]; this problem is exacerbated by the fact that many MapReduce jobs are created by high-level languages such as Apache Pig [105] or Apache Hive [104]: jobs compiled by these frameworks are highly likely to make use of these "tricks", which hinders the application of Natjam.

Natjam proposes to handle such stateful tasks with hooks that systematically serialize and deserialize task state. Besides requiring manual intervention to support suspension, this approach has the drawback of always requiring the overhead for serialization, writing to disk, and deserialization of a state that could be large. In contrast, our approach does not incur in a systematic serialization overhead, since it relies on OS paging to swap to disk the state of the tasks, *if* and *when* needed.

Chapter 3

Size-based scheduling with estimated sizes

Size-based scheduling policies are well known to be efficient and fair when job sizes are known. While the properties of those schedulers are very interesting from a theoretical perspective, their practical implementation poses some problems. In many cases, in fact, the job size is unknown and must be estimated while the job is running. When the scheduler estimates the job sizes, it makes errors that have an impact on the overall performance of the system.
In the literature, the study of such an impact has not been addressed exhaustively, not even in the single server case. Before starting the design of a scheduler for DISC systems, therefore, we need to understand precisely the influence of inexact job size information on the scheduling decision. In order to isolate such influence, we consider the basic *single server case*, so that to obtain useful insights that will be used in later chapters during the design of our scheduler.
In this Chapter, therefore, we present a study of the state-of-the-art size-based scheduling policies for single server queues in presence of errors on the estimated sizes (Section 3.1.1). Based on the observations on the impact of estimation errors on the scheduling decisions, we then propose a new scheduling technique derived from the existing ones, that performs similarly to existing schedulers also in presence of estimation errors (Section 3.1.2).
In Section 3.2 we describe the evaluation methodology, while in Section 3.3 we provide an extensive set of results of the different schedulers.

3.1 Scheduling Based on Estimated Sizes

We introduce the SRPTE and FSPE size-based scheduling protocols in Section 3.1.1; afterwards, in Section 3.1.2, we describe FSPE+PS, a size-based scheduling protocol based on FSPE that works also in conditions in which FSPE has problems.

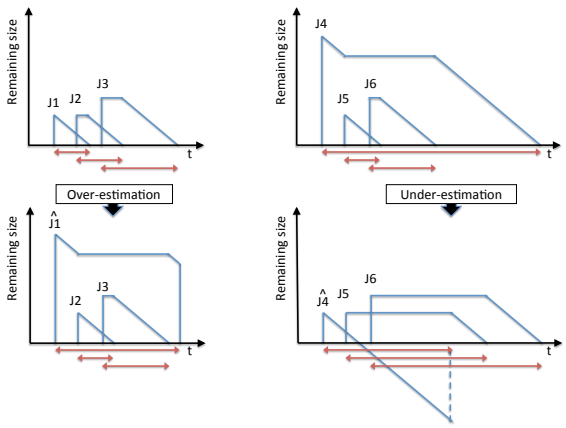


Figure 3.1: Examples for job under- and over-estimation.

3.1.1 SRPTE and FSPE

SRPTE and *FSPE* are variations of, respectively, SRPT and FSP that work with estimated sized instead of real ones.

In Figure 3.1, we provide an illustrative example where a single job size is over- or under-estimated while the others are estimated correctly, focusing (because of its simplicity) on the behavior of SRPTE; job sojourn times are represented by the horizontal arrows. The left column of Figure 3.1 illustrates the effect of over-estimation. In the top, we show how the scheduler behaves without errors, while in the bottom we show what happens when the size of job $J1$ is over-estimated. The graphs shows the remaining (estimated) processing time of the jobs over time (assuming a normalized service rate of 1). Without errors, jobs $J2$ does not preempt $J1$, and $J3$ does not preempt $J2$. Instead, when the size of $J1$ is over-estimated, both $J2$ and $J3$ preempt $J1$. Therefore, the only job suffering (*i.e.*, experiencing higher sojourn time) is the one that has been over-estimated. Jobs with smaller sizes are always able to preempt an over-estimated job, therefore the basic property of SRPT (favoring small jobs) is not significantly compromised.

The right column of Figure 3.1 illustrates the effect of under-estimation. With no estimation errors (top), a large job, $J4$, is preempted by small ones ($J5$ and $J6$). If the size of the large job is under-estimated (bottom), its estimated remaining processing time eventually reaches zero: we call *late* a job with zero or negative estimated remaining processing time. *A late job cannot be preempted* by newly arrived jobs, since their size estimation will always be larger than zero. In practice, since preemption is inhibited, the under-estimated job *blocks the system* until the end of its service, with a negative impact on multiple waiting jobs.

This phenomenon is particularly harmful when job sizes are heavily skewed: if the workload has few very large jobs and many small ones, a single late large job can significantly delay several small ones, which will need to wait for the late job to complete before having an opportunity of being served.

Even if the impact of under-estimation seems straightforward to understand, surprisingly *no work in the literature has ever discussed it.* To the best of our knowledge, we are the first to identify this problem, which significantly influences scheduling policies dealing with inaccurate job size.
In FSPE, the phenomena we observe are analogous: job size over-estimation delays only the over-estimated job; under-estimation can result in jobs terminating in the virtual PS queue before than in the real system. We therefore define *late* jobs in FSPE as those whose execution is completed in the virtual system but not yet in the real one and we notice that, analogously to SRPTE, also in FSPE late jobs can never be preempted by new ones, and they block the system until they are all completed.

3.1.2 FSPE+PS

Now that we have identified the issue with existing size-based scheduling policies, we propose our countermeasure. Several alternatives are envisionable, including for example updating job size estimations if new information becomes available as work progresses: such a solution may not however be always feasible, due to limitations in terms of information or computational resources available to the scheduler.
We propose, instead, a simple solution that requires no additional job size estimation, based on the simple idea that *late jobs should not prevent executing other ones.* This goal is achievable by performing simple modifications to preemptive size-based scheduling disciplines such as SRPT and FSP. The key property is that the scheduler takes corrective actions when one or more jobs are *late*, guaranteeing that – even when very large late jobs are being executed – newly arrived small jobs will get executed soon.
We show here *FSPE+PS*, which is a modification to FSPE: the only difference is that, when one or more jobs are late, (*i.e.*, they have completed in the emulated virtual system and not in the real one), *all late jobs are scheduled concurrently in a PS fashion.* FSPE+PS inherits from FSP and FSPE the guarantee that starvation is absent, it is essentially as complex to implement as FSP is and, as we show in Section 3.3, it performs close to optimally in most experimental settings we observe. Due to the dominance of FSP with respect to PS, if there are no size estimation errors no jobs can ever become late: therefore, with no error FSPE+PS is equivalent to FSP.
Several alternatives to FSPE+PS are possible: we experimented for example with similar policies that are based on SRPT rather than on FSP, that use a least-attained-service policy rather than a PS one for late jobs, and/or that schedule aggressively jobs that are not late yet as soon as at least one reaches the "late" stage. With respect to the metrics we use in this work, their behavior is very similar to the one of FSPE+PS, and for reasons of conciseness we do not report about them here. We however encourage the interested reader to examine their implementation at bit.ly/schedulers. In practice, most of the performance gain is due to the *explicit management of the late jobs*, and how late jobs are handled has no significant impact on such a gain.

```
fun NextVirtualCompletionTime:
    if |O| = 0: return -1
    else: return t + w_0 * |O|
fun ProcessJob:
    if |L| ≠ 0: return {(l_i, 1/|L|) | l_i ∈ L}
    elif |O| = 0: return ∅
    else:
        k ← min{i | c_i}
        return {(j_k, 1)}
fun UpdateVirtualTime(s):
    for (_, w_i, _) ∈ O:  w_i ← w_i − (s − t)/|O|
    t ← s
fun VirtualJobCompletion(s):
    UpdateVirtualTime(s)
    if c_0: add j_0 to L
    remove the first element from O
fun RealJobCompletion(j):
    find i such that j_i = j
    c_i ← False
fun JobArrival(s, j, w):
    UpdateVirtualTime(s)
    insert (j, w, True) in O maintaining ordering
```

Algorithm 2: FSPE+PS.

Algorithm 2 presents our implementation of FSPE+PS, which is based on Friedman and Henderson's original description of FSP [75, Section 4.4]. System state is kept in three variables: the virtual PS queue state is kept in a list $\mathcal{O}$, containing (j_i, w_i, c_i) tuples and ordered by the w_i values: each such tuple represents a job j_i having remaining processing time w_i in the virtual system, while the c_i boolean flag is set to True if j_i is running in the real system; late jobs are stored in a $\mathcal{L}$ set; the variable t stores the last time at which the information in $\mathcal{O}$ has been updated.

Computation is triggered by three events: if a job j of estimated size w arrives at time s, JobArrival(s, j, w) is called; when a job j completes, RealJobCompletion(j) is called; finally, when a job completes in the virtual system at time s, UpdateVirtualTime(s) is called (NextVirtualCompletionTime is used to discover when to call VirtualJobCompletion). After each event, the ProcessJob procedure is called to determine the new set of scheduled jobs: its output is a set of (j, s) pairs where j is the job identifier and s is the fraction of system resources allocated to it.

3.2 Evaluation Methodology

Understanding size-based scheduling systems when there are estimation errors is not a simple task. The complexity of the system makes an analytical study feasible only if strong assumptions, such as a bounded error [71], are imposed. Moreover, to the best of our knowledge, no analytical model for FSP (without estimation error) is available, making an analytical evaluation of FSPE and FSPE+PS even more difficult.
For these reasons, we evaluate our proposed scheduling policies through simulation. The simulative approach is extremely flexible, allowing to take into account several parameters – distribution of the arrival times, of the job sizes, of the errors. Previous simulative studies (*e.g.*, [70]) have focused on a subset of these parameters, and in some cases they have used real traces. In our work, we developed a tool that is able to both reproduce real traces and generate synthetic ones. Moreover, thanks to the efficiency of the implementation, we were able to run an extensive evaluation campaign, exploring a large parameter space. For these reasons, we are able to provide a broad view of the applicability of size-based scheduling policies, and show the benefits and the robustness of our solution with respect to the existing ones.

3.2.1 Scheduling Policies Under Evaluation

In this work, we take into account different scheduling policies, both size-based and blind to size. For the size-based disciplines, we consider SRPT as a reference for its optimality with respect to the MST. When introducing the errors, we evaluate SRPTE, FSPE and our proposal, FSPE+PS, described in Section 3.1.
For the scheduling policies blind to size, we have implemented the *First In, First Out* (FIFO) and *Processor Sharing* (PS) disciplines. These policies are the default disciplines used in many scheduling systems – *e.g.*, the default scheduler in Hadoop [2] implements a FIFO policy, while Hadoop's FAIR scheduler is inspired by PS; the Apache web server delegates scheduling to the Linux kernel, which in turn implements a PS-like strategy [85]. Since PS scheduling divides the resources evenly among running jobs, it is generally considered as a reference for its fairness (see the next section on the performance metrics). Finally, we consider also the *Least Attained Service* (LAS) [74] policy. LAS scheduling, also known in the literature as *Foreground-Background* (FB) [76] and *Shortest Elapsed Time* (SET) [77], is a preemptive policy that gives service to the job that has received the least service, sharing it equally in a PS mode in case of ties. LAS scheduling has been designed considering the case of heavy-tailed job size distributions, where a large percentage of the total work performed in the system is due to few very large jobs, since it gives a higher priority to small jobs than what PS would do.

Table 3.1: Simulation parameters.

Parameter	Explanation	Default
sigma	σ in the log-normal error distribution	0.5
shape	shape for Weibull job size distribution	0.25
timeshape	shape for Weibull inter-arrival time	1
njobs	number of jobs in a workload	10,000
load	system load	0.9

3.2.2 Parameter Settings

Our goal is to empirically evaluate scheduling policies in a wide spectrum of cases. Table 3.1 synthetize the parameters that our simulator can accept as inputs; they are explained in detail in the following.

Job Size Distribution

Job sizes are generated according to a Weibull distribution, which allows us to evaluate both heavy-tailed and light-tailed job size distributions. Indeed, the ***shape*** parameter allows to interpolate between heavy-tailed distributions (shape < 1), the exponential distribution (shape$= 1$), the Raleigh distribution ($\text{shape} = 2$) and bell-shaped distributions centered around the '1' value ($\text{shape} > 2$). We set the *scale* parameter of the distribution to ensure that its mean is 1.
Since scheduling problems have been generally analyzed on heavy-tailed workloads with job sizes using distributions such as Pareto, we consider a default heavy-tailed case of $\text{shape} = 0.25$. In our experiments, we vary the shape parameter between a very skewed distribution with $\text{shape} = 0.125$ and a bell-shaped distribution with $\text{shape} = 4$.

Size Error Distribution

We consider log-normally distributed error values. A job having size s will be estimated as $\hat{s} = sX$, where X is a random variable with distribution

$$\text{Log-}\mathcal{N}(0, \sigma^2). \tag{3.1}$$

This choice satisfies two properties: first, since error is multiplicative, the absolute error $\hat{s} - s$ is proportional to the job size s; second, under-estimation and over-estimation are equally likely, and for any σ and any factor $k > 1$ the (non-zero) probability of under-estimating $\hat{s} \leq \frac{s}{k}$ is the same of over-estimating $\hat{s} \geq ks$. This choice also is substanciated by empirical results: in our implementation of the HFSP scheduler for Hadoop [72], we found that the empirical error distribution was indeed fitting a log-normal distribution. The ***sigma*** parameter controls σ in Equation 3.1, with a default – used if no other information is given – of 0.5; with this value, the median factor k reflecting relative error

is 1.40. In our experiments, we let sigma vary between 0.125 (median k is 1.088) and 4 (median k is 14.85).
It is possible to compute the correlation between the estimated and real size as σ varies. In particular, when sigma is equal to 0.5, 1.0, 2.0 and 4.0, the correlation coefficient is equal to 0.9, 0.6, 0.15 and 0.05 respectively.
The mean of this distribution is always larger than 1, and growing as sigma grows: the system is biased towards overestimating the aggregate size of several jobs, limiting the underestimation problems that FSPE+PS is designed to solve. Even in this setting, the results in Section 3.3 show that the improvements obtained by using FSPE+PS are still significant.

Job Arrival Time Distribution

For the job inter-arrival time distribution, we use a Weibull distribution for its flexibility to model heavy-tailed, memoryless and light-tailed distributions. We set the default of its shape parameter (***timeshape***) to 1, corresponding to "standard" exponentially distributed arrivals. Also here, timeshape varies between 0.125 (very bursty arrivals separated by long intervals) and 4 (regular arrivals).

Other Parameters

The ***load*** parameter is the mean arrival rate divided by the mean service rate. As default value, we use the same value of 0.9 used by Lu *et al.* [70]; in our experiments we let the load parameter vary between 0.5 and 0.999.
The number of jobs (***njobs***) in each simulation round is 10,000 (in additional experiments – not shown for space reasons – we varied this parameter, without obtaining significant differences). For each experiment, we perform at least 30 repetitions, and we compute the confidence interval for a confidence level of 95%. For very heavy-tailed job size distributions (shape ≤ 0.25), results are very variable and therefore, in order to obtain stable averages, we performed hundreds and/or thousands of experiment runs, until the confidence levels have reached the 5% of the estimated values.

3.2.3 Simulator Implementation Details

Our simulator is available under the Apache V2 license at https://bitbucket.org/bigfootproject/schedsim. It has been conceived with ease of prototyping in mind: for example, our implementation of FSPE as described in Section 3.1 requires 53 lines of code. Workloads can be both replayed from real traces and generated synthetically.
The simulator has been written with a focus on computational efficiency. It is implemented using an event-based paradigm, and we used efficient data structures based on B-trees (stutzbachenterprises.com/blist/). As a result of these choices, a "default" workload of 10,000 jobs is simulated in around half a second, while using a single core in

our machine with an Intel T7700 CPU. We use IEEE 754 double-precision floating point values to represent time and job sizes.

3.3 Experimental Results

We now present our experimental findings. For all the results shown in the following, the parameters whose values are not explicitly stated take the default values shown in Table 3.1. For the readability of the figures, we do not show the confidence intervals: for all the points, in fact, we have performed a number of runs sufficiently high to obtain a confidence interval smaller than 5% of the estimated value. We first present our results on synthetic workloads generated according to the methodology of Section 3.2.2; we then show the results by replaying two real-world traces from workloads of Hadoop and of a Web cache.

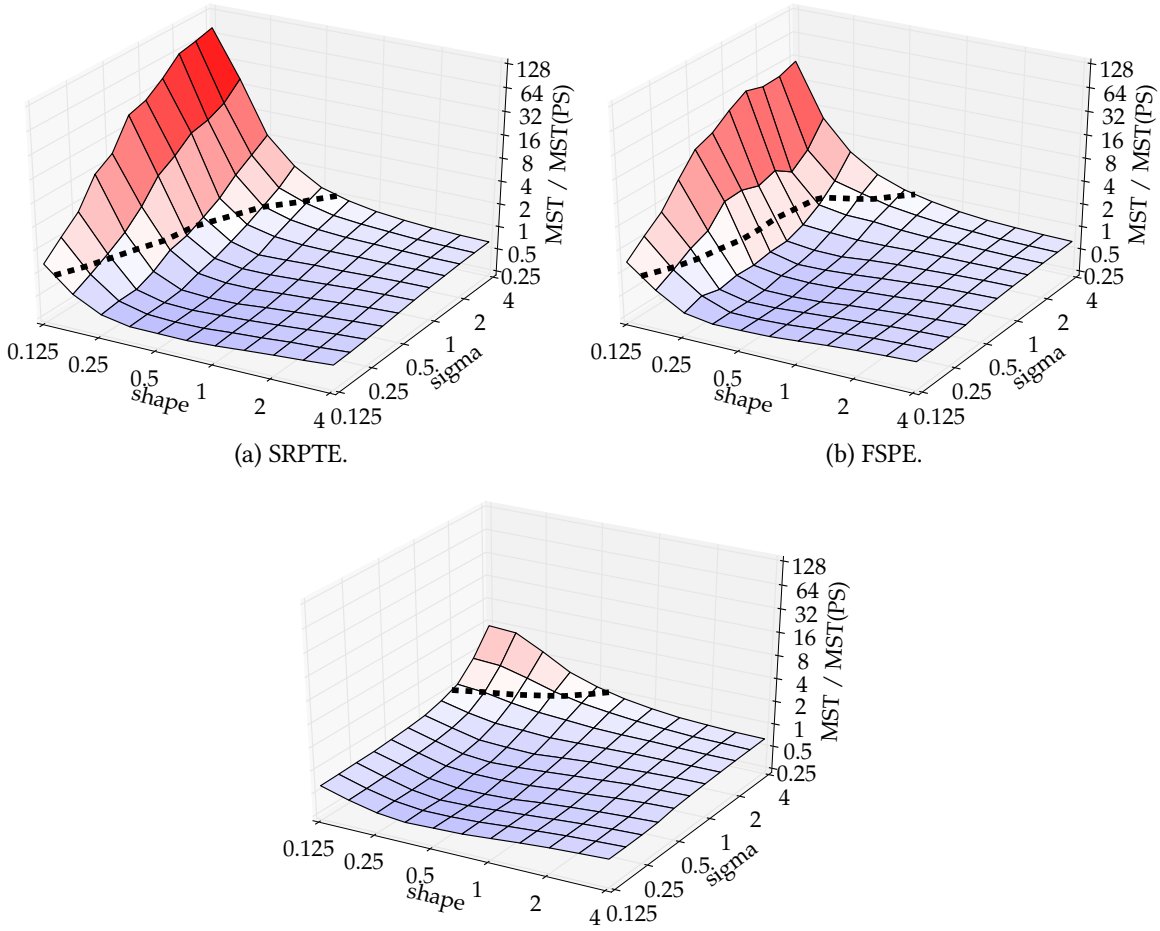
3.3.1 Synthetic Workloads

Mean Sojourn Time Against PS

We begin our analysis by comparing the three size-based scheduling policies, using PS as a baseline because PS and its variants are the most widely used set of scheduling policies in real systems. In Figure 3.2 we plot the value of the MST obtained using respectively SRPTE, FSPE and FSPE+PS, normalizing it against the MST of PS. We vary the sigma and shape parameters influencing respectively job size distribution and error rate; we will show that these two parameters are the ones that influence performance the most. Values lower than one (below the dashed line in the plot) represent regions where size-based schedulers perform better than PS.

In accordance with intuition and to what is known from the literature, we observe that the performance of size-based scheduling policies depends on the accuracy of job size estimation: as sigma grows, performance suffers. From Figures 3.2a and 3.2b, we however observe a new phenomenon: *job size distribution impacts performance even more than size estimation error.* On the one hand, we notice that large areas of the plots (shape > 0.5) are almost insensitive to estimation errors; on the other hand, we see that MST becomes very large as job size skew grows (shape < 0.25). We attribute this latter phenomenon to the fact that, as we highlight in Section 3.1, late jobs whose estimated remaining (virtual) size reaches zero are never preempted. If a large job is under-estimated and becomes *late* with respect to its estimation, small jobs will have to wait for it to finish in order to be served.

As we see with Figure 3.2c, *FSPE+PS outperforms PS in a large class of heavy-tailed workloads* where SRPTE and FSPE suffer. The net result is that a size-based policy such as FSPE+PS is outperformed by PS only in extreme cases where *both* the job size distribution is extremely skewed *and* job size estimation is very imprecise.



(a) SRPTE.

(b) FSPE.

(c) FSPE+PS.

Figure 3.2: Mean sojourn time against PS.

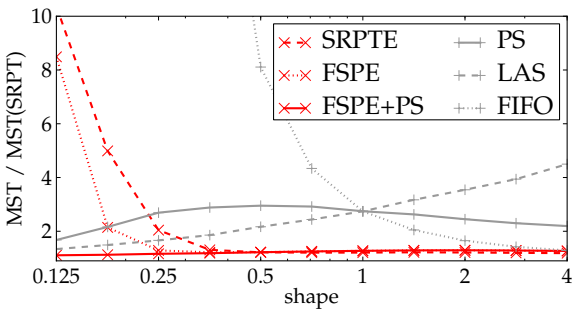


Figure 3.3: Impact of shape.

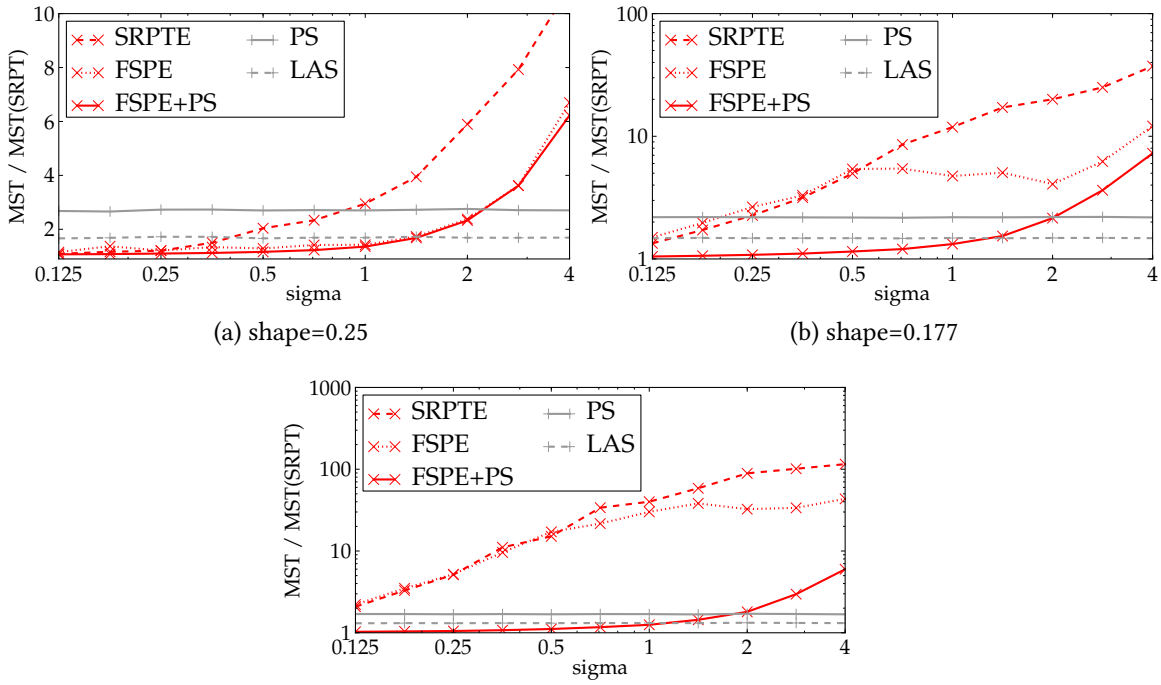
It may appear surprising that, when job size skew is not extreme, size-based scheduling can outperform PS even when size estimation is very imprecise: even a small correlation between job size and its estimation can direct the scheduler towards choices that are beneficial on aggregate. In fact, as we see more in detail in the following, sub-optimal scheduling choices become less penalized as the job size skew diminishes.

Impact of shape

We now delve into details and examine how schedulers perform when compared to the optimal MST that SRPT obtains. In the following Figures, we show the ratio between the MST obtained with the scheduling policies we implemented and the optimal one of SRPT.
From Figure 3.3 on the previous page, we see that the shape parameter is fundamental for evaluating scheduler performance. We notice that FSPE+PS has *almost optimal performance for all shape values considered* with the default sigma=0.5, which corresponds to a correlation coefficient between job size and it estimate of 0.9, while SRPTE and FSPE perform poorly for highly skewed workloads. Regarding non size-based policies, PS is outperformed by LAS for heavy-tailed workloads (shape < 1) and by FIFO for light-tailed ones having shape > 1 ; PS provides a reasonable trade-off when the job size distribution is unknown. When the job size distribution is exponential (shape $= 1$), non size-based scheduling policies perform analogously; this is a result which has been proven analytically (see *e.g.* the work by Harchol-Balter [80] and the references therein). It is interesting to consider the case of FIFO: in it, jobs are scheduled in series, and the priority between jobs is not correlated with job size: indeed, the MST of FIFO is equivalent to the one of a random scheduler executing jobs in series [84]. FIFO can be therefore seen as the limit case for a size-based scheduler such as FSPE or SRPTE when estimations carry no information at all about job sizes; the fact that errors become less critical as skew diminishes can be therefore explained with the similar patterns observed for FIFO.

Impact of sigma

The shape of the job size distribution is fundamental in determining the behavior of scheduling algorithms, and heavy-tailed job size distributions are those in which the behavior of size-based scheduling differs noticeably. Because of this, and since heavy-tailed workloads are central in the literature on scheduling, we focus on those.
In Figure 3.4 on the facing page, we show the impact of the sigma parameter representing the error for three heavily skewed workloads. In all three plots, the values for FIFO fall outside of the plot. These plots demonstrate that FSPE+PS is robust with respect to errors in all the three cases we consider, while SRPTE and FSPE suffer as the skew among job sizes grows. In all three cases, FSPE+PS performs better than PS as long as sigma is lower than 2: this corresponds to lax bounds on size estimation quality, requiring a correlation coefficient between job size and its estimate of 0.15 or more.
In all three plots, FSPE+PS performs better than SRPTE; the difference between FSPE+PS and FSPE, instead, becomes discernible only for shape < 0.25. We explain this difference by noting that, when several jobs are in the queue, size reduction in the virtual queue of FSPE is slow: this leads to fewer jobs being late and therefore non preemptable. As the distribution becomes more heavy-tailed, more jobs become late in FSPE and differ-



(a) shape=0.25

(b) shape=0.177

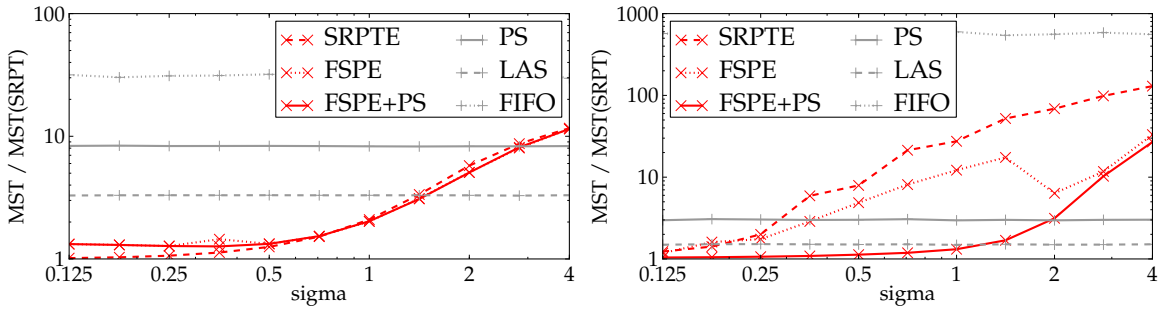
(c) shape=0.125

Figure 3.4: Impact of error on heavy-tailed workloads, sorted by growing skew.

ences between FSPE and FSPE+PS become more significant, reaching differences of even around one order of magnitude.
In particular in Figure 3.4b, there are areas ($0.5 < \text{sigma} < 2$) in which increasing errors decreases (slightly) the MST of FSPE. This counterintuitive phenomenon is explained by the characteristics of the error distribution: the mean of the log-normal distribution grows as sigma grows, therefore the aggregate amount of work for a set of several jobs is more likely to be over-estimated; this reduces the likelihood that several jobs at once become late and therefore non-preemptable. In other words, FSPE works better with estimation means that tend to over-estimate job size; it is however always better to use FSPE+PS, which provides a more reliable and performant solution to the same problem.

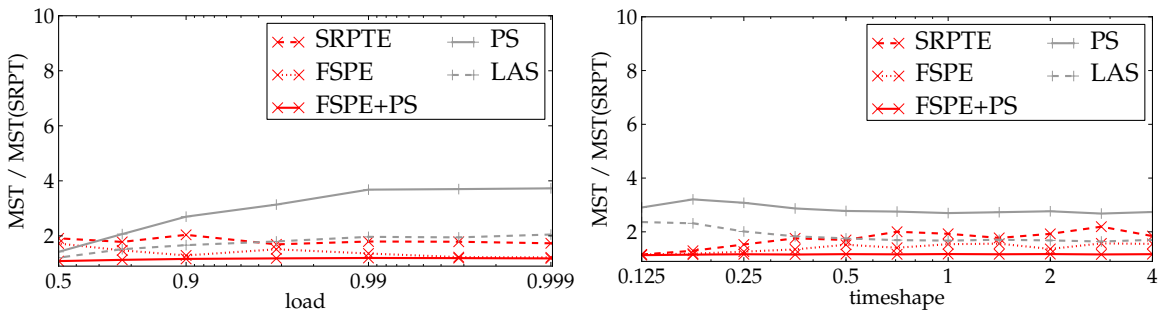
Pareto Job Size Distribution

In the literature, workloads are often generated using the Pareto distribution. To help comparing our results to the literature, in Figure 3.5 on the next page we show results for job sizes having a Pareto distribution, using $x_m = 0$ and $\alpha = \{1, 2\}$. The results we observe for the Weibull distribution are still qualitatively valid for the Pareto distribution; the value of $\alpha = 1$ is roughly comparable to a shape of 0.15 for the Weibull distribution, while $\alpha = 2$ is comparable to a shape of around 0.5, where the three size-based disciplines we take into account still have similar performance.



(a) $\alpha = 2$ (b) $\alpha = 1$

Figure 3.5: Pareto job size distributions, sorted by growing skew.



(a) Varying load. (b) Varying timeshape.

Figure 3.6: Impact of load and timeshape.

Impact of Other Parameters

In Figure 3.6, we show the impact of varying the load and timeshape parameters, while keeping sigma and shape at their default values.

Figure 3.6a shows that performance of size-based scheduling protocols is not heavily impacted by load, as the ratio between the MST obtained and the optimal one remains roughly constant (note that the graph shows a ratio, not the absolute values, that increase as the load increases); conversely, non size-based schedulers such as PS and LAS deviate more from optimal as the load grows.

Figure 3.6b shows the impact of changing the timeshape parameter: with low values of timeshape, job submissions are bursty and separated by long pauses; with high values, job submissions are evenly spaced. We note that size-based scheduling policies respond very well to bursty submissions where several jobs are submitted at once: in this case, adopting a size-based policy that focuses all the system resources on the smallest jobs pays best; as the intervals between jobs become more regular, SRPTE and FSPE become slightly less performant; FSPE+PS remains close to optimal.

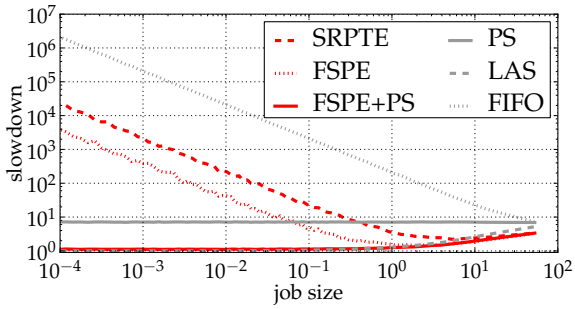


Figure 3.7: Mean conditional slowdown.

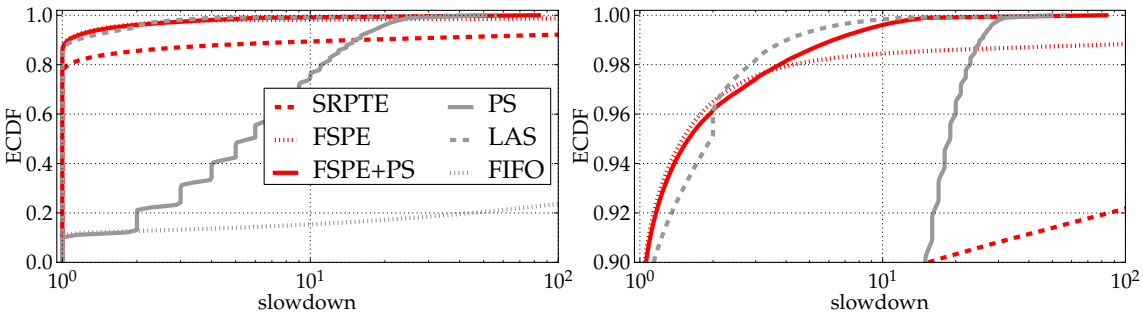


Figure 3.8: Per-job slowdown: full CDF (top) and zoom on the 10% more critical cases (bottom).

Conditional Slowdown

We now consider the topic of fairness, intending here – as discussed in Section 2 – that jobs' running time should be proportional to their size, and therefore not experience large slowdowns.

To better understand the reason for the unfairness of FIFO, SRPTE and FSPE, in Figure 3.7 we evaluate *mean conditional slowdown*, comparing job size with the average slowdown (job sojourn time divided by job size) obtained at that size using our default simulation parameters. The figure has been obtained by sorting jobs by size and binning them in 100 equally sized classes of jobs with similar size; points plotted are obtained by averaging job size and slowdown in each of the 100 class.

The almost parallel lines of FIFO, SRPTE and FSPE for smaller jobs are explained by the fact that, below a certain size, *job sojourn time is essentially independent from job size*: indeed, it depends on the total size of older (for FIFO) or late (for SRPTE and FSPE) jobs at submission time.

We confirm experimentally the fact that the expected slowdown in PS is constant, irrespectively of job size [81]; FSPE+PS and LAS, on the other hand, have close to optimal slowdown for small jobs. The better MST of FSPE+PS is instead due to better performance for larger jobs, which are more penalized in LAS.

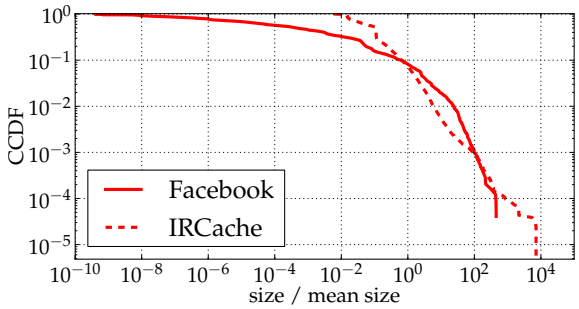


Figure 3.9: CCDF for the real workloads.

Per-Job Slowdown

The results we have shown testify that, for FSPE+PS and similarly to LAS, slowdown values are homogeneous across classes of job sizes: neither small nor big jobs are penalized when using FSPE+PS. This is a desirable result, but the reported results are still averages: in order to ensure that sojourn time is commensurate to size *for all jobs*, we need to investigate the *per-job* slowdown distribution.
In Figure 3.8 on the preceding page, we plot the CDF of per-job slowdown for our default simulator parameters. By serving efficiently smaller jobs, all size-based scheduling techniques and LAS manage to obtain an optimal slowdown of 1 for the majority of jobs. However, some jobs experience very high slowdown values: jobs with a slowdown larger than 100 are around 1% for FSPE and around 8% for SRPTE.
PS, LAS, and FSPE+PS perform well in terms of fairness, with no jobs experiencing slowdown higher than 100 in our experiment runs.[1] While PS is generally considered the reference for a "fair" scheduler, it obtains slightly better slowdown than LAS and FSPE+PS only for the most extreme cases, while being outperformed for a large majority of the jobs. We remark that slowdown values for PS are clustered around integer values, because they are obtained in the common case where a small job is submitted when n larger ones are running.

3.3.2 Real Workloads

We now consider two real workloads in order to confirm that the phenomena we observed in our experiments are not an artifact of the synthetic traces that we generated, and that they indeed apply in realistic cases. From the traces we obtain two data points per job: submission time and job size. In this way, we move away from the assumptions of the $G/G/1$ model, and we provide results that can account for more general cases where periodic patterns and correlation between job size and submission times are present.

[1] Figure 3.8 plots the results of 121 experiment runs, representing therefore 1,210,000 jobs in this simulation.

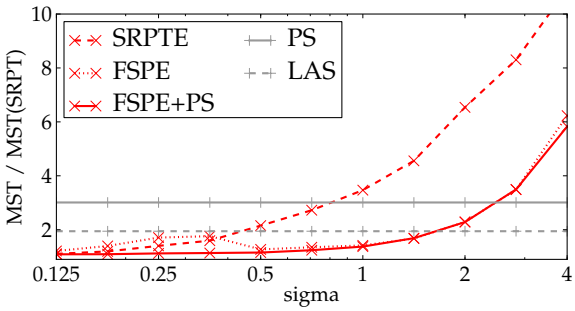


Figure 3.10: MST of the Facebook workload.

Hadoop at Facebook

We consider a trace from a Facebook Hadoop cluster in 2010, covering one day of job submissions. The trace has been collected and analized by Chen *et al.* [88]; it is comprised of 24,443 jobs and it is available online.[2] For the purposes of this work, we consider the job size as the number of bytes handled by each job (summing input, intermediate output and final output): the mean size is 76.1 GiB, and the largest job processes 85.2 TiB. To understand the shape of the tail for the job size distribution, in Figure 3.9 on the preceding page we plot the complementary CDF (CCDF) of job sizes (normalized against the mean); the distribution is heavy-tailed and the largest jobs are around 3 orders of magnitude larger than the average size. For homogeneity with the results of Section 3.3.1, we set the processing speed of the simulated system (in bytes per second) in order to obtain a load (total size of the submitted jobs divided by total length of the submission schedule) of 0.9.

In Figure 3.10, we show MST, normalized against optimal MST, while varying the error rate. We remark that these results are very similar to those that we observe from Figure 3.4 on page 47: also in this case, FSPE and FSPE+PS perform well even when job size estimation errors are far from negligible. These results show that this workload is well represented by our synthetic workloads, when shape is around 0.25.

We performed more experiments on these traces; extensive results are available in a technical report [82].

Web Cache

IRCache (ircache.net) is a research project for web caching; traces from the caches are freely available. We performed our experiments on a one-day trace of a server from 2007 totaling 206,914 requests;[3] the mean request size in the traces is 14.6KiB, while the maximum request size is 174 MiB. In Figure 3.9 on the facing page we show the CCDF of job size; as compared to the Facebook trace analyzed previously, the workload is more

[2] https://github.com/SWIMProjectUCB/SWIM/blob/master/workloadSuite/FB-2010_samples_24_times_1hr_0.tsv

[3] ftp://ftp.ircache.net/Traces/DITL-2007-01-09/pa.sanitized-access.20070109.gz.

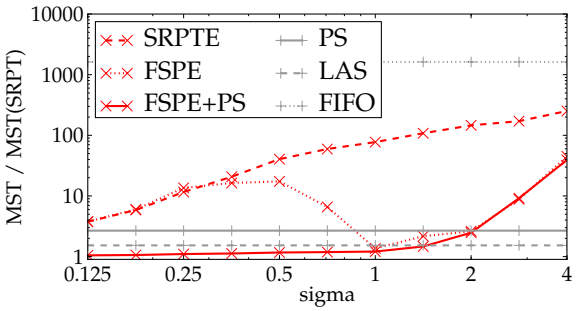


Figure 3.11: MST of the IRCache workload.

heavily tailed: the biggest requests are four orders of magnitude larger than the mean. As before, we set the simulated system processing speed in bytes per second to obtain a load of 0.9.

In Figure 3.11 we plot the evolution of MST as the sigma parameter controlling error grows. Since the job size distribution is more heavily tailed, sojourn times are more influenced by job size estimation errors (notice the logarithmic scale on the y axis), confirming the results we have from Figure 3.2 on page 45. The performance of FSPE does not worsen monotonically as error grows, but rather becomes better for $0.5 < \text{sigma} < 1$; this is a phenomenon that we also observe – albeit to a lesser extent – for synthetic workloads in Figure 3.4b on page 47 and for the Facebook workload in Figure 3.10 on the preceding page. The explanation that we provided in Section 3.3.1 applies: since the mean of the log-normal distribution grows as sigma grows, the aggregate amount of work for a given set of jobs is likely to be over-estimated in total, reducing the likelihood that several jobs at once become late and therefore non-preemptable. Also in this case, we still remark that FSPE+PS consistently outperforms FSPE. Once again, the results for the slowdown distribution are qualitatively analogous to those reported in Section 3.3.1.

3.4 Summary

In this Chapter we presented a set of size-based scheduling policies for single server queue that work on estimated sizes, namely SRPTE and FSPE, and we showed that they can deal with estimated sizes when the workload is not too skewed and errors on estimated sizes are not extreme.

We then presented FSPE+PS, a size-based scheduling policy that is even more tolerant to estimation errors and provides close to optimal response times and good fairness in all but the most extreme of cases.

We stress that, as we are going to see in the next chapter, DISC systems such as Hadoop do not present the extreme conditions that require FSPE+PS; both SRPTE and FSPE are good candidates to be implemented in a DISC system. We chose FSPE to avoid problems related to job starvation and for the fair properties that it provides.

In the next Chapter we are going to present HFSP, an implementation of the FSPE scheduling policy for a DISC system such as Hadoop. The problems raised by a complex system such as Hadoop are many, but the simple nature of FSPE makes easy to adapt it to such a system without losing its properties.

Chapter 4

The Hadoop Fair Sojourn Protocol

In this Chapter we present and analyze the *Hadoop Fair Sojourn Protocol* scheduler (HFSP), which is an adaptation of the FSPE scheduling policy to Hadoop, a data intensive scalable computing system based on Google MapReduce [1]. The main difference between FSPE, presented in the previous Chapter, and HFSP is that while FSPE is an abstract scheduling policy for a single server queue, HFSP is a fully fledged implementation for a distributed system that must take into account the complexity of a real system.
In this Chapter we present the problems that we have encountered in extending FSPE to HFSP and how we solved them. Implementing HFSP raises a number of challenges: a few of them come from MapReduce itself – *e.g.*, the fact that a job is composed by tasks – while others come from the size-based nature of the scheduler in a context where the size of the jobs is not known *a-priori*. In this section we describe each challenge and the proposed corresponding solution.
The Chapter is organized as follow: we first introduce the concept of job size in MapReduce (Section 4.1), we then describe the estimation module that estimate job sizes (Section 4.2), after it we present the aging module (Section 4.3) and finally how estimation and aging module are composed to create the HFSP scheduler (Section 4.4).

4.1 Jobs

In MapReduce, jobs are scheduled at the granularity of *tasks*, and they consist of two separate phases, called Map and Reduce. The Reduce phase can run only after the Map phase is completed, *i.e.* when all Map tasks are done. Our scheduler splits logically the job in the two phases and treats them independently; therefore the scheduler considers the job as composed by two parts with different sizes, one for the Map and the other for the Reduce phase. When a resource is available for scheduling a Map (resp. Reduce) task, the scheduler sorts jobs based on their virtual Map (resp. Reduce) sizes, and grants the resource to the job with the smallest size for that phase.

The size of each phase, to which we will refer as *real size*, is unknown until the phase itself is complete. Our scheduler, therefore, works using an *estimated size*: starting from this estimate, the scheduler applies job aging, *i.e.*, it computes the *virtual size*, based on the time spent by the job in the waiting queue. The estimated and the virtual sizes are calculated by two different modules: the *estimation module*, that outputs a phase *estimated size*, and the *aging module*, that takes in input the estimated size and applies an aging function.

4.2 The Estimation Module

The role of the estimation module is to assign a size to a job phase such that, given two jobs, the scheduler can discriminate the smallest one for that phase. When a new job is submitted, the module assigns for each phase an *initial size* S_i, which is based on the number of its tasks. The initial size is necessary to quickly infer job priorities. A more accurate estimate is done immediately after the job submission, through a *training stage*: in such a stage, a subset of t tasks, called the *training tasks*, is executed, and their execution time is used to update S_i to a *final estimated size* S_f. Choosing t induces the following trade-off: a small value reduces the time spent in the training stage, at the expense of inaccurate estimates; a large value increases the estimation accuracy, but results in a longer training stage. As we will show in Section 4.4, our scheduler is designed to work with rough estimates, therefore a small t is sufficient for obtaining good performances.

4.2.1 Tiny Jobs

Every job phase with less than t tasks is considered as *tiny*: in this case, the scheduler sets $S_f = 0$, which grants them the highest priority. Indeed, tiny jobs use a negligible fraction of cluster resources: giving them the highest priority marginally affects other jobs. Note that the virtual size of all other jobs is constantly updated, therefore every job will be eventually scheduled, even if tiny jobs are constantly submitted.

4.2.2 Initial Size

The initial size of a job phase with n tasks is set to $S_i = n \cdot \xi \cdot \overline{s}$ where $\overline{s}$ is the average task size computed so far by the system considering the jobs that have already completed; $\xi \in [1, \infty)$ is a tunable parameter that represents the propensity of the system to schedule jobs that have not completed the training stage yet. If $\xi = 1$, new jobs are scheduled quite aggressively based on the initial estimate, with the possible drawback of scheduling a particularly large job too early. Setting $\xi > 1$ mitigates this problem, but might result in increased response times. Finally, if the cluster does not have a past job execution

history, the scheduler sets $S_i = s_0$, where $s_0 > 0$ is an arbitrary constant, until the first job completes.

4.2.3 Final Size

When a job phase completes its training stage, the estimation module notifies the aging module that it is ready to update the size of that phase for that job.
The final size is computed as:

$$S_f = \tilde{s} \cdot [(n-t) + \sum_{k=1}^{t}(1-p_k)],$$

where $(n-t)$ is the number of tasks of a job phase that still need to be scheduled, that is the total number of tasks minus the training tasks. The definitions of $\tilde{s}$ and p_k are more subtle. As observed in prior works [33, 26], MAP task execution times are generally stable and short, whereas the distribution of REDUCE task execution times is skewed. For this reason, $\tilde{s}$ represents the average size of the t tasks that either *i)* completed in the training stage (and thus have an individual execution time s_k), or *ii)* that made enough progress toward their completion, which is determined by a timeout Δ (60 s in our experiments). The progress term p_k, which is measured by Hadoop *counters*, indicates the percentage of input records processed by a training task t_k. More formally, we thus have that:

$$\tilde{s} = \frac{1}{t}\sum_{k=1}^{t}\tilde{s}_k,$$

where

$$\tilde{s}_k = \begin{cases} s_k & \text{, if training task } t_k \text{ completes within } \Delta \\ \frac{\Delta}{p_k} & \text{, otherwise} \end{cases}.$$

In HFSP, once S_f is set for the first time, after the training stage, it will not be updated, despite additional information on task execution could be used to refine the first estimate. Indeed, continuous updates to S_f may result in a problem that we call "flapping", which leads to poor scheduling performance: Figure 4.1 shows an example of this effect. If S_f estimates are not updated, there are two possible cases: the first estimate leads to *i)* a correct scheduling decision, or *ii)* an "inversion" between two jobs.
In Figure 4.1, a correct scheduling decision implies that J_2 completes at time 2 and J_1 completes at time 5, while an inversion implies that J_1 completes at time 3 and J_2 completes at time 5. The mean *response time* in the first and second case are 3.5 and 4 respectively. The bottom scheme in Figure 4.1 illustrates the effects of a continuous update to S_f, which leads to flapping. In this case, the response times are 5 for J_1 and 4 for J_2 resulting in a mean response time of 4.5.

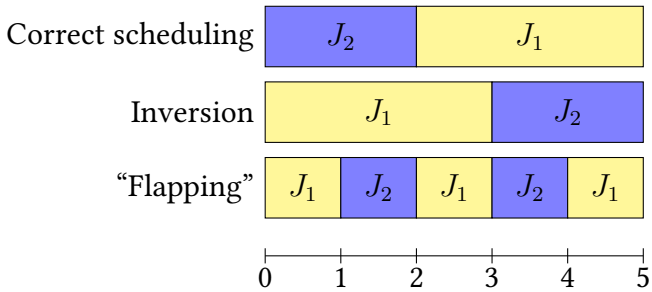


Figure 4.1: Example of continuous updates of job sizes.

Clearly, jobs with similar sizes exacerbate the phenomenon we illustrate above. Since estimation errors for jobs with similar sizes is likely to occur, as we are going to show in Section 4.5, HFSP avoids "flapping" problems using a unique estimate for S_f.

4.3 The Aging Module

The aging module takes in input the estimated sizes to compute *virtual sizes*. The use of virtual sizes is a technique applied in many practical implementations of well-known schedulers [55, 38, 60]: it consists in keeping track of the amount of the remaining work for each job phase in a *virtual "fair" system*, and update it every time the scheduler is called. The result is that, even if the job doesn't receive resources and thus its real size does not decrease, in the virtual system the job virtual size slowly decreases with time. Job aging avoids starvation, achieves fairness, and requires minimal computational load, since the virtual size does not incur in costly updates [55, 38].
Figure 4.2 shows an example of how the job virtual size is decreased and how that affects the scheduling policy. We recall that HFSP schedules jobs with the smallest virtual size first. In Figure 4.2a, we show job virtual sizes and in Figure 4.2b the job real sizes. At time 0 there are two jobs in the queue, Job 1 with size 3 and Job 2 with size 4. Note that at the beginning, the virtual size corresponds to the real job size. The scheduling policy chooses the job with the smallest virtual size, that is Job 1, and grants it all the resources. At time 3, Job 3 enters the job queue and Job 1 completes. Job 3 has a smaller real size than Job 2 but a bigger virtual size; this is due to the fact that while Job 1 was working, the virtual times of both Job 1 and Job 2 have been decreased. When Job 1 finishes, Job 2 has virtual size 2.5 while Job 3 has virtual size equal to its current real size, that is 3.

4.3.1 Virtual Cluster

The part of the aging module that implements the virtual system to simulate processor sharing is called *Virtual Cluster*. Jobs are scheduled at the granularity of tasks, thus the virtual cluster simulates the same resources available in the real cluster: it has the same number of "machines" and the same configuration of (Map or Reduce) resources per

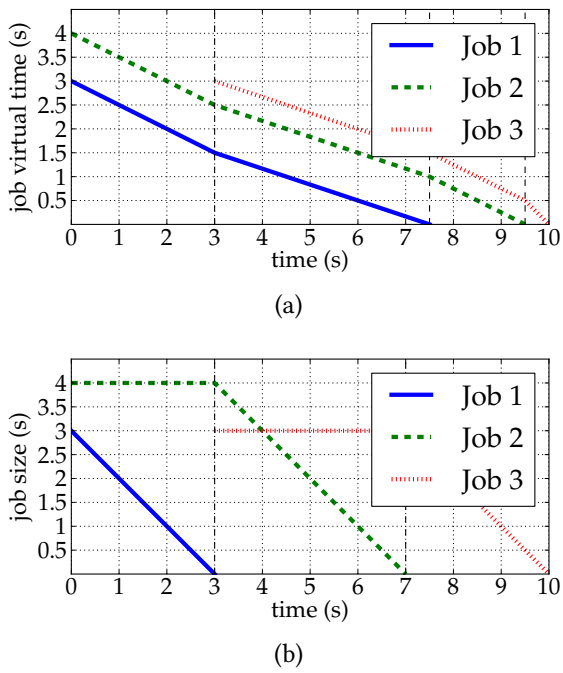


Figure 4.2: An example of virtual time and job size progress in HFSP.

machine. When the resources change in the real cluster, the scheduler notifies it to the aging module that updates the virtual cluster resources. We simulate a *Max-Min Fairness* criterion to take into account jobs that require *less* compute resources than their fair share (*i.e.*, $1/n$-th of the resources if there are n active jobs): a round-robin mechanism allocates virtual cluster resources, starting from small jobs (in terms of the number of tasks). As such, small jobs are implicitly given priority, which reinforces the idea of scheduling small jobs as soon as possible.

4.3.2 Aging Speed

The fact that the virtual cluster has the same resources of the real one and the virtual scheduler assigns (virtual) resources to tasks has an interesting effect on the aging speed of jobs: a job with more tasks than another can age faster if enough resources are free in the virtual cluster. This models the core idea of MapReduce that jobs with a *higher parallelism degree*, *i.e.* more tasks, can do more work than others with smaller parallelism.

4.3.3 Estimated Size Update

When the estimated size of a job phase is updated from the initial estimation S_i to the final estimation S_f, the estimation module alerts the aging module to update the job phase size to the new value. After the update, the aging module runs the scheduler on the virtual cluster to reassign the virtual resources.

4.3.4 Failures

The aging module is robust with respect to failures. The same technique used to support cluster size updates is used to update the resources available when a failure occurs; once Hadoop detects the failure, job aging will be slower. Conversely, adding nodes will result in faster job aging reflecting the fact that with more resources the cluster can do more work.

4.3.5 Manual Priority and QoS

Our scheduler does not currently implement the concept of different job priorities assigned by the user who submitted the job; however, the aging module can be easily modified to simulate a Generalized Processor Sharing discipline, leading to a scheduling policy analogous to Weighted Fair Queuing [56]. A simple approach is to consider the user assigned priority as a *speed modifier* to the aging of the job phases virtual sizes: when the aging module decreases the virtual size of the job, it subtracts to the job virtual size the virtual work done multiplied by the job modifier. A job modifier bigger (resp. smaller) than 1 speeds up (resp. slows down) the aging of a job.

4.4 The Scheduling Policy

In this section we describe how the estimation and the aging modules coexist to create a task scheduler[1] for Hadoop that strives to be both efficient and fair.

4.4.1 Job Submission

Figure 4.3 shows the lifetime of a job in HFSP, from its submission to its completion and removal from the job queue. When a job is submitted, for each phase of the job, the scheduler asks to the estimation module if that phase is tiny. If the answer is affirmative, the scheduler assigns $S_f = 0$, meaning that the job must be scheduled as soon as possible. Otherwise, the scheduler starts the training stage and sets the virtual time to the initial size S_i given by the estimator module. Periodically, the scheduler asks to the estimation module if it has completed its training stage, and, if the answer is positive, it notifies the aging module to update the virtual size of that job and removes the job from the training stage.

4.4.2 Priority To The Training Stage

The training stage is important because, as discussed in Section 4.2, the initial size S_i is imprecise, compared to the final size S_f. Completing the training stage as soon as

[1] In Hadoop, jobs are scheduled at granularity of tasks, and thus the scheduler is called Task Scheduler

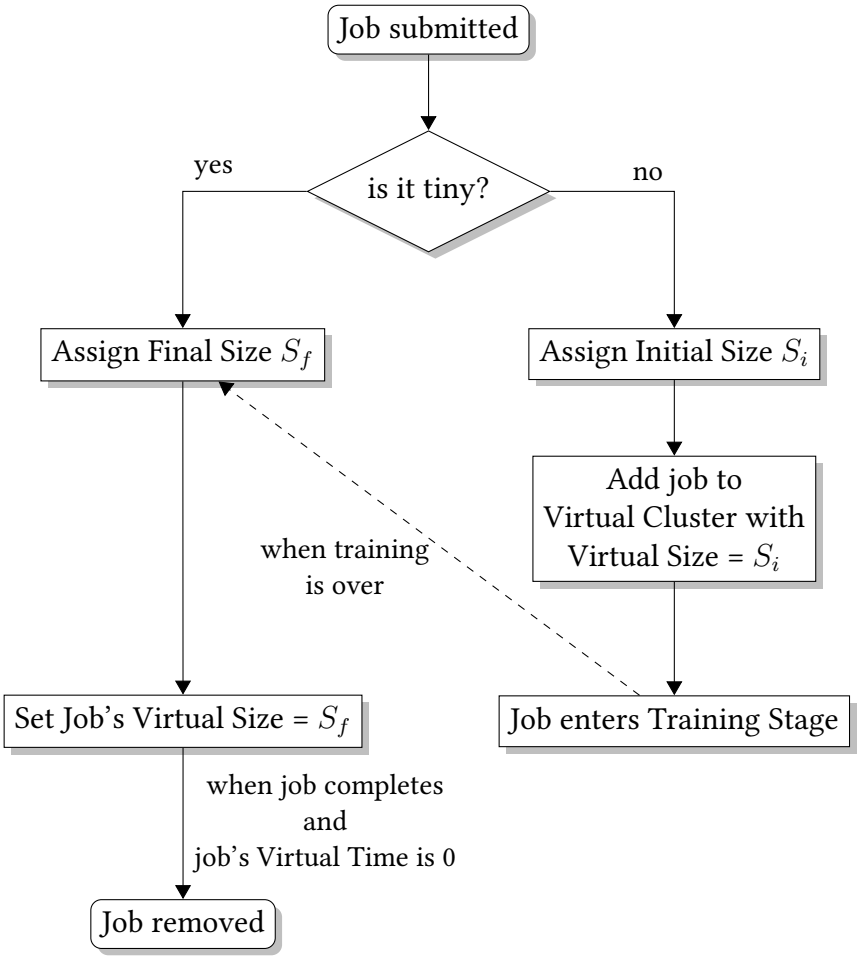


Figure 4.3: Job's lifetime in HFSP

possible is fundamental for an efficient scheduling policy. There are two strategies that are used by the scheduler to speed up the training stage: the first strategy is to set a low number of training tasks t, as discussed in Section 4.2; the second strategy is to give priority to the training tasks across jobs – up to a threshold of $T \in [0, T_{max}]$ where T_{max} is the total number of resources in the cluster. Such threshold avoids starvation of "regular" jobs in case of a bursty job arrival pattern. When a resource is free and there are jobs in the training stage, the scheduler assigns the resource to a training task independently from its job position in the job queue. In other words, training tasks have the highest priority. Conversely, after a job has received enough resources for its training tasks, it can still obtain resources by competing with other jobs in the queue.

4.4.3 Virtual Time Update

When a job phase completes its training stage, the scheduler asks to the estimation module the final size S_f and notifies the aging module to update the virtual size accordingly. This operation can potentially change the order of the job execution. The scheduler should consider the new priority and grant resources to that job, if such job is the smallest one in the queue. Unfortunately, in Hadoop MapReduce the procedure to free resources that are used by the tasks, also known as *task preemption*, can waste work. The default strategy used by HFSP is to wait for the resources to be released by the working tasks. Section 4.6 describes the preemption strategies implemented in HFSP and their implications.

4.4.4 Data locality

For performance reasons, it is important to make sure that MAP tasks work on local data. To this aim, we use the *delay scheduling* strategy [33], which postpones scheduling tasks operating on non-local data for a fixed amount of attempts; in those cases, tasks of jobs with lower priority are scheduled instead.

4.4.5 Scheduling Algorithm

HFSP scheduling – which is invoked every time a MapReduce slave claims work to do to the MapReduce master – behaves as described by Algorithm 3. The procedure `AssignPhaseTasks` is responsible for assigning tasks for a certain phase. First, it checks if there are jobs in training stage for that phase. If there are any, and the number of current resources used for training tasks T_{curr} is smaller or equal than T, the scheduler assigns the resource to the first training task of the smallest job. Otherwise, the scheduler assigns the resource to the job with the smallest virtual time. When a task finishes its work, the procedure `releaseResource` is called. If the task is a training task, then the number T_{curr} of training slots in use is decreased by one.

```
fun assignPhaseTasks(resources):
    foreach resource s ∈ resources:
        if ∃ (Job in training stage) and T_curr < T:
            job ← select job to train with smallest initial virtual size
            assign(s, job)
            T_curr ← T_curr + 1
        else:
            job ← select job with smallest virtual time
            assign(s, job)

fun assign(resource, job):
    task ← select task with lower ID from job
    assign task to resource

fun releaseResource(task):
    if task is a training task:
        T_curr ← T_curr − 1
```

Algorithm 3: HFSP resource scheduling for a job phase.

4.5 Impact of Estimation Errors

In this section we describe what is the impact of an erroneous scheduling, *i.e.* giving resources to the wrong job, and how HFSP handles estimation errors.
The output of the estimation module is an estimated size, thus it can contain an error e. We identify e as the ratio between the estimated size[2] S and the real size R, therefore $S = R \cdot e$.
We recall from Chapter 3 that two kinds of errors are possible: the job size can be under-estimated ($e < 1$) or over-estimated ($e > 1$). Both errors have an impact on the scheduler; however, the case of under-estimation is more problematic. Indeed, a job whose size is over-estimated obtains a lower priority: this impacts only that job, delaying the moment in which it obtains the resources to work. Even if the job enters the queue with a lower priority, *i.e.* a large virtual size, the aging function will raise its priority as the time goes by, and the job will obtain eventually the resources. Instead, job under-estimation increases the priority of a job, and this can potentially affect all the other jobs in the queue. The aging function plays a crucial role in making HFSP tolerant to estimations errors.
The study of the estimation error for a single job is not enough to understand the impact of the errors on our scheduling policy. Indeed, we need to consider the estimation errors of all the jobs in the queue and how their interplay can ultimately lead the scheduler to

[2]For the sake of clearness, in this Section we simplify the notation: S refers to S_f, as described previously.

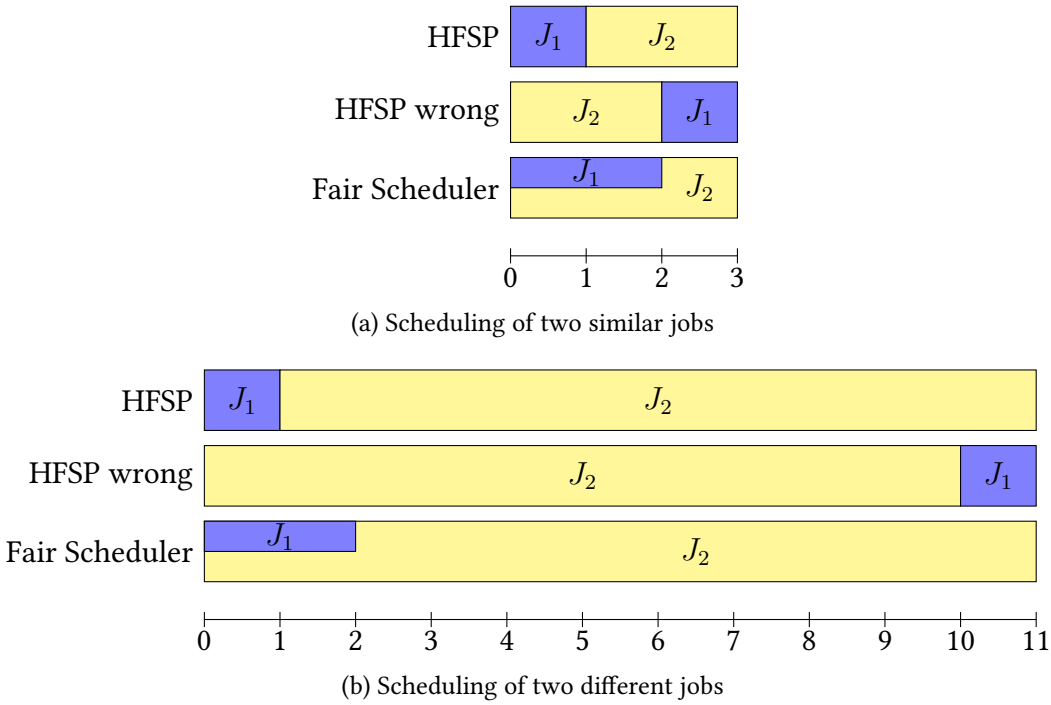


Figure 4.4: Illustrative examples of the impact of estimation errors on HFSP.

take wrong scheduling decisions. We exemplify this problem for two jobs: let us denote the size of an arbitrary phase of a job J_k as R_k, and its estimated size as $S_k = R_k \cdot e_k$, where e_k expresses the estimation error. Two jobs J_1 and J_2 with $R_1 < R_2$ are scheduled incorrectly (*i.e.* J_2 is given priority over J_1) if $S_1 > S_2$, *i.e.* $e_1/e_2 > S_2/S_1$. If J_1 and J_2 are similar in size, even a small e_1/e_2 ratio may invert the priorities of these jobs; if S_2/S_1 is instead large (*e.g.* because the two sizes differ by orders of magnitude), then also e_1/e_2 must be analogously large. This works perfectly with a size-based scheduler because if two jobs are similar, inverting the priorities of the two has only a small impact on the overall performance. Instead, if two jobs are very different, switching them can lead to poor performance because the job with highest priority has to wait for an amount of time that may be much larger than its size. This, however, is unlikely because it requires that e_1 and/or e_2 are very far from 1. In other words, estimation errors should be very large for switching very different jobs.

Figure 4.4a and Figure 4.4b exemplify how a wrong scheduling can affect HFSP. Each figure has three cases: the first case, on top, is the correct HFSP policy; the second case, in the middle, shows what happens when HFSP gives priority to the wrong job and the third case, at the bottom, shows an ideal "fair" scheduler adopting a processor-sharing scheduling policy.

Figure 4.4a shows what happens when the two jobs have similar sizes, that are 1 and 2 seconds. If the estimation module outputs estimated sizes S_1 and S_2 such that $S_1 < S_2$, HFSP schedules first J_1 and then J_2 and the resulting mean response time is $\frac{1+3}{2} = 2$. On the contrary, if the estimation is wrong and $S_1 > S_2$, HFSP will schedule first J_2 and

then J_1. While this scheduling choice is incorrect, the resulting mean response time is $\frac{2+3}{2} = 2.5$, which is more than the one with the correct choice. While a response time of 2.5 is not optimal, matches the mean response time obtained with Processor Sharing. We conclude that if two jobs are similar, switching them by giving priority to the bigger one does not affect heavily a metric such as the mean response time.
Figure 4.4b illustrates what happens when two jobs have sizes very different, in the example 1 and 10 seconds. As in the previous example, HFSP schedules J_1 and J_2 based on the estimator output. Here, however, the difference in the mean response time is large: the mean response time is $\frac{1+11}{2} = 6$ when J_1 is scheduled first and $\frac{10+11}{2} = 10.5$ in the other case. The wrong scheduling choice does not only lead to an almost doubled mean response time: it also heavily penalizes J_1, which has to wait 10 times its size before having any resource granted. The difference between the two scheduling choices is even more clear when they are compared to the Processor Sharing, that has a mean response time of $\frac{2+11}{2} = 6.5$. When two jobs are very different, a wrong scheduling choice can lead to very poor performance of HFSP. This situation, however, is much less likely than the former, requiring a value of $e_1 > 10e_2$.
In summary, scheduling errors – when caused by estimations that are not very far from the real job sizes – result in response times that are similar to those obtained by processor sharing policies, especially under heavy cluster load. Estimation errors lead to very bad scheduling decisions only when job sizes are very different from their estimations. As we show in our experimental analysis, such errors rarely occur with realistic workloads.

4.6 Task Preemption

HFSP is a preemptive scheduler: jobs with higher priority should be granted the resources allocated to jobs with lower priorities. In Hadoop, the main technique to implement preemption is by *killing* tasks. Clearly, this strategy is not optimal, because it wastes work, including CPU and I/O. Other works have focused on mitigating the impact of KILL on other MapReduce schedulers [29]. Alternatively, it is possible to WAIT for a running task to complete, as done by Zaharia *et al.* [33]. If the runtime of the task is small, then the waiting time is limited, which makes WAIT appealing. This is generally the case of MAP tasks but not of REDUCE tasks, which can potentially run for a long time. HFSP supports both KILL and WAIT and by default it is configured to use WAIT. In this section we describe how HFSP works when the KILL preemption is enabled.

4.6.1 Task Selection

Preempting a job in MapReduce means preempting some or all of its running tasks. It may happen that not all the tasks of a job have to be preempted. In this case, it is very important to pick the right tasks to preempt to minimize the impact of KILL. HFSP chooses to preempt the "*youngest*" tasks, *i.e.* those that have been launched last, for three

practical reasons: *i)* old tasks are the most likely ones to finish first, freeing resources to other tasks; *ii)* killing young tasks wastes less work; *iii)* young tasks are likely to have smaller memory footprints, resulting in lower cleanup overheads due to *e.g.* purging temporary data.

4.6.2 When Preemption Occurs

Preemption may occur for many different reasons. First, the training tasks always have priority over non-training tasks. Therefore, training tasks can preempt other tasks even if they are part of a job with higher priority than theirs. This measure makes the training phase faster to complete and, considering that the number of training tasks is bounded, does not significantly affect the runtime of other jobs. Tasks that complete in the training stage lose their "status" of training task and, consequently, can be preempted as well.
Newly submitted jobs can, of course, preempt running tasks when the virtual size of the new job is smaller than the virtual size of the job that owns the running tasks.
Task preemption may also happen when the estimation module updates the size of a job, from S_i to S_f. This can move the job to a new position in the job queue.
The aging function can also be responsible for task preemption: as we saw in Section 4.3, since we implement max-min fairness, jobs may have different degrees of parallelism, *e.g.* small jobs may require less than their fair share; this results in a different aging, that may change the order in the execution list.
Finally, the last reason for task preemption is when the cluster resources shrink, *e.g.* after a failure. In this case, HFSP grants "lost" resources to jobs with higher priority using preemption.

4.7 Summary

In this Chapter we have presented the Hadoop Fair Sojourn Protocol, a preemptive size-based scheduler for Hadoop MapReduce that is both efficient and fair. Exploring the performance of a scheduler like HFSP can be challenging; the next Chapter is dedicated to the experimental campaign used to evaluate HFSP performance in both a simulative and an experimental way.

Chapter 5

System Evaluation

The evaluation of a scheduler like the Hadoop Fair Sojourn Protocol is a difficult task due to the complexity of a concrete implementation of an advanced scheduler such as HFSP.
This Chapter provides an extensive evaluation of the performance of HFSP, which is compared to that of the default scheduler available in most of the current Hadoop distributions (*e.g.* Cloudera), namely the Fair scheduler [33, 19]. We omit from this section a comparative analysis of HFSP and the FIFO scheduler available in stock Hadoop: in our experiments, FIFO is drastically outperformed by all other schedulers, a rather expected result.
The first part of this Chapter is dedicated to the cluster configuration: in particular, Section 5.1 describes the *BigFoot* platform, on which we run the experiments, and Section 5.2 provides details on the Hadoop configuration.
The rest of the Chapter is organized as follows: Sections 5.3 is dedicated to the mean response time and the slowdown analysis; Section 5.4 provides a study of the cluster utilization and of the job size estimation error; the same section shows a comparative analysis between HFSP with the different preemption primitives available for Hadoop (Kill and Wait).

5.1 The BigFoot Platform

The platform on which we run the experiments is the *BigFoot cluster*.[1] In this section we describe the cluster itself, from the machines that compose the cluster to the software installed.
The BigFoot cluster uses a heterogeneous set of physical machines: we have two *master* nodes running on a dual quad-core Xeon L5320 server clocked at 1.86GHz, with 16GB of RAM, two 1TB hardware RAID5 volumes, and two 1Gbps network interfaces; *worker* nodes execute on six dual exa-core Xeon E5-2650L (with hyperthreading enabled) servers

[1] https://www.bigfoot-project.eu/

clocked at 1.8GHz, with 128GB of RAM, ten 1TB disks (configured as JBOD, Just a Bunch of Disks) and four 1Gb/s network cards. In this work we use a single network interface per host, as splitting traffic or using bonding would have added complexity to the system that, instead, we strive to keep as simple as possible, since in this work we are interested only in baseline performance.
The hardware and network configuration closely resembles the one suggested by commercial private cloud providers, such as Rackspace[109]. In particular storage is distributed on the master and compute hosts and is not concentrated on a separate storage network.
Each machine in the cluster runs the same Linux distribution, a Ubuntu 12.04.2 LTS, updated with the most recent patches. All energy saving settings in the BIOS are disabled, since they cause severe performance penalties. We use the KVM hypervisor, with `virtio` and `vhost_net` acceleration modules enabled. Virtualization support in the CPUs is enabled (VMX) and KVM uses it automatically. The hypervisor is configured by Nova, a component of OpenStack used to manage customized clouds systems[2], to use LVM for VM storage. VMs use the unmodified Ubuntu 13.10 image from the Ubuntu Cloud archives.
We use the *Grizzly* release of OpenStack, which is installed via the Ubuntu cloud repository. One of the master nodes runs the OpenStack management services: the web-based dashboard console, `cinder`, `glance`, `keystone`, and `quantum` (including the server, layer 3 services, OpenVSwitch and DHCP agents).[3]
Worker nodes are configured as compute-only nodes, and they host all the VMs created by our tenants and users. Currently, we configure `quantum` to use GRE tunnels over a physical network that interconnects all nodes of our cluster.
We implemented the most common setup, where `quantum` is configured to use the OpenVSwitch [110] (OVS) plugin to provide connectivity between VMs. OVS is a software switch implementation that materializes as a virtual switch spanning across multiple physical hosts. In our configuration, `quantum` creates a single OVS switch for all VMs, using VLAN tagging to separate traffic from different tenants.
To provide connectivity between tenants and the external network, our virtual network is configured according to the *provider router with private networks* use-case described in the OpenStack documentation [114]. Thus, each tenant has its own IP subnet, and exchange traffic between each other and the Internet using a single virtual router connected to the subnets of each tenant from one side and to the external network on the other side. The `quantum` virtual router is implemented as network namespace on the master node, where a number of NAT and routing rules provide interconnection, external access and floating IPs allocated to the VMs.

[2] http://docs.openstack.org/developer/nova/
[3] For more informations on what each component does please refer to http://openstack.org

These settings are the result of a tedious trial and error process that lasted several months. The OpenStack installation manuals only cover the basics to setup a system that is (mostly) operational, but far from being optimized for performance. Other parameters are buried in bug reports, OpenStack *blueprints* (informal feature proposals to the community) and mailing list archives.

5.2 Experimental Setup

We run our experiments on a cluster composed of 20 TaskTracker worker machines with 4 CPUs and 8 GB of RAM each. We configure Hadoop according to current best practices [19, 22]: the HDFS block size is 128 MB, with replication factor 3; each TaskTracker has 2 map slots with 1 GB of RAM each and 1 reduce slots with 2 GB of RAM. The `slowstart` factor is configured to start the Reduce phase for a job when 80% of its Map tasks are completed.
HFSP operates with the following parameters: the number of tasks for the training stage is set to $t = 5$ tasks; the timeout to estimate task size is set to $\Delta = 60$ seconds; we schedule aggressively jobs that are in the training stage, setting $\xi = 1$ and $T = 10$ slots for both Map and Reduce phases. The Fair scheduler has been configured with a single job pool, thus all jobs have the same priority.
We generate workloads using PigMix [6], a benchmarking suite used to test the performance of Apache Pig releases. PigMix is appealing to us because, much like its standard counterparts for traditional DB systems such as TPC [7], it *both* generates realistic datasets and defines queries inspired by real-world data analysis tasks. PigMix contains both Pig scripts and native MapReduce implementation of the scripts. Since Pig has generally an overhead over the native implementations of the same job, in our experiments we use the native implementations. For illustrative purposes, we include a sample Pig Latin query that is included in our workloads next:

```
REGISTER pigperf.jar;
A = LOAD 'pigmix/page_views' USING ...
    AS (user, action, timespent,
    query_term, ip_addr, timestamp,
    estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'pigmix/power_users_samples' USING ...
    AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name, phone;
C = JOIN B BY user LEFT OUTER,
    beta BY name PARALLEL 20;
```

The above Pig job performs a "projection" on two input datasets, and "joins" the result.

Job arrival follows a Poisson process, and jobs are generated by choosing uniformly at random a query between the 17 defined in PigMix. The amount of work each job has to do depends on the size of the data it operates on. For this reason, we generate four different datasets of sizes respectively 1 GB, 10 GB, 100 GB and 1 TB. For simplicity, we refer to these datasets as to *bins* – see the first two columns of Table 5.1. The third column of the table shows ranges[4] of number of MAP tasks for each bin.
For each job, we randomly map it to one of the four available datasets: this assignment follows three different probability distributions – see the last three columns of Table 5.1. The overall composition of the jobs therefore defines three workloads, which we label[5] according to the following scheme:

- **DEV:** this workload is indicative of a "development" environment, whereby users rapidly submit several small jobs to build their data analysis tasks, together with jobs that operate on larger datasets. The composition of this workload is inspired by the Facebook 2009 trace observed by Chen *et al.* [26]. The mean interval between job arrivals is $\mu = 30\ s$.

- **TEST:** this workload represents a "test" environment, whereby users evaluate and test their data analysis tasks on a rather uniform range of dataset sizes, with 20% of the jobs a large dataset. The composition of this workload is inspired by the Microsoft 2011 traces as described by Appuswamy *et al.* [63] The mean interval between jobs is $\mu = 60\ s$.

- **PROD:** this workload is representative of a "production" environment, whereby data analysis tasks operate predominantly on large datasets. The mean interval between jobs is $\mu = 60\ s$.

In this work, each workload is composed of 100 jobs, and both HFSP and Fair have been evaluated using the same jobs, the same inputs and the same submission schedule. For each workload, we run five experiments using different seeds for the random assignments (query selection and dataset selection), to improve the statistical confidence in our results.
Additional results – not included here for lack of space – obtained on different platforms (Amazon EC2 and the Hadoop Mumak emulator), and with different workloads (synthetic traces generated by SWIM [26]), confirm the ones shown in this paper. They are available in a technical report [57].

[4]Note that PigMix queries operate on different subsets of the input datasets, which result in a variable number of MAP/REDUCE tasks.

[5]Such labeling, although arbitrary, corresponds to our experience in several research and industrial projects we have been involved in, *e.g.* the BigFoot project[113].

Table 5.1: Summary of the workloads used in our experiments.

Bin	Dataset Size	Averag. num. MAP Tasks	Workload		
			DEV	TEST	PROD
1	1 GB	< 5	65%	30%	0%
2	10 GB	$10 - 50$	20%	40%	10%
3	100 GB	$50 - 150$	10%	10%	60%
4	1 TB	> 150	5%	20%	30%

5.3 Macro Benchmarks

In this section we present the aggregate results of our experiments for response times and per-job slowdown, across all schedulers we examine.

5.3.1 Response Time

Figure 5.1 describe the *mean* response times for all workloads we evaluate: the mean response times are indicative of system responsiveness, and lower values are best. Overall, HFSP is substantially more responsive than the Fair scheduler, a claim that we confirm also by inspecting the full distribution of response times, later in this Section. It is important to note that a responsive system does not only cater to a "development" workload, which requires *interactivity*, but also to more "heavy" workloads, that require an *efficient* utilization of resources. Thus, in summary, HFSP is capable of absorbing a wide range of workloads, with no manual (and static) configuration of resource pools, and only a handful parameters to set. Globally, our results can also be interpreted in another key: a system using HFSP can deliver the same responsiveness as one running other schedulers, but with less hardware, or it can accommodate more intensive workloads with the same hardware provisioning.
Next, we delve into a more detailed analysis of response times, across all workloads presented in the global performance overview.
The DEV workload is mostly composed of jobs from bin 1 that are treated in the same way by both HFSP and Fair schedulers: this biases the interpretation of results using only first order statistics. Therefore, in Figure 5.3, we show the EDCF of job response times for all bins except bin 1. We notice that jobs that have a response time less or equal to 80 seconds are 60% in HFSP and only 20% in Fair. The reason of this boost in performance is the fact that HFSP runs jobs in sequence while Fair runs them in parallel. By running jobs in series, HFSP focuses all the resources on one job that finishes as soon as possible without penalizing other jobs, leading to increased performance overall.
The TEST workload is the most problematic for HFSP because the jobs are distributed almost uniformly among the four bins. In Figure 5.4, we decompose our analysis per bin, and show the *empirical cumulative distribution function* (ECDF) of the job response times for all bins except for those in bin 1. For jobs in bin 2, the median response times are of 31

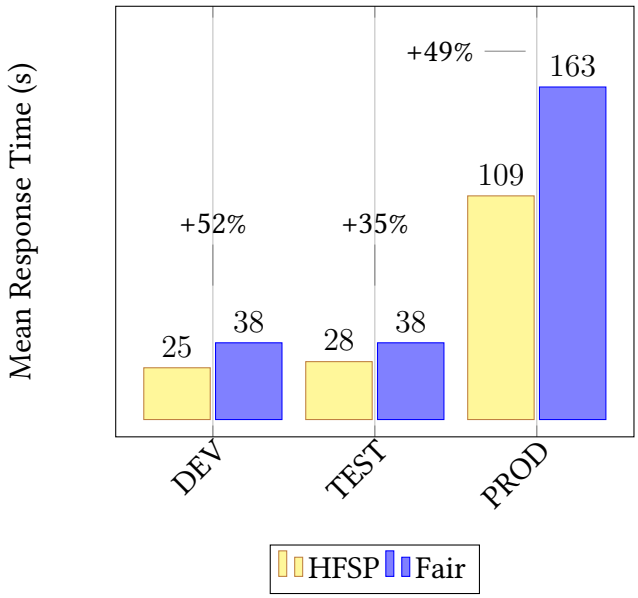


Figure 5.1: Aggregate *mean* response times for all workloads.

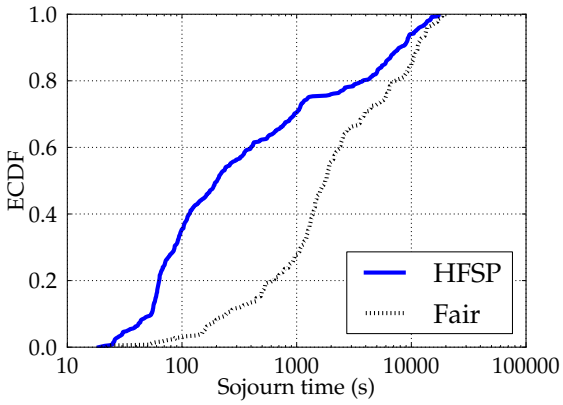


Figure 5.2: ECDF of the response times for the PROD workload.

seconds and 45 seconds for HFSP and Fair, respectively. For jobs in bin 3, median values are more distant: 98 seconds and 290 seconds respectively for HFSP and Fair. Finally, for jobs in bin 4, the gap further widens: 1000 seconds versus almost 2000 seconds for HFSP and Fair, respectively. The submission of jobs from bin 3 and 4 slows down the Fair scheduler, while HFSP performs drastically better because it "focuses" cluster resources to individual jobs.

Our results for the PROD workload are substantially in favor of HFSP, that outperforms the Fair scheduler both when considering the *mean* response times (see Figure 5.1), and the full distribution (see Figure 5.2) of response times: in this latter case, *e.g.* median response times of HFSP are one order of magnitude lower than those in the Fair scheduler.

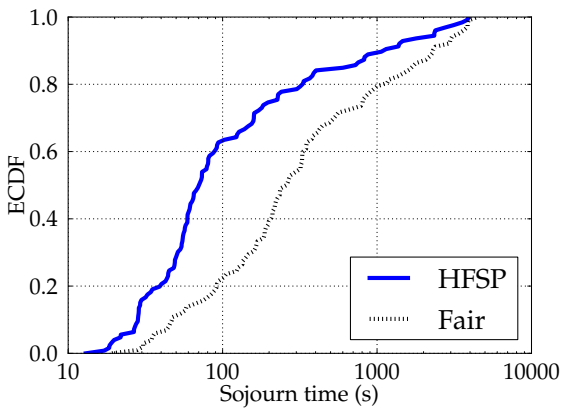
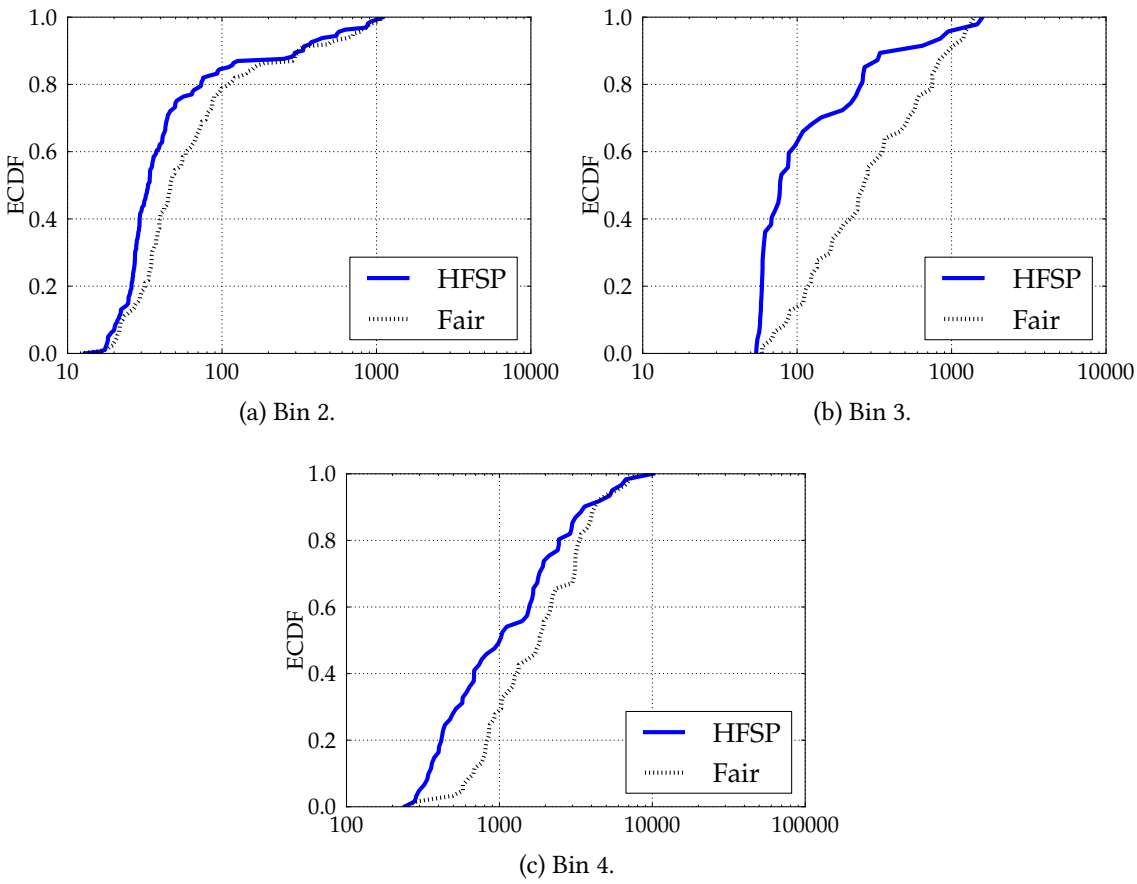


Figure 5.3: ECDF of the response times for the DEV workload, excluding jobs from bin 1.



(a) Bin 2.

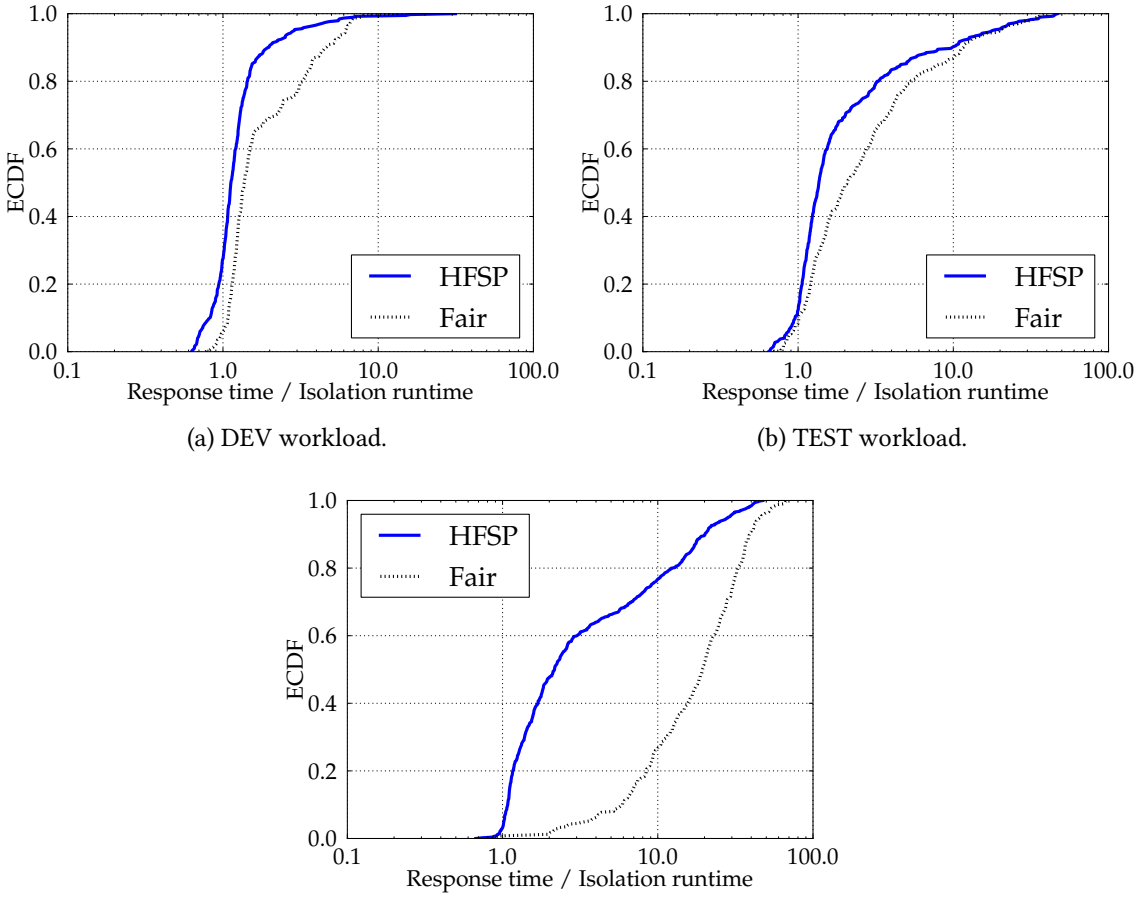
(b) Bin 3.

(c) Bin 4.

Figure 5.4: ECDF of the response times for the TEST workload, grouped per bin.

5.3.2 Slowdown

Figure 5.5 shows the ECDF of the per-job slowdown for the three workloads. Recall that, the slow down of a job equals its response time divided by its size: hence, values close or



(a) DEV workload.

(b) TEST workload.

(c) PROD workload.

Figure 5.5: ECDF of the per-job slowdown, for all workloads.

equal to 1 are best. Thus, we use Figure 5.5 to compare the HFSP and Fair schedulers with respect to the notion of "fairness" we introduced earlier: globally, our results indicate that HFSP is always more fair to jobs than the Fair scheduler.

In particular, we notice that TEST and PROD workloads are particularly difficult for the Fair scheduler. Indeed, a large fraction of jobs is mistreated, in the sense they have to wait long before being served. With the Fair scheduler, job mistreatment worsens when workloads are "heavier"; in contrast, HFSP treats well the vast majority of jobs, and this is true also for demanding workloads such as TEST and PROD. For example, we can use the median slowdown to compare the behavior of the two schedulers: the gap between HFSP and Fair widens from a few units, to one order of magnitude for the PROD workload.

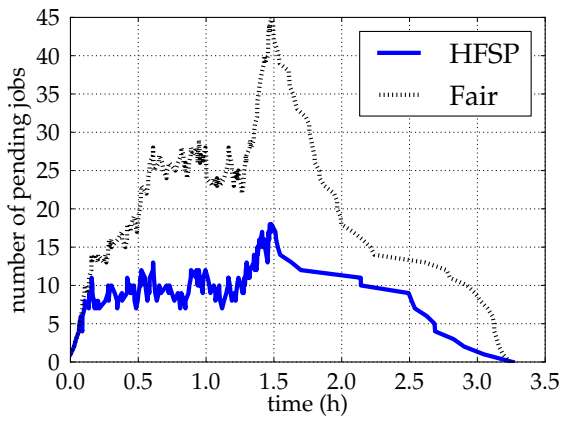


Figure 5.6: Time-series of the cluster load, for an individual experiment with the PROD workload.

5.4 Micro Benchmarks

In this Section we study additional details of HFSP and Fair schedulers, and introduce new results that measure cluster utilization and that allow to assess the extent of job size estimation errors. Finally, we focus on job and task preemption and discuss about the impact on performance of such mechanism.

Cluster load

We now study the implications of job scheduling from the perspective of cluster utilization: when workloads (like the ones we use in our experiments) present bursts of arrivals, it is important to understand the ability of the system to "absorb" such bursts, without overloading the queue of pending jobs. We thus define the *cluster load* as the number of jobs currently in the system, either running or waiting to be granted resources to execute.

To understand how cluster load varies with the two schedulers, we focus on a individual run of the PROD workload, because its high toll in cluster resources. Figure 5.6 illustrates the time-series of the cluster load for both HFSP and Fair scheduler. In the experiment we show, there are two significant bursts: in the first, between 0.4 and 0.6 hours, 18 new jobs arrive; in the second, between 1.34 and 1.5 hours, 35 new jobs are submitted.

Clearly, Figure 5.6 shows that HFSP handles bursts of arrivals better than the Fair scheduler: for the latter, the cluster load soars corresponding to each burst, whereas the HFSP scheduler induces a smoother cluster. Indeed, since HFSP schedules jobs in series, it is able to serve jobs faster – and thus free up resources more quickly – than the Fair scheduler, which instead prolongs jobs service, by granting few resources to all of them.

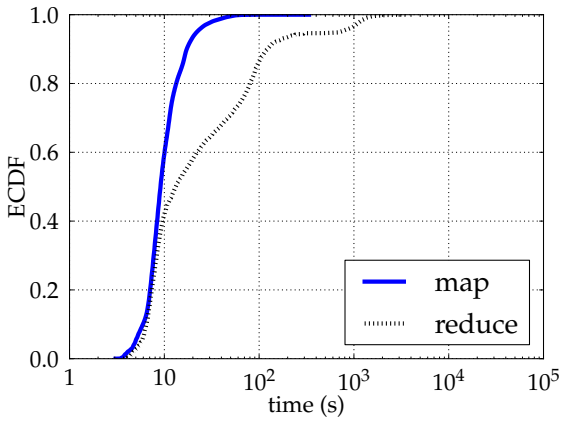


Figure 5.7: ECDF of task run times, for the MAP and REDUCE phases of all jobs across all workloads for HFSP.

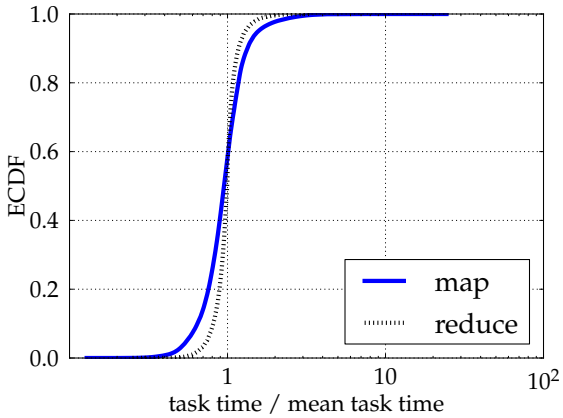


Figure 5.8: ECDF of the normalized task run time, for the MAP and REDUCE phases of all jobs across all workloads for HFSP.

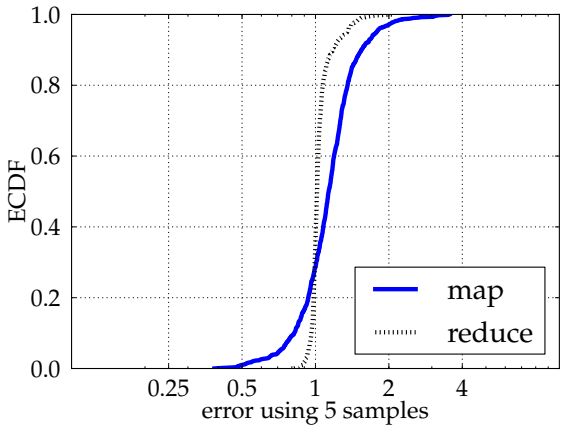
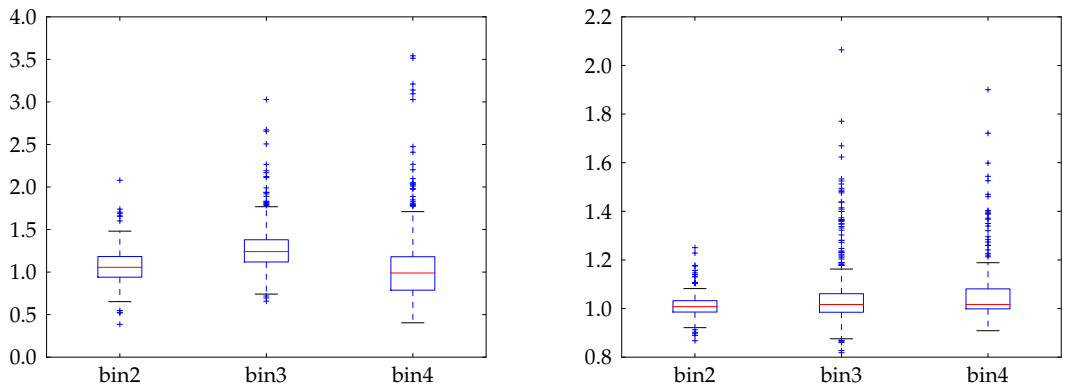


Figure 5.9: ECDF of estimation errors, for the MAP and REDUCE phases of all jobs across all workloads for HFSP.



(a) Map (b) Reduce

Figure 5.10: Most important percentiles and outliers of estimation errors, grouped by bin, for the phases of jobs across all workloads for HFSP.

Task time and error

We now focus solely on HFSP, and analyze a seemingly delicate component thereof: the job size estimation module. As alluded in Section 4.5, our experimental results indicate that HFSP is resilient to job size estimation errors: we show that by focusing on a detailed analysis of *task times* (which determine the size of a job), and the estimation errors induced by the simple estimator we implemented,[6] which only takes a few training tasks to deduce job sizes.

Figure 5.7 shows an aggregate distribution of task times for Map and Reduce phases for all jobs in all workloads and for all our experiments. It is possible to see that most Map tasks complete within 60 seconds, and the variability among different tasks is limited. Instead, Reduce task times variability is extremely high, with differences peaking at 2 orders of magnitude. Given the skew of Reduce task times, it is reasonable to question the accuracy of the simple estimation mechanism we use in HFSP.

A closer look at task time distribution, however, reveals that within a single job, task times are rather stable. Figure 5.8 shows the ECDF of the normalized Map and Reduce task times: this is obtained by grouping task times belonging to the same job together, computing the mean task time for each job, and normalizing task times by the corresponding mean task time of the job they belong to. For instance, if a job has two tasks with task time equal to 10 and 30 seconds respectively, then the mean task time for that job would be 20 seconds, and the normalized task times of these its tasks would be 0.5 and 1.5 seconds respectively. As such, Figure 5.8 indicates that using a small subset of

[6]In practice, HFSP is designed to support a variety of job size estimators, that can be plugged in as simple modules. This can be useful when prior knowledge on a workload composition is available, an assumption we do not make in this work.

tasks to compute a suitable job size estimate is sufficient: this allows to distinguish large jobs from small ones, thus avoiding "inversions" in the job schedule.
We support the above claim with Figure 5.9, that shows the ECDF of the estimation error we measured for MAP and REDUCE phase across all jobs and workloads. For the MAP phase, some jobs (less than 5%) are under-estimated by a factor of 0.4/0.5, while some jobs (less than 4%) are over-estimated by a factor of 2/3.5. For the reduce phase, the number of under-estimated jobs is very small and in general under-estimation is negligible while over-estimation happens only for 10% of jobs by a factor inferior to 2. As a consequence, job size estimates are in the same order of magnitude of real job sizes. This means that two jobs from the same "bin" can be switched but two jobs from two different bins are hardly or never switched (see Section 4.5).
In Figures 5.10a and 5.10b we decompose estimation errors according to the size of the job (bin for the dataset used), to understand whether errors are more likely to occur for larger or smaller jobs. Results for jobs in bin 1 are omitted because those jobs have less than 5 tasks and they finish before the estimation is done. The boxplots (indicating quartiles in the boxes and outliers outside of whiskers) show that HFSP tends to over-estimate rather than under-estimate job sizes. In our experiments, estimation error is bounded by the factor of 3.5 for the MAP phase and 2 for the REDUCE phase. The majority of the estimated sizes are around 1, showing that often HFSP estimates a size that is very close to the correct one. Since jobs from bins 3 and 4 have more tasks than jobs from bin 2, the estimations for bins 3 and 4 are less precise than the estimations for bin 2 (Table 5.1). In other words, HFSP gives a correct priority to jobs from bin 2 while it tends to give a lower priority to jobs from bins 3 and 4. This is in line with the philosophy behind HFSP of favoring smaller jobs.
Given these results and the results of Chaper 3, we conclude that the output of our estimator is good enough for HFSP to schedule jobs correctly.

Preemption techniques

As discussed in Section 4.6, job preemption can be implemented with two approaches: one can preempt each task of the job with the KILL primitive, or one can wait for each task to complete (the WAIT primitive, which is the default behavior of HFSP) before granting the resources to the tasks of the preempting job. Here we analyze the impact on the system performance and on the fairness of both approaches. To this aim, we focus on the TEST workload, which is the most problematic workload for HFSP (see Section 5.3.1). Figures 5.11 and 5.12 show the ECDF of response times and slowdown.
We notice that killing tasks improves HFSP fairness: indeed, when a waiting job is granted higher priority than a running job, the scheduler kills immediately the running tasks of the latter, and freeing up resources almost immediately. Without the KILL primitive, the scheduler can only grant resources to smaller, higher priority jobs when running tasks are complete.

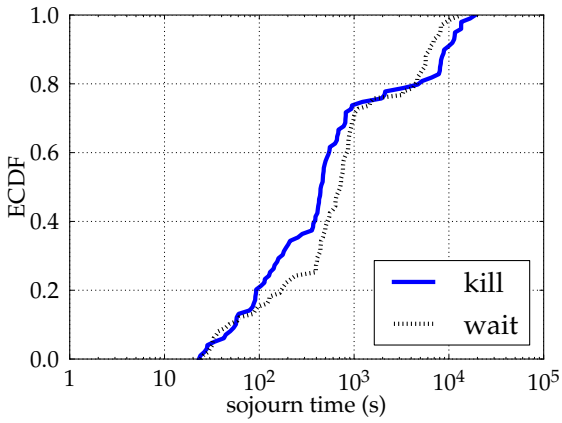


Figure 5.11: ECDF of job response times, for the TEST workload. Comparison between the KILL and WAIT preemption primitives.

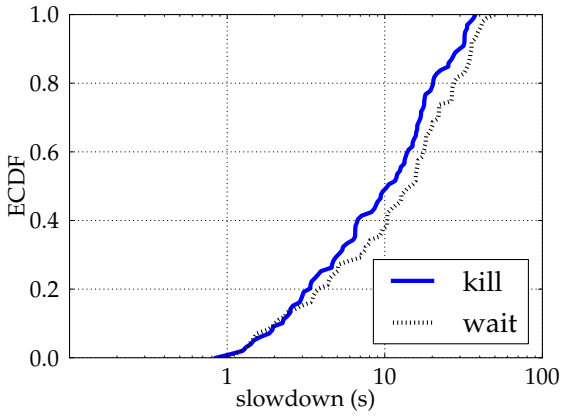


Figure 5.12: ECDF of the per-job slowdown, for the TEST workload. Comparison between the KILL and WAIT preemption primitives.

The situation changes when considering response times. Indeed, killing tasks wastes work, and this is particular true for tasks that are close to completion. Figure 5.11 shows that the KILL primitive is especially problematic for large jobs: as they last longer, they have more chances to have their tasks killed.
In summary, the lesson we learned is that the choice of the primitive to use to perform job preemption depends on the "functional" objectives of an administrator. If fairness is considered important, then the KILL primitive is more appropriate. Instead, if system responsiveness is the objective, then the WAIT primitive is a sensible choice. As a future research direction, we plan to integrate HFSP with new preemption primitives [65, 64] that aim at striking a good balance between fairness and responsiveness.

5.5 Summary

In this Chapter we presented a study of the HFSP performances in a real Hadoop deployment against an industrial quality scheduler such as the Fair Scheduler. HFSP performs very well in all the proposed scenario, in particular when the cluster is heavy loaded. In other words, HFSP permits a better utilization of the cluster resources and allows to use smaller clusters compared to other schedulers.
While the results about the performance of our scheduler are self-explanatory, we believe there is another lesson to be learned from this Chapter. HFSP is based on FSPE but, since it is a real implementation, its design posed many new practical issues – issues that are not present when dealing with simple abstract models. Hadoop is very different from an ideal single-server system like the one used in Chaper 3, nevertheless we showed with the implementation of the HFSP scheduler that the FSPE policy can be adapted to a practical and complex system.
FSPE is a preemptive scheduling policy and, consequently, also HFSP is preemptive. The lack of an efficient preemption mechanism in Hadoop – the primitive KILL wastes resources, while the primitive WAIT introduces a delay – makes it difficult to implement such a scheduler. In the next Chapter we are going to address this problem by introducing a new preemption mechanism for MapReduce.

Chapter 6

OS-Assisted Task Preemption

In this Chapter we present a new task preemption primitive for Hadoop that is able to preempt running tasks without wasting work. Our work is based on the Operating System preemption mechanisms and has the advantage that it works with existing MapReduce jobs without requiring any further implementation effort from developers, unlike works such as Natjam [65]. This Chapter is organized as follow: Section 6.1 describes the idea, Section 6.2 evaluates the benefits of our idea against existing preemption techniques in Hadoop and Section 6.3 discusses advantages and disadvantages of our preemption primitive.

6.1 OS-assisted Task Preemption

We now describe our task preemption primitive, that implements task suspension and resume operations. First, we outline how process suspension and memory paging work in modern operating systems. Then, we present the implementation of our preemption mechanism.

6.1.1 Suspension and Paging in the OS

Here we provide a synthetic description of the way OSes perform memory management, which motivates our design and implementation. A more in-depth description of such mechanisms can be found, for example, in the work of Arpaci-Dusseau [94, Chapters 20 and 21].

In general, system RAM is occupied by file-system (disk) cache and runtime memory allocated by processes (including map/reduce tasks); when RAM is full – for whatever reason – the OS needs to *evict* pages from memory, either by reclaiming space (and evict pages) from the file-system cache or by *paging out* runtime memory to the swap area. Since Hadoop workloads involve large sequential reads from disks, it is a best practice to configure the Linux kernel to give precedence to runtime memory, always evicting file-system cache first [95]. The system therefore only pages out runtime memory to

avoid "out of memory" conditions, *i.e.* when the memory allocated by *running* processes exceeds the physical RAM.
To decide which pages to swap to disk, OSes generally employ a policy which is a variant of least-recently-used (LRU) [96]; *clean* pages – *i.e.*, those that have not been modified since the last time they have been read from disk – do not need to be written and get prioritized when performing eviction. Page-out operations are generally clustered to improve disk throughput (and amortize on seek costs) by writing multiple pages to disk in a batch. These implementation policies ensure that paging is efficient and with small overheads, especially if a suspended processes leads to swapping. Most importantly for our case, pages from suspended processes are evicted before those from running ones.
We recall that it is necessary to make sure that the aggregate memory size for all processes – both running and suspended – does not exceed the size of the swap space on disk, because in such a case the operating system would be forced to kill processes. Since Hadoop tasks can only allocate a limited amount of memory, this can be ensured by configuring the scheduler so that also the number of suspended tasks per task-tracker is limited.

6.1.2 Thrashing

Paging, in general, is not problematic unless *thrashing* happens, a phenomenon where data is continuously read from and written to swap space [90] on disk. Thrashing is caused by a *working set* – *i.e.*, the set of pages accessed by running programs – which is larger than main memory.
In Hadoop, thrashing is avoided because two mechanisms are in place: *i)* the number of running tasks per machine is limited (and controlled via a configuration parameter); and *ii)* the heap space size that a given task can allocate is limited (and also controlled via configuration). Proper Hadoop configuration can thus limit working set size and avoid thrashing.
The aforementioned mechanisms prevent thrashing in the same way even when suspension is used. Memory allocated by suspended processes is *outside the working set* and hence *cannot cause thrashing*; pages allocated for the suspended processes are paged out and in *at most once*, respectively after suspension and resuming. Thrashing could only happen if a given job is continuously suspended and resumed by the scheduling mechanism: the moderate cost of a suspend-resume cycle can be thus multiplied by the number of cycles. A reasonable scheduler implementation should take into account that suspending and resuming a job has a cost, and should take measures to avoid paying it too often.

6.1.3 Implementation Details

The concepts that we illustrate here are valid for both Hadoop 1 [44], which is the most widely used Hadoop implementation in production, Hadoop 2, which uses a new infrastructure for resource negotiation called YARN [97], and even other frameworks such as Spark [86]. Currently, our implementation targets Hadoop version 1.
Our preemption primitive exposes an API that can be used both by users on the command line and by schedulers. Mirroring the implementation of the KILL primitive in Hadoop, we introduce *i)* new messages between the JobTracker (a centralized machine responsible for keeping track of system state and scheduling) and TaskTrackers (machines responsible for running Map/Reduce tasks), and *ii)* new identifiers for task states in the JobTracker.

JobTracker

Hadoop has a "heartbeat" mechanism where, at fixed intervals and every time a task finishes, TaskTrackers inform the JobTracker about their state.
As soon as the JobTracker receives the command to suspend a task from the user or the scheduler, that task is marked as being in a MUST_SUSPEND state. At the following heartbeat from the involved TaskTracker, the JobTracker piggybacks the command to suspend the task. The following heartbeat notifies the JobTracker whether the task has been suspended – which triggers entering the SUSPENDED state in the JobTracker – or whether it completed in the meanwhile.
Analogous steps are taken to resume tasks, exchanging appropriate messages and handling the MUST_RESUME state, returning the state to RUNNING when the process is over.

TaskTracker

In Hadoop, Map and Reduce tasks are regular Unix processes running in child JVMs spawned by the TaskTracker. This means that they can safely be handled with the POSIX signaling infrastructure. In particular, to suspend and resume tasks, our preemption primitive uses the standard POSIX SIGTSTP and SIGCONT signals.
These signals are used because (unlike SIGSTOP) they allow handlers to be written to manage external state, *e.g.*, when closing and reopening network connections.

Job and Task Scheduler

We factor out the role of task eviction policies implemented by the scheduler, such as the one presented in Section 4.6 for HFSP, by building a new scheduling component for Hadoop – a dummy scheduler – which dictates task eviction according to static configuration files. This allows to specify, using a series of simple triggers, which jobs/tasks

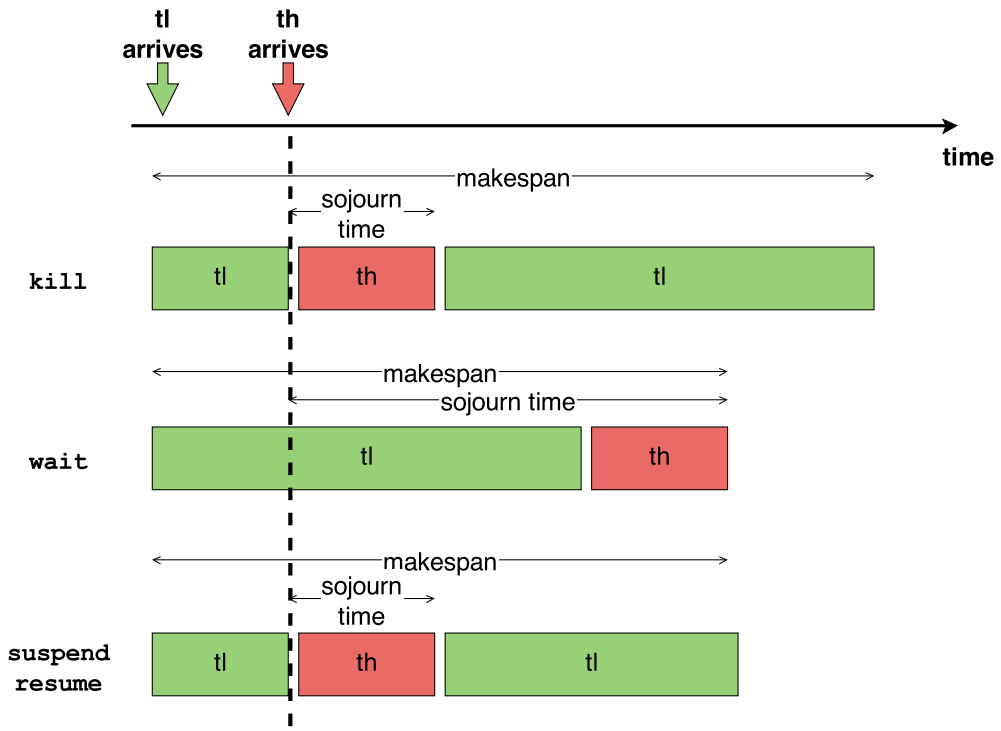


Figure 6.1: Task execution schedules.

are run in the cluster and which are preempted. In addition to executing jobs and preempting tasks with our SUSPEND/RESUME primitives, the dummy scheduler also allows using the KILL primitive and to WAIT, for the purpose of a comparative analysis.

6.2 Experimental Evaluation

In our experiments, we evaluate preemption primitives in terms of the latency they introduce and the amount of redundant work they require. We show that our approach outperforms other preemption primitives and has a small overhead both when jobs are lightweight in terms of memory, and when they are memory-hungry.

6.2.1 Experimental Setup

Our SUSPEND/RESUME primitives operate at the task level, and behave in the same way for both Map and Reduce tasks. We evaluate the behavior of the system in a simple setup: our dummy scheduler runs two single-task, map-only jobs, called t_h and t_l (h and l stand for high and low priority respectively). t_l processes a single-block file stored on HDFS, with size 512 MB; t_h processes single HDFS input block of size 512 MB. Both jobs run synthetic mappers, which read and parse the randomly generated input. The physical memory of the machine running the tasks is 4 GB. We remark that this setup is

analogous to the one used by Cho *et al.*, who evaluated their preemption primitive using similar synthetic jobs created by the SWIM workload generator [98].
In our experiments, our dummy scheduler preempts the low-priority task t_l after it has reached a completion rate $r\%$ (*i.e.*, $r\%$ of the input tuples have been processed) and grants the task slot to the high priority task t_h. Once t_h is completed, the scheduler resumes t_l, which can complete as well.
Next, we evaluate the behavior of our SUSPEND/RESUME preemption mechanism against the two baseline primitives available in Hadoop: WAIT and KILL. When waiting, task t_h is simply executed after t_l completes; when killing, task t_l is killed as soon as t_h is scheduled, and t_l is rescheduled from scratch after the completion of t_h. This simple experimental setup is illustrated in Figure 6.1 on the preceding page.
According to Hadoop configuration best practices, in our experimental setup we prioritize runtime memory over disk cache and therefore limit swapping, as discussed in Section 6.1.1, by setting the Linux `swappiness` parameter to 0.

6.2.2 Performance Metrics

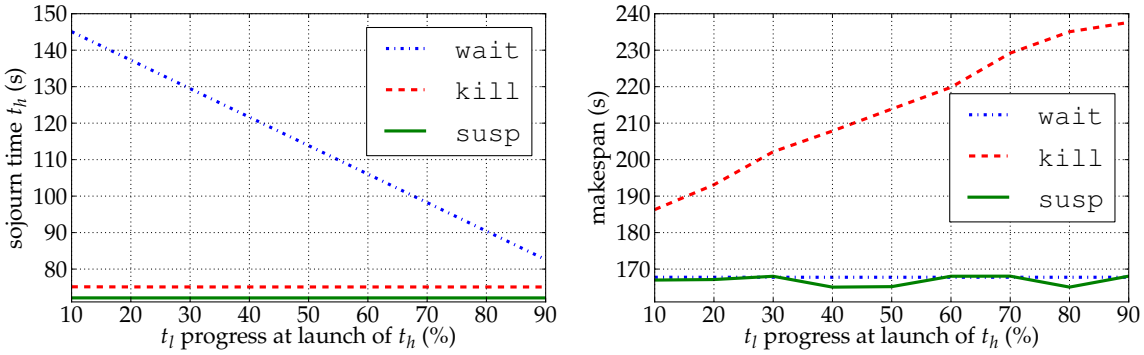
Our goals are ensuring low latency for high-priority tasks, and avoid wasting work: we quantify them, respectively, with the *sojourn time of* t_h and the *makespan* of the workload. *Sojourn Time of* t_h is the time that elapses between the moment t_h is submitted and when it completes; *makespan* is the time that passes between the moment in which the first task t_l is submitted and when *both* tasks are complete.

6.2.3 Results

We focus on experimental results in case of light-weight tasks. This is the standard case for "functional", stateless, mappers and reducers. In this case, the amount of memory that tasks allocate is essentially due to the Hadoop execution engine (*i.e.*, JVM, I/O buffers, overhead due to sorting, *etc.*).
Stateful mappers and reducers, instead, can allocate non-negligible amounts of memory; we thus complement our experiments by studying our performance metrics and overheads for memory-hungry jobs, which represent a worst-case scenario for our preemption primitive.
All our results are obtained by averaging 20 experiment runs; we omit error bars for readability: in all data points reported, minimum and maximum values measured are within 5% of the average values.

Baseline Experiments.

Figure 6.2a on the following page illustrates the sojourn time of t_h: the arrival rate of $_h$ is a parameter defined as a function of t_l progress, as shown on the x-axis.



(a) Sojourn time of t_h

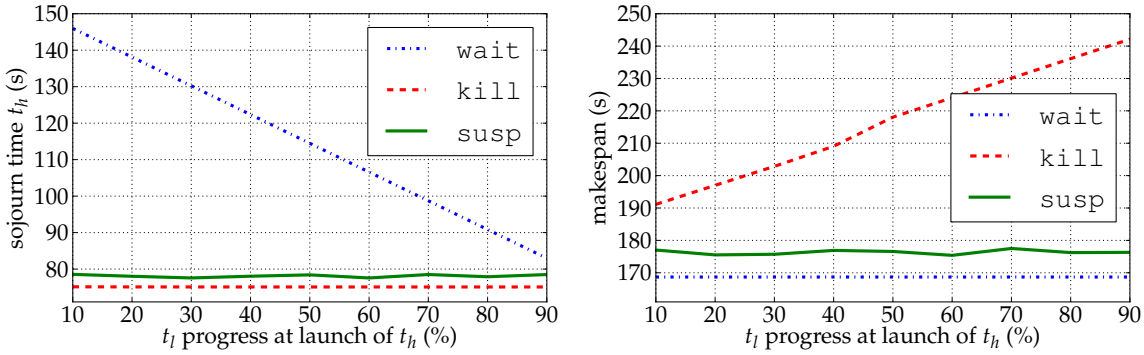
(b) Makespan

Figure 6.2: Baseline experiments: a comparison of the three preemption primitives with light-weight tasks.

The KILL and our SUSPEND/RESUME primitives achieve small sojourn times, as opposed to WAIT, in particular when t_h arrives early. However, they both incur in some overheads: KILL runs a cleanup task to remove temporary outputs of the killed task; SUSPEND/RESUME may slow down t_h in case paging out memory occupied by t_l is needed. In our baseline setup, both jobs are light-weight, hence the suspended process resides only in memory. This explains the small advantage for our mechanism, which outperforms all other primitives even when t_h arrives at 90% completion rate of task t_l.
Figure 6.2b illustrates our results for the makespan metric, using the same setup described above. In this case, the makespan is heavily affected by a preemption primitive that wastes work. The WAIT policy, at the cost of delaying t_h, avoids supplementary work and achieves a small makespan; the KILL primitive, instead, wastes all the work done by t_l before preemption. Finally, our preemption primitive behaves similarly to the WAIT policy, despite the possible overhead due to page-out/page-in cycles.
For light-weight jobs, we conclude that our primitive is superior to both alternatives, as both sojourn times and makespan are small. We note that the authors of Natjam measured an overhead of around 7% in terms of makespan, in similar experimental settings as ours. Our findings suggest that the overhead in our case is negligible.

Worst-Case Experiments.

The experiments discussed above are valid for simple implementations of Map and Reduce tasks, that carry out stateless computations on their input. Stateful tasks can, however, allocate memory, which may force the OS to swap. Since clusters often have plentiful available memory [106], such a situation is unlikely to be frequent. However, we still consider a "worst case" scenario to stress our primitive: both t_l and t_h allocate a large amount of memory (2 GB in our case; we note that this requires an *ad hoc* change to the Hadoop configuration since Hadoop jobs are not generally allowed to allocate such an amount of memory). This value makes sure that, when running a single task the system



(a) Sojourn time of t_h

(b) Makespan

Figure 6.3: Worst-case experiments: a comparison of the three preemption primitives with memory-hungry tasks.

does not have to recur to swap; conversely, when the two tasks are present in the system at the same time, one of them is forced to page out memory. We ensure that tasks allocate memory and that the OS marks pages as "dirty", by writing random values to all memory at task startup, and reading them back when finalizing the tasks.
Figures 6.3a and 6.3b present the sojourn time and the makespan for the worst-case experimental setup. While our preemption primitive still outperforms both alternatives with respect to both metrics, it is possible to notice that the overheads related to paging are visible: with respect to the sojourn time, the KILL primitive achieves a slightly lower value; similarly, the WAIT primitive achieves slightly smaller makespan. Overall, the overhead due to our preemption primitive is marginal: we further investigate and quantify it in the next section.

6.2.4 Impact of Memory Footprint.

We now focus on a detailed analysis of the overheads imposed by the OS paging mechanism on the performance of our preemption primitive. To do so, we vary the amount of memory a task allocates in the setup phase.[1] In our experiments t_l allocates 2.5 GB of memory, and we parametrize over the amount of memory t_h allocates. For each experimental run, we measure the number of bytes swapped by the process executing t_l, and compute the degradation of sojourn time and makespan compared to the KILL and WAIT primitives, respectively.
Figure 6.4 indicates that the overheads due to paging are roughly linearly correlated to the amount of data swapped to disk. For the sojourn time, our preemption primitive degrades when t_h allocates more than 1.5 GB of RAM: in the worst-case, sojourn time is 20% larger than with the KILL primitive. Similarly, for the makespan, our mechanism degrades when t_h allocates more than 1.3 GB: in the worst-case, makespan is 12% larger

[1]This is where, generally, auxiliary data structures are created to maintain an internal state in a task.

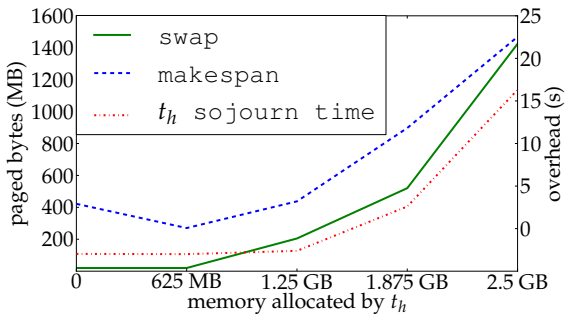


Figure 6.4: Overheads when varying memory usage.

than with the WAIT primitive. Finally, we note that swapped data grows more than linearly because of an approximate implementation of the page replacement algorithm in Linux (and other modern OSes), which can lead to more swapping than strictly necessary [107, Chapter 17].

6.3 Discussion

We now elaborate more on the implications of the new preemption primitive we introduce in this work.

6.3.1 Scheduling and Eviction Strategies

As we discussed in the Introduction, our SUSPEND primitive gives one more opportunity to the developers of schedulers, in order to perform more efficient preemption. As we have shown with the results of Section 6.2, our primitive generally performs close to optimally in most cases; however, for freshly started tasks, it may be preferable to use the KILL primitive, and for tasks that are very close to completion it may be better to simply WAIT for them to finish.

Task Eviction Policies.

An important topic that falls under the responsibility of the schedulers is to decide *which* task(s) to evict once a high-priority job needs time to execute. Cho *et al.* [65] propose to suspend tasks that are closest to completion, in order to have all tasks of a job as close to each other as possible. If the goal is instead to avoid redundant work and reduce makespan, another possible strategy may aim to suspend tasks with smaller memory footprints, which reduces overheads according to our experimental results.

Resume Locality.

In our implementation, a suspended process can only be resumed on the same machine it was suspended on. If the same task gets scheduled on a different machine, it has to be

restarted from scratch, losing work done so far: in that case, the SUSPEND is effectively analogous to a delayed KILL. We call this issue *resume locality* due to its similarity with the *data locality* issue – *i.e.*, the problem of running mappers on the machines that have local copies of data.
Hadoop schedulers, such as HFSP (Section 4.4.4), generally handle data locality by using the simple technique of delay scheduling [99]: waiting a fixed amount of time before scheduling non-local copies of data. Only if that threshold is exceeded, a non-local mapper is run. The same technique can be used for our resume locality issue.
Analogously to our approach, Natjam only supports local resumes. As a future improvement, the authors suggest moving the checkpoints used to mark task state and reduce inputs over the network; a similar approach could be taken also in our case, using process migration facilities such as CRIU [93]. However, extreme care should be taken before attempting to use such a non-local resume in particular for reducers, since the cost of moving non-local inputs can be very large.

6.3.2 Implications on Task Implementation

In most cases, our SUSPEND/RESUME mechanism is transparent towards the implementation, and task implementations that correctly handle error conditions and the possibility of being killed by the scheduler will also handle suspension correctly. However, we add a few notes regarding tasks with external state and ways in which task implementation can control the memory footprint.

External State.

In some cases, Hadoop jobs can interact with the external world through more than inputs and outputs: they can use network connections and/or use "Hadoop Streaming", whereby arbitrary executables can be used as mappers or reducers, interacting with the Hadoop framework through Unix pipes. In these cases, there are interactions that happen outside the control of Hadoop; in the most common case, external software would correctly pause waiting for the next input from a suspended task; however, when the interaction happens with a complex program, the fact that they correctly handle suspended programs should be tested.

Controlling Memory Footprint.

We have seen that the memory footprint allocated by a process has an impact on the overheads due to suspension; when writing task implementations, it is good measure to take this into account and optimize for lower memory footprints.
Java garbage collectors differ in the way they are implemented: some of them release memory to the OS when they stop using it, others do not [92]. It is therefore a good idea to configure Java to use a garbage collector that does release memory, such as the

new G1 implementation [91]. It is also possible to hint the garbage collector to run using `System.gc()`; this is advisable after disposing of large objects in memory.

6.4 Summary

In this Chapter we presented a new task preemption technique for Hadoop and we showed its impact on the framework. Task suspension works better than existing techniques in almost every case and we believe it could be used by current and new Hadoop preemptive schedulers.

Chapter 7

Conclusion

This thesis focused on size-based scheduling policies for real systems. While this thesis provides various contributions to the existing scheduling theory and practice, the main and fundamental contribution of our work is the definition and the implementation of HFSP, a size-based scheduler for Hadoop that performs better than the existing, industrial-ready schedulers by exploiting job sizes. To achieve this result, many challenges have been addressed and resolved.

The most important problem with size-based scheduling for real systems is the lack of knowledge of jobs sizes: when the information on job size is missing, our idea is to estimate the job sizes while they are running. Estimating the size of a job introduces an error: Chapter 3 has been dedicated precisely to study the impact of such errors in the scheduling decisions. We presented SRPTE, FSPE and FSPE+PS, three schedulers that work with estimated sizes, and we showed that all three can tolerate estimations errors under certain conditions of practical interest. Every system that meets such conditions is a candidate for the implementation of one of the above size-based scheduling policies.

In Chapter 4, we presented the design and architecture of HFSP, a size-based scheduler for Hadoop based on FSPE. HFSP raised many challenges, such as how to estimate sizes, or how to deal with job aging, which we solved to create a working size-based preemptive scheduler. HFSP leads to smaller mean response time compared to existing schedulers, while being fair among jobs. Our results showed that HFSP can deal with bursts of jobs better than the Fair Scheduler, and thus it can be used in smaller clusters. While our work is for Hadoop, we believe that HFSP can be ported to other real systems such as Spark or Naiad or even other non DISC systems.

Finally, in Chapter 6, we extended Hadoop MapReduce to provide a new task preemptive mechanism, called Suspend, that is as efficient as the Kill preemption, but avoids all the problems that Kill has. We also discussed when Suspend should be used, and in which scenario the Kill should be used instead.

We believe that the approach to size-based scheduling proposed in this dissertation could be a starting point for future works with all the advantages that the scheduling theory has shown in the past.

While the thesis answers fundamental questions, it inevitably raises many new ones that will require further studies in the domain.

7.1 Future Work

In this Section we present some of the most promising ideas, the first two related to the job size estimation, and the last related to the task suspension.

Complex Jobs

HFSP is a scheduler for MapReduce and works by considering each job independent from the others. While this is how a standard job scheduler for MapReduce works, a raising trend in industry is to avoid defining jobs at MapReduce level and use higher-level libraries, such as Cascading [115], Scalding [116], Cascalog [117] and Scoobi [118], or languages, such as Pig [105] and Hive [104]. The main feature of high-level libraries and languages is to abstract from MapReduce and let the user define its MapReduce program in an easier way. For example, Hive lets the user define a MapReduce program in SQL. Those "high-level" programs are then compiled in a set of MapReduce jobs and are submitted as a chain of jobs, where each job of the chain can start working only after the previous one completes. While for the MapReduce engine this chain of jobs is seen as many independent jobs, the scheduler should take in account that the jobs are part of the same chain and should consider them as a big *complex job*.
Evaluating the size of a *complex job* is indeed different from evaluating the size of a normal job. High-level languages and libraries offer hints on the performances of some operations. Those hints are usually used to do cost-based optimization but can also be used to have a better idea of the size of a complex job. We believe this is a research direction for our work.

Recurring Jobs

Hadoop workloads often contain recurring jobs, that are jobs that are periodically submitted to the cluster. For instance, Agarwal *et al.* [52] show that in their production system, recurring jobs are the 40% of all the submitted jobs in their cluster. In our opinion this kind of jobs can contribute to an advanced version of HFSP, in which job sizes are estimated based not only on their sizes, but also on the past runs. We recall that the output of a job execution is its real size, which is exactly what a size-based scheduling policy needs. Morevoer, a scheduler can obtain a very accurate size for them by observing all the past runs and then build a job profile that aggregates all the informations about the job. While it is easy to collect the real sizes of a job from its past runs, there are plenty of challenges raised by this technique. First, and most importantly, the job size depends on the input. If the input does not change between two runs, then the size

of a job is likely to be nearly identical to the previous one – some variations may de due to data-locality. On the contrary, if the input is different, then the only way to use the information about past runs is to understand the characteristics of the input, *e.g.* the distribution of the keys, and then adapt the past runs information to the new input.

Extending schedulers with task suspension

In Chapter 5.4 we showed the performance of HFSP with or without the Hadoop built-in preemption mechanism activated. In Chapter 6 we presented a new task preemption mechanism called SUSPEND that aims at having the same advantages of the built-in KILL primitive without the drawbacks. When suspending a task of a job, we need to select, among the possible tasks, the ones according to some metric[1]. In Section 4.6 we adopted a strategy that is inspired by the approach proposed when using the KILL primitive. Nevertheless, there could be other strategies, tailored to the characteristics of the SUSPEND primitive, that may result in a more efficient use of the resources.
Another interesting aspect is the so called *Task Locality Problem*: when a task is suspended, it can be resumed only on the same machine. This problem can lead to poor performance if, for instance, some resources become available (*e.g.*, another task completed) on a machine different from the one in which the task has been suspended. The task cannot be resumed on the other machine, and thus the resources may be wasted. there are different approaches to solve this problem – approaches that should be investigated in depth to understand their advantages and disadvantages. A strategy that could be applied without modifying Hadoop is to use the *Speculative Execution*, *i.e.*, to run a copy of the suspended task on the machine with available resources. In this case, there are two possibilities: i) the speculative copy finishes the work and the suspended task can be killed ii) the suspended task is resumed and completes in the machine where it has been suspended and the speculative copy is killed. While speculative execution is a simple strategy, it basically consists in running the task more than once, and thus the advantages of using the suspension may be lost. Moreover, output data of the task is partially duplicated, and this wastes both disk and networking resources.
A second strategy could be the modification of Hadoop to support task check-pointing: when a task is suspended and resources are released on another machine, on the other machine Hadoop starts a "special task" that continues the work of the previous one, without redoing all the work done by the suspended task. Basically, Hadoop considers the task as split in two tasks, and treats them independently. This strategy has the problem that, if the task has some internal state, like an accumulator, this state must be migrated to the new machine.

[1]Consider the case in which, while a large job with many tasks is running, a small job that requires only few tasks arrive: only a fraction of the task of the large job should be suspended to give the resources to the small job, while the other tasks can continue to work.

As final note, we highlight that all the preemptive schedulers in Hadoop, such as the Fair or the Capacity Schedulers, could be extended with our suspension mechanism to improve their performance.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[2] T. White, *Hadoop: The Definitive Guide: The Definitive Guide.* O'Reilly Media, 2009.

[3] M. Zaharia *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proc. of USENIX NSDI*, 2012.

[4] B. Hindman *et al.*, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proc. of USENIX NSDI*, 2011.

[5] Apache, "Oozie Workflow Scheduler," http://oozie.apache.org/.

[6] ——, "PigMix," https://cwiki.apache.org/PIG/pigmix.html.

[7] TPC, "Tpc benchmarks," http://www.tpc.org/information/benchmarks.asp.

[8] Apache, "Hadoop nextgen MapReduce (yarn)," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[9] ——, "Hadoop on demand," http://hadoop.apache.org/docs/stable/hod_scheduler.html.

[10] M. Schwarzkopf *et al.*, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. of EuroSys*, 2013.

[11] B. Moseley *et al.*, "On scheduling in map-reduce and flow-shops," in *In Proc. of ACM SPAA*, 2011.

[12] K. Fox and B. Moseley, "Online scheduling on identical machines using SRPT," in *In Proc. of ACM-SIAM SODA*, 2011.

[13] D. Lu, H. Sheng, and P. Dinda, "Size-based scheduling policies with inaccurate scheduling information," in *Proc. of IEEE MASCOTS*, 2004.

[14] , "`sysctl` linux kernel documentation," http://www.kernel.org/doc/Documentation/sysctl/vm.txt.

[15] Apache, "Hadoop wiki, powered by," http://wiki.apache.org/hadoop/PoweredBy.

[16] , "AWS elastic compute cloud," http://aws.amazon.com/ec2/.

[17] Y. Chen *et al.*, "Statistical workload injector for MapReduce," https://github.com/SWIMProjectUCB/SWIM.

[18] , "Hadoop Mumak," https://issues.apache.org/jira/browse/MAPREDUCE-728.

[19] Apache, "Hadoop fair scheduler," http://hadoop.apache.org/docs/stable/fair_scheduler.html.

[20] ——, "Hadoop capacity scheduler," http://hadoop.apache.org/docs/stable/capacity_scheduler.html.

[21] , "Hadoop cluster setup," http://hadoop.apache.org/docs/stable/cluster_setup.html.

[22] Apache, "Hadoop MapReduce JIRA 1184," https://issues.apache.org/jira/browse/MAPREDUCE-1184.

[23] Y. Chen, S. Alspaugh, and R. Katz, "Interactive query processing in big data systems: A cross-industry study of MapReduce workloads," in *Proc. of VLDB*, 2012.

[24] K. Ren *et al.*, "Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads," in *Proc. of VLDB*, 2013.

[25] Y. Chen, "We don't know enough to make a big data benchmark suite - an academia-industry view," in *Proc. of WBDB*, 2012.

[26] Y. Chen, A. Ganapathi, R.Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *Proc. of IEEE MASCOTS*, 2011.

[27] M. Harchol-Balter, "Queueing disciplines," in *Wiley Encyclopedia Of Operations Research and Management Science.* John Wiley & Sons, 2009.

[28] G. Anantharanayanan *et al.*, "True elasticity in multi-tenant clusters through amoeba," in *Proc. of ACM SOCC*, 2012.

[29] L. Cheng, Q. Zhang, and R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous MapReduce workloads," in *Proc. of CNSM*, 2011.

[30] C. Tian, H. Zhou, Y. He, and L. Zha, "A dynamic MapReduce scheduler for heterogeneous workloads," in *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on.* IEEE, 2009, pp. 218–224.

[31] A. D. Popescu *et al.*, "Same queries, different data: Can we predict query performance?" in *Proc. of SMDB*, 2012.

[32] Y. Kwon *et al.*, "Skewtune: mitigating skew in MapReduce applications," in *Proc. of ACM SIGMOD*, 2012.

[33] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of ACM EuroSys*, 2010.

[34] K. Kc and K. Anyanwu, "Scheduling Hadoop jobs to meet deadlines," in *Proc. of CloudCom*, 2010.

[35] H. Chang *et al.*, "Scheduling in MapReduce-like systems for fast completion time," in *Proc. of IEEE INFOCOM*, 2011.

[36] A. S. Schulz, "Scheduling to minimize total weighted completion time: Performance guarantees of lp-based heuristics and lower bounds," in *Integer Programming and Combinatorial Optimization*. Springer, 1996, pp. 301–315.

[37] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in Hadoop," in *Proc. of JSSPP*, 2010.

[38] E. Friedman and S. Henderson, "Fairness and efficiency in web server protocols," in *Proc. of ACM SIGMETRICS*, 2003.

[39] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 24–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972490

[40] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. of USENIX NSDI*, 2011.

[41] T. Sandholm and K. Lai, "MapReduce optimization using regulated dynamic prioritization," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. ACM, 2009, pp. 299–310.

[42] J. Tan, X. Meng, and L. Zhang, "Delay tails in MapReduce scheduling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp. 5–16, 2012.

[43] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. of USENIX OSDI*, 2004.

[44] Apache, "Hadoop: Open source implementation of MapReduce," http://hadoop.apache.org/.

[45] M. Isard *et al.*, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. of ACM EuroSys*, 2007.

[46] J. Tan, X. Meng, and L. Zhang, "Performance analysis of coupling scheduler for MapReduce/Hadoop," in *Proc. of IEEE INFOCOM*, 2012.

[47] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for MapReduce environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 235–244.

[48] ——, "Two sides of a coin: Optimizing the schedule of MapReduce jobs to minimize their makespan and improve cluster performance," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*. IEEE, 2012, pp. 11–18.

[49] M. Isard *et al.*, "Quincy: fair scheduling for distributed computing clusters," in *Proc. of ACM SOSP*, 2009.

[50] P. Costa *et al.*, "Camdoop: Exploiting In-network Aggregation for Big Data Applications," in *Proc. of USENIX NSDI*, 2012.

[51] J. Zhang *et al.*, "Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions," in *Proc. of USENIX NSDI*, 2012.

[52] S. Agarwal *et al.*, "Re-optimizing Data-Parallel Computing," in *Proc. of USENIX NSDI*, 2012.

[53] M. Harchol-Balter *et al.*, "Size-based scheduling to improve web performance," *ACM TOCS*, vol. 21, no. 2, 2003.

[54] M. Dell'Amico, "A simulator for data-intensive job scheduling," EURECOM, Tech. Rep. RR-13-282, 2013.

[55] J. Nagle, "On packet switches with infinite storage," *Communications, IEEE Transactions on*, vol. 35, no. 4, 1987.

[56] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM TON*, vol. 6, no. 5, 1998.

[57] M. Pastorelli *et al.*, "Practical size-based scheduling for MapReduce workloads," *CoRR*, vol. abs/1302.2749, 2013.

[58] K. Ousterhout *et al.*, "The case for tiny tasks in compute clusters," in *Proc. of USENIX HotOS*, 2013.

[59] E. B. Nightingale *et al.*, "Flat datacenter storage," in *Proc. of USENIX OSDI*, 2012.

[60] S. Gorinsky and C. Jechlitschek, "Fair efficiency, or low average delay without starvation," in *Proc. of IEEE ICCCN*, 2007.

[61] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. of USENIX OSDI*, 2010.

[62] K. Ren, G. Gibson, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence; a comparative workloads analysis from three research clusters." in *SC Companion*, 2012, p. 1452.

[63] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs scale-out for hadoop: Time to rethink?" in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 20.

[64] P. M. Mario Pastorelli, Matteo Dell'Amico, "Os-assisted task preemption for hadoop," in *Proc. of IEEE ICDCS workshop*, 2014.

[65] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 6.

[66] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and stretch metrics for scheduling continuous job streams," in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1998, pp. 270–279.

[67] A. S. Tanenbaum and A. Tannenbaum, *Modern operating systems*. Prentice hall Englewood Cliffs, 1992, vol. 2.

[68] Stallings, *Operating Systems*. Prentice hall, 1995.

[69] M. Dell'Amico, D. Carra, M. Pastorelli, and P. Michiardi, "Revisiting size-based scheduling with estimated job sizes," *arXiv preprint arXiv:1403.5996*, 2014.

[70] D. Lu, H. Sheng, and P. Dinda, "Size-based scheduling policies with inaccurate scheduling information," in *MASCOTS*. IEEE, 2004.

[71] A. Wierman and M. Nuyens, "Scheduling despite inexact job-size information," in *SIGMETRICS PER*, vol. 36, no. 1. ACM, 2008.

[72] M. Pastorelli, A. Barbuzzi, D. Carra, M. Dell'Amico, and P. Michiardi, "HFSP: size-based scheduling for Hadoop," in *BIGDATA*. IEEE, 2013.

[73] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, "FLEX: A slot allocation scheduling optimizer for MapReduce workloads," *Middleware*, 2010.

[74] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack, "Analysis of LAS scheduling for job size distributions with high variance," *SIGMETRICS PER*, vol. 31, no. 1, 2003.

[75] E. J. Friedman and S. G. Henderson, "Fairness and efficiency in web server protocols," in *SIGMETRICS PER*, vol. 31, no. 1. ACM, 2003.

[76] L. Kleinrock, *Queueing systems. Volume 1: Theory.* Wiley, 1975.

[77] E. G. Coffman and P. J. Denning, *Operating systems theory.* Prentice-Hall, 1973, vol. 973.

[78] L. E. Schrage and L. W. Miller, "The queue M/G/1 with the shortest remaining processing time discipline," *Operations Research*, vol. 14, no. 4, 1966.

[79] A. Wierman, "Fairness and scheduling in single server queues," *Surveys in Operations Research and Management Science*, vol. 16, no. 1, 2011.

[80] M. Harchol-Balter, "Queueing disciplines," *Wiley Encyclopedia of Operations Research and Management Science*, 2009.

[81] A. Wierman, "Fairness and classifications," *SIGMETRICS PER*, vol. 34, no. 4, 2007.

[82] M. Dell'Amico, "A simulator for data-intensive job scheduling," *arXiv preprint arXiv:1306.6023*, 2013.

[83] R. J. Lipton and J. F. Naughton, "Query size estimation by adaptive sampling," *JCSS*, vol. 51, no. 1, 1995.

[84] S. A. Klugman, H. H. Panjer, and G. E. Willmot, *Loss models: from data to decisions.* John Wiley & Sons, 2012, vol. 715.

[85] B. Schroeder and M. Harchol-Balter, "Web servers under overload: How scheduling can help," *ACM TOIT*, vol. 6, no. 1, 2006.

[86] Apache, "Spark," http://spark.incubator.apache.org/.

[87] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: an analysis of hadoop usage in scientific workloads," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 853–864, 2013.

[88] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[89] Ö. Babaoğlu, K. Marzullo, and F. B. Schneider, "A formalization of priority inversion," *Real-Time Systems*, vol. 5, no. 4, pp. 285–303, 1993.

[90] P. J. Denning, "Thrashing: Its causes and prevention," in *Fall Joint Computer Conference*. ACM, 1968.

[91] M. Beckwith, "G1: One garbage collector to rule them all," July 2013. [Online]. Available: http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All

[92] S. Krause, "JDK 7 GC behavior: To free or not to free," August 2011. [Online]. Available: http://www.stefankrause.net/wp/?p=14

[93] "CRIU: Checkpoint/restore in userspace." [Online]. Available: http://criu.org

[94] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 2013. [Online]. Available: http://pages.cs.wisc.edu/~remzi/OSTEP/

[95] "CDH 4 installation guide – tips and guidelines," Cloudera. [Online]. Available: http://www.cloudera.com/content/cloudera-content/cloudera-docs//CDH4/4.2.0/CDH4-Installation-Guide/cdh4ig_topic_11_6.html

[96] "Page replacement design," LinuxMM. [Online]. Available: http://linux-mm.org/PageReplacementDesign

[97] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop Yarn: Yet another resource negotiator," in *SoCC*. ACM, 2013.

[98] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *MASCOTS*. IEEE, 2011.

[99] 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 *EuroSys*. ACM, 2010.

[100] M. Zaharia, "Job scheduling with the fair and capacity schedulers," 2009.

[101] K. Kc and K. Anyanwu, "Scheduling Hadoop jobs to meet deadlines," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 388–392.

[102] L. Cheng, Q. Zhang, and R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous mapreduce workloads," in *Proceedings of the 7th International Conference on Network and Services Management*. International Federation for Information Processing, 2011, pp. 189–197.

[103] J. Lin and C. Dyer, "Data-intensive text processing with MapReduce," *Synthesis Lectures on Human Language Technologies*, 2010.

[104] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *PVLDB*, vol. 2, no. 2, 2009.

[105] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*. ACM, 2008.

[106] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *USENIX NSDI*, 2012.

[107] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. O'Reilly & Associates Inc, 2005.

[108] M. Pastorelli, A. Barbuzzi, D. Carra, M. Dell'Amico, and P. Michiardi, "Practical size-based scheduling for MapReduce workloads," *CoRR*, vol. abs/1302.2749, 2013.

[109] Rackspace, "Rackspace private cloud installation manual." [Online]. Available: http://www.rackspace.com/knowledge_center/article/rackspace-private-cloud-installation-prerequisites-and-concepts

[110] "Open vswitch." [Online]. Available: http://openvswitch.org/

[111] OpenStack Foundation, "OpenStack." [Online]. Available: http://openstack.org

[112] Amazon, "Web services console." [Online]. Available: http://console.aws.amazon.com/

[113] "The bigfoot project." [Online]. Available: http://www.bigfootproject.eu

[114] OpenStack Foundation, "Openstack networking administration guide," grizzly 2013.1. [Online]. Available: http://docs.openstack.org/grizzly/openstack-network/admin/content/use_cases_single_router.html

[115] [Online]. Available: www.cascading.org/

[116] "Scalding." [Online]. Available: https://github.com/twitter/scalding

[117] "Cascalog." [Online]. Available: http://cascalog.org/

[118] "Scoobi." [Online]. Available: https://github.com/nicta/scoobi

[119] "Cloudera." [Online]. Available: http://www.cloudera.com/

[120] "Facebook." [Online]. Available: https://www.facebook.com/

[121] "Microsoft research." [Online]. Available: http://research.microsoft.com/en-us/

[122] "Naiad." [Online]. Available: http://research.microsoft.com/en-us/projects/naiad/

[123] "Ibm infosphere biginsights." [Online]. Available: http://wwwibm.com/software/data/infosphere/biginsights/

Appendix A

French Summary

Abstract

La dernière décennie a vu l'émergence de systèmes parallèles pour l'analyse de grosse quantités de données (DISC) , tels que Hadoop, et la demande qui en résulte pour les politiques de gestion des ressources, pouvant fournir des temps de réponse rapides ainsi qu'équité. Actuellement, les schedulers pour les systèmes de DISC sont axées sur l'équité, sans optimiser les temps de réponse. Les meilleures pratiques pour surmonter ce problème comprennent une intervention manuelle et une politique de planification ad-hoc , qui est sujette aux erreurs et qui est difficile à adapter aux changements.

Dans cette thèse, nous nous concentrons sur la planification basée sur la taille pour les systèmes DISC. La principale contribution de ce travail est le scheduler dit Hadoop Fair Sojourn Protocol (HFSP), un ordonnanceur préemptif basé sur la taille qui tient en considération le vieillissement, ayant comme objectifs de fournir l'équité et des temps de réponse réduits. Hélas, dans les systèmes DISC, les tailles des job d'analyse de données ne sont pas connus a priori, donc, HFSP comprends un module d'estimation de taille, qui calcule une approximation et qui affine cette estimation au fur et a mesure du progrès d'un job.

Nous démontrons que l'impact des erreurs d'estimation sur les politiques fondées sur la taille n'est pas significatif. Pour cette raison, et en vertu d'être conçu autour de l'idée de travailler avec des tailles estimées, HFSP est tolérant aux erreurs d'estimation de la taille des jobs. Nos résultats expérimentaux démontrent que, dans un véritable déploiement Hadoop avec des charges de travail réalistes, HFSP est plus performant que les politiques de scheduling existantes, a la fois en terme de temps de réponse et d'équité. En outre, HFSP maintiens ses bonnes performances même lorsque le cluster de calcul est lourdement chargé, car il focalises les ressources sur des jobs ayant priorité.

HFSP est une politique préventive: la préemption dans un système DISC peut être mis en œuvre avec des techniques différentes. Les approches

actuellement disponibles dans Hadoop ont des lacunes qui ont une incidence sur les performances du système. Par conséquence, nous avons mis en œuvre une nouvelle technique de préemption, appelé suspension, qui exploite le système d'exploitation pour effectuer la préemption d'une manière qui garantie une faible latence sans pénaliser l'avancement des jobs a faible priorité.

Introduction

La planification est l'un des problèmes les plus étudiés dans de nombreux domaines, et en particulier en informatique. Du point de vue abstrait, le problème d'ordonnancement peut être vu la manière suivante: étant donné un ensemble de ressources et un ensemble de travaux, où un emploi a besoin d'un sous-ensemble des ressources pour progresser, comment peuvent-ils être affectés pour les travaux tels que tous les emplois seront éventuellement complète tout en optimisant certains métrique? Les instances de ce problème sont les mêmes pour les systèmes d'exploitation, bases de données, réseaux, systèmes de cloud computing et beaucoup d'autres. Programmation joue un rôle fondamental car la performance d'un système varie considérablement sur la base de la politique adoptée. L'impact d'une politique d'ordonnancement est généralement mesurée par une performance indice: les indicateurs les plus couramment utilisés sont, par exemple, la moyenne le temps de réponse (également dénommé temps de séjour), qui est le temps d'un emploi est resté dans le système jusqu'à ce qu'il soit complètement servi, ou la moyenne temps d'attente, c'est à dire le temps passé dans la file d'attente avant de recevoir service ou l'équité, qui indique combien assez un travail est traitée. Certains paramètres sont parfois difficiles à optimiser à la fois, donc un politique d'ordonnancement peut avoir besoin pour faire face à un compromis. Par exemple, l' Processor Sharing (PS) politique d'ordonnancement, qui divise

également la ressources entre les emplois actuellement dans le système, fournit l'équité, mais il amène à des temps de réponse moyens élevés.

Politiques d'ordonnancement peuvent être classés en quatre familles, sur la base de savoir si une la politique est basée sur la taille ou aveugle à la taille, ou si une politique est préventive ou non préemptif - nous considérons que le travail de conservation politiques, ce qui signifie que, si il ya des emplois pour être servi, le serveur est toujours occupé à travailler sur un emploi, et aucun travail qui est fait est perdu. Une politique d'ordonnancement basé sur la taille est une politique qui prend des décisions d'ordonnancement par compte tenu de la taille d'un emploi. Par exemple, le restant le plus court Délai de traitement (SRPT) la politique est basée sur la taille, car il planifie le travail avec le plus petit temps restant de la première transformation. la First-In-First-Out (FIFO) ordonnanceur est, au contraire, aveugle à la taille depuis il horaires des emplois en fonction de leur heure d'arrivée. Une politique de planification préventive est une politique qui peut suspendre un travail en service avant complète, *i.e.* c'est à dire qu'il peut supprimer des ressources déjà attribuées à un emploi. préventive ordonnanceurs sont connus dans la littérature pour offrir de meilleures performances, car un grand travail en service ne peuvent bloquer le système, tandis que la préemption peut suspendre l'exécution d'un si grand travail en cas d'arrivée d'un petit boulot. Par exemple, le PS est un ordonnanceur préemptif tout FIFO est pas.
Parmi les quatre familles, les politiques de planification qui sont basées sur la taille et préventive sont connus pour fournir les meilleures performances: la politique de SRPT, par exemple, est optimale en termes de temps de réponse moyen [78], tandis que le Protocole de séjour Fair (FSP) [75] garantit l'équité, tout en offrant un temps moyen de réponse similaire à SRPT.

Dans cette thèse, nous nous concentrons sur les politiques de planification pour Data-Intensive Scalable Computing (DISC) systems . Ces systèmes sont composés par des centaines ou des milliers de serveurs, et les em-

plois peuvent utiliser soit toutes les ressources ou seulement une fraction d'entre eux. Compte tenu de l'adoption généralisée de ces systèmes, et leur importance croissante, il est intéressant d'étudier l'impact de la disciplines de planification sur les performances de tels parallèles et distribués systèmes.

A.1 Motivations

Politiques d'ordonnancement basés sur la taille sont connus dans la littérature pour leur supérieur performance par rapport à des politiques qui sont aveugles à la taille. Malgré cela, politiques fondées sur la taille ont peu adoption dans les systèmes réels en raison de la problèmes qui se posent lorsque ces politiques sont convertis à partir des politiques théoriques à ordonnanceurs pratiques. Par conséquent, les politiques qui sont aveugles à la taille sont les norme de facto pour les systèmes de disque.

Ordonnanceurs pour les systèmes de DISC sont complexes: en effet, la planification des emplois multiples, où chaque emploi est composé de tâches qui peuvent être exécutées en parallèle, dans un environnement distribué est une tâche difficile. Systèmes de disque, tels que Hadoop [44], Spark [86] et Naiad [122], l'utilisation des ordonnanceurs qui sont basées sur deux stratégies de base différentes: PS et FIFO. dans la production systèmes, les administrateurs système de déployer généralement des variations de cette politiques: par exemple, il peut présenter différentes classes de priorité qui peut favoriser les travaux interactifs à l'égard de travaux de traitement par lots, et utiliser FIFO au sein de chaque classe de priorité. Ces approches nécessitent que le système administrateur configure manuellement le planificateur (*i.e.*, comment gérer des priorités différentes) en fonction de la charge de travail et le réglage du système. ce processus nécessite une grande connaissance des deux la charge de travail et le système et tend être source d'erreurs, difficile à la fois valider et déboguer et ne peut pas être adapté facilement à la charge de travail et les modifications du système. En outre, dans un système de disque

où ressources sont réparties entre plusieurs machines, la configuration manuelle est encore plus critique et implique de nombreux paramètres à être affinée et régulièrement cochée.

Basé sur la taille, nous pensons que, en raison de l'absence de travaux de recherche sur la façon d'utiliser ordonnanceurs dans des environnements réalistes, les systèmes actuels sont vraiment absents de la possibilité d'utiliser de meilleures politiques de planification pour éviter les inconvénients des aveugles politiques de taille et de réduire considérablement les problèmes liés à l'emploi configurations.

Comme nous allons le montrer dans cette thèse, non seulement les politiques d'ordonnancement basés sur la taille DISC sont une solution viable, mais ils jouent aussi très bien dans de nombreux scénarios de niveau de production. La motivation principale de cette thèse est de montrer que basée sur la taille de la programmation est de montrer en concevant et en mise en œuvre de la Foire Protocole Hadoop de séjour (de HFSP), une basée sur la taille planification politique pour un système réel et complexe comme Hadoop, et par fournir une étude approfondie de l'architecture de l'ordonnanceur et sa performance nécessaire de comprendre comment et pourquoi cela fonctionne.

Thesis Statement: Une politique d'ordonnancement préemptif basé sur la taille des systèmes de DISC peut être à la fois efficace et équitable, et il peut améliorer les performances en ce qui concerne aux politiques de planification actuels état-of-the-art.

A.2 Challenges

Mettre en œuvre une politique d'ordonnancement basé sur la taille des systèmes de DISQUE soulève beaucoup défis; dans ce qui suit, nous résumons les plus importants.

Tailles d'emploi sont inconnues: ordonnanceurs fondés sur la taille, en dépit de leur supérieur performance, sont très rarement déployé dans la pratique. Une des principales raisons est que, dans systèmes réels, la taille de l'emploi sont presque jamais connus a priori. Lors de la conception un ordonnanceur basé sur la taille, donc, le premier problème est de savoir comment obtenir un emploi taille de sorte qu'il peut être utilisé par le planificateur pour trier les tâches. Nous supposons que le informations sur la taille de l'emploi n'est pas donné (*e.g.*, fourni par l'utilisateur): planificateur doit trouver un moyen de déterminer la taille d'un emploi une fois le travail est arrivé, *i.e.*, alors que la tâche est en attente et d'autres travaux sont en cours d'exécution. Depuis la mesure de la taille de l'emploi peut être imprécise, nous devons faire face à une autre problème: les erreurs d'estimation.

Les erreurs d'estimation: dans un système réel, il n'est pas possible de déterminer la taille exacte d'un emploi. Au lieu de cela, il est souvent possible de prévoir estimations de la taille des tâches. Cela signifie que l'ordonnanceur a besoin d' faire face aux dimensions erronées.
Peut-être étonnamment, même en considérant le cas simple d'un seul serveur, basées taille très peu de travaux dans la littérature ont abordé le problème de la la planification de renseignements inexacts de la taille de l'emploi. En outre, ces quelques œuvres donnent des résultats quelque peu pessimistes, ce qui suggère que la planification basée sur la taille est efficace seulement lorsque l'erreur de la taille estimée est faible. Néanmoins, ces études portent sur une famille restreinte de la charge de travail, et leurs réponses ne sont pas exhaustives.

Lors de la conception d'un ordonnanceur basé sur la taille, par conséquent, nous avons besoin de comprendre l'impact des erreurs d'estimation de la taille sur la performance globale, par exemple, si le planificateur est en mesure de faire suffisamment de bonnes décisions malgré des informations imprécises sur la taille du travail. Au meilleur de notre connaissance, aucune étude abordé la conception de techniques d'ordonnancement basés sur la taille qui sont explicitement conçu dans le but de faire face à des erreurs dans la taille de l'emploi informations, pas même dans le cas d'un serveur unique. Pour cette raison, nous devons d'abord comprendre l'impact des erreurs dans le cas de serveur unique, puis prolonger la leçon à tirer de cette configuration de base pour les systèmes de DISQUE plus complexes.

Préemption d'emploi: les politiques de planification qui ont les meilleures performances en terme d'équité et le temps moyen de séjour sont de préemption. En effet, de nombreux travaux montrent qu'une politique de planification préventive est souvent meilleur que ce n'est pas préemptive contre-partie. Cela peut être compris intuitivement considérer le cas où une nouvelle petit travail arrive alors que le système est au service d'un grand travail: sans préemption, le petit travail doit attendre jusqu'à ce que le grand travail est complètement servi.
Dans de nombreux systèmes réels, tels que Hadoop, emploi préemption est souvent absent ou partiellement mises en œuvre. Quand il ya une préemption primitif mis en œuvre, il a généralement les inconvénients et les limitations empêchant d'être efficacement utilisés parce que ses inconvénients sont pire que ses avantages.
Lors de la conception d'un ordonnanceur basé sur la taille, par conséquent, nous avons besoin de comprendre comment de fournir des primitives efficaces pour la préemption, et comment mettre en œuvre ces primitives dans le système actuel de manière transparente.

Emploi famine: les politiques de planification fondés sur la taille comme SRPT peuvent causer famine quand un travail est maintenu d'obtenir les

ressources en arrivant en continu petits travaux. Certains travaux ont abordé ce problème dans le cas d'un serveur unique la file d'attente en proposant des politiques qui rétablissent l'équité; Néanmoins, de telles politiques n'ont jamais été mis en œuvre dans des systèmes réels (même pas de systèmes de serveur unique).

A.3 Contributions et Organisation de thèse

La principale contribution de cette thèse est le Protocole de séjour Hadoop Fair, basée sur la taille de politique d'ordonnancement pour Hadoop qui est à la fois juste et efficace. Parce que HFSP repose sur l'estimation de la taille de l'emploi et que l'estimation de la taille est si importante pour la planification basée sur la taille, en général, le premier chapitre (Chapitre 3) est consacré à l'étude de l'impact de la taille estimée sur les politiques de planification existants dans certains scénarios simples. Avec ce travail en arrière-plan, on procède ensuite à définir le programme au (chapitre 4) et expérimentalement l'évaluer (chapitre 5). Dans le HFSP évaluation Chapitre, nous fournissons également une étude de HFSP avec la préemption primitives actuellement disponibles dans Hadoop, et nous suggérons qu'un nouveau préemption primitif doit être mis en oeuvre de surmonter les problèmes primitives Hadoop de préemption. Le prochain chapitre (Chapitre 6) est ensuite consacré à une nouvelle préventive primitive résout ces problèmes.
The rest of this section is dedicated to an overview of our contributions.

A.3.1 Planification de Taille-base avec taille estimée

Avant de commencer la conception du protocole HFSP, nous avons besoin de comprendre si l'information imprécise sur les tailles peut avoir un impact significatif sur le choix de programmation. Depuis ce problème n'a pas été bien étudiée dans le littérature, nous analysons ce à partir d'une configuration de base: la seule cas de serveur. Même si le cas de serveur unique peut sembler simple, il fournit beaucoup des points de vue qui sont utiles pour comprendre le comportement du système. Chapitre 3 est donc

consacrée à l'étude de politiques d'ordonnancement qui utilisent taille estimée. Les connaissances acquises dans ce Chapitre sera utilisé pour conduire la conception d'un ordonnanceur plus complexe DISC systèmes tels que HFSP.
Dans la première partie du chapitre, nous étudions le problème de la planification en présence des erreurs d'estimation et de fournir un aperçu du courant état-of-the-art à la fois pour la performance basée sur la taille et aveugle aux planificateurs de taille.
Nous décrivons ensuite l'impact des erreurs d'estimation sur les politiques de planification par définir SRPTE et FSPE, deux variantes de programmation fondé sur la taille bien connu politiques pour les files d'attente de serveur unique qui travaillent avec des tailles estimées. Il existe deux types d'erreurs qui peuvent être fait si la taille de l'emploi est estimé: il peut être surestimée ou sous-estimée si sa taille est estimée, respectivement, plus ou plus petit que sa taille réelle. Ces deux types d'erreurs ont deux impacts complètement différents sur les politiques et basés sur la taille nécessitent des méthodes différentes stratégies pour faire face avec eux. En particulier, alors que les emplois dont la taille surestimé retarder seulement eux-mêmes, avec des emplois sous-estimé la taille peuvent potentiellement retarder tout les emplois dans la file d'attente. Entre les deux, le second type d'erreurs comporte l' plus d'impact sur la politique d'ordonnancement et peut conduire à de très mauvaises performances.
La prochaine partie du chapitre est consacrée à définir FSPE + PS, une programmation basée sur la taille politique de file d'attente de serveur unique qui est tolérante aux erreurs d'estimation. Notre évaluation de simulation, qui considère à la fois la variation de la composition de la charge de travail et à l'erreur d'estimation, montre deux résultats principaux:

- en dépit de leurs problèmes avec des emplois avec des tailles sous-estimé, à la fois SRPTE et FSPE ont d'excellentes performances dans de nombreux cas et ils sont bons choix pour une mise en œuvre d'un véritable ordonnanceur.

- FSPE + PS est encore supérieure et offre de meilleures performances, même dans les cas extrêmes à la fois la composition de la charge de travail et de l'erreur.

Le résultat de ce chapitre est que l'ordonnancement basé sur la taille, même en présence de informations imprécises, est une politique réaliste qui peut être mis en œuvre en temps réel systèmes.

A.3.2 The Hadoop Fair Sojourn Protocol

Dans Hadoop, un travail est composé de tâches, tâches et peut être exécuté en parallèle. Comme nous l'avons vont montrer au chapitre 4, la plupart des fois les tâches de la même travail ont des tailles très similaires. Quelques tâches peuvent être exécutées dans le système et puis, sur la base de leurs performances, il est possible de déduire la taille estimée d'un travail avec une petite erreur.
Nous avons mis l'ordonnanceur Hadoop Foire Protocole de séjour, une politique d'ordonnancement pour les systèmes de DISC sur la base de la politique de la FSPE défini pour une file d'attente à un seul serveur. Bien que la principale contribution de ce chapitre est un à part entière planificateur pour un système distribué, les choix architecturaux qui conduisent à ce planning sont pas moins important. En effet, l'adaptation des politiques de planification défini pour le cas de serveur unique (comme FSPE) à un système réel comme Hadoop soulève beaucoup contesté qui doit être abordée.
HFSP se compose de deux éléments principaux, qui sont décrits en détail dans le chapitre 4: le module d'estimation et le module de vieillissement. Le module d'estimation est le composant qui estime la taille des tâches. Il permet d'abord une estimation très approximative quand un travail est soumis, puis met à jour la taille à un une plus précis sur la base de la performance d'un sous-ensemble des tâches. ce stratégie de tailles d'estimation est conçu autour des systèmes de disque, dans lequel un travail est composé par des tâches, et conduit à de très bons estimations à la fin.

Le module de vieillissement est le composant qui permet d'éviter la famine travail en appliquant ce qui est appelé "vieillissement" à un emploi. HFSP ne prend pas des décisions uniquement sur la base de la taille, mais aussi basé sur combien de temps le travail reste dans la file d'attente. De cette façon, même relativement gros travail finira par obtenir des ressources, ce qui résout le problème de la famine. Pour les travaux d'âge, HFSP simule Max-Min Processeur Partage d'un cluster virtuel avec les mêmes caractéristiques que le vrai.
HFSP est un ordonnanceur préemptif. Hadoop fournit deux options possibles qui peuvent être utilisés pour le travail préemption. La première consiste à arrêter l'exécution des tâches appartenant à la tâche à préempté par les tuer - nous appelons cette stratégie Kill. La deuxième option consiste à attendre pour chaque tâche et pour terminer, une fois que les ressources seront disponibles sur une tâche en fonction des travaux par les assigne système tels ressources à la tâche préemptée - nous appelons cette stratégie Wait [1]. La dernière partie du chapitre 4 est consacrée à l'analyse des avantages et des inconvénients de ces deux approches.

Le chapitre suivant, chapitre 5, est dédié à un dispositif expérimental évaluation des HFSP. L'évaluation de la mise en œuvre réelle d'une politique de planification pour une système tels que Hadoop est une tâche très complexe. Nous avons décidé de valider de manière expérimentale pour pouvoir comparer HFSP pour le plus utilisé ordonnanceurs pour Hadoop, qui sont la Foire - une mise en œuvre de la discipline PS - et les planificateurs de FIFO. le principal raison est que nous voulons mettre l'accent sur le fait que HFSP est un véritable ordonnanceur qui surpasse ordonnanceurs industrielle prêts. Nos résultats montrent que HFSP est toujours mieux que les deux ordonnanceurs pour la deux paramètres observés - l'équité et le temps de réponse (ou séjour) signifient. nous a également observé que la nature séquentielle de HFSP conduit à plus petite charge de cluster et une

[1] Même si cette approche peut ne pas sembler une préemption de l'emploi, nous devons considérer que l'emploi peut exiger plus de tâches que le système peut fournir, par conséquent, quand une tâche terminée, le système réaffecte les ressources à la tâche en cours d'exécution afin qu'il puisse procéder; avec le Wait primitive, à la place, le système attribue les ressources au travail de préemption.

meilleure “absorption” de salves de soumissions de tâches. La charge de cluster et rafale la tolérance font HFSP capable de traiter plus petit groupe que leurs homologues pour la même charge de travail, conduisant à moins de coûts de munitions et de meilleure ressource utilisation.
La partie suivante de l’évaluation Chapitre est consacré à des erreurs d’estimation. Les résultats confirment que les erreurs faites par estimation de la taille de l’emploi avec notre module d’estimation est assez petit pour être en mesure pour justifier l’utilisation d’ordonnanceurs à base de taille avec imparfait informations de taille de travail.
La dernière partie de ce chapitre présente une évaluation expérimentale de HFSP avec Kill mécanisme de préemption permis - le comportement par défaut de HFSP est d’utiliser Wait - avec des résultats très intéressants. par rapport à Wait, Kill est un bon moyen de parvenir à une meilleure équité pour tous les emplois et des temps de réponse plus petites pour les petites et moyennes d’emplois, mais il a un impact sur la plus grands emplois.

A.3.3 OS-Assisted Task Preemption

Chapitre 6 est dédiée à la conception d’une nouvelle tâche préemption autre primitive de Wait et Kill (les primitives disponibles pour Hadoop), que nous avons nommé tâche suspension. Notre solution fonctionne à l’ Systèmes d’exploitation (OS) niveau: en effet, les tâches ne sont que des processus OS, donc nous pouvons contrôler l’exécution des travaux en utilisant les primitives de Suspend et Resume. ce approche est totalement transparente pour les utilisateurs et exploite le système fonctionnalités. Notre mécanisme de préemption peut être utilisé dans la production même des tâches stateful, *i.e.*, les tâches qui ont un état et des besoins cet état de continuer le calcul.
Après la définition de notre mécanisme, nous comparons les résultats obtenus à l’aide de notre Suspend , l’ Kill et le Wait primitives. Nos résultats montrent que, dans presque tous les cas, à l’exception de cas d’angle lorsque le travail est juste commencé ou a presque fini, notre Suspend primitive effectue toujours mieux que les deux autres primitives.

A.3.4 Conclusion and Perspectives

Le dernier chapitre de la thèse résume les principaux résultats que nous avons obtenus. la conception d'un planificateur pour un système complexe comme Hadoop soulève de nombreuses questions, et nous abordons beaucoup d'entre eux dans notre travail. Néanmoins, le système lui-même est évolution, et son adoption généralisée introduit différentes fonctionnalités que on rend le système de plus en plus complexe. Dans la dernière partie du chapitre, nous fournir un ensemble de directions futures possibles qui tiennent compte de cette évolution système complexe.

Conclusion

Cette thèse a porté sur planification des politiques fondés sur la taille pour les systèmes réels. tandis que cette thèse propose différentes contributions à la théorie de l'ordonnancement existant et la pratique, la contribution principale et fondamentale de notre travail est la définition et la mise en œuvre de HFSP, un planificateur basé sur la taille pour Hadoop qui donne de meilleurs résultats que les planificateurs, industrielle prêts existants par l'exploitation de la taille des tâches. Pour parvenir à ce résultat, de nombreux défis ont été traités et résolus.
Le problème le plus important de la programmation basée sur la taille des systèmes réels est l' manque de connaissance des emplois tailles: quand l'information sur la taille de l'emploi est manquant, notre idée est de estimer la taille d'emploi alors qu'ils sont en cours d'exécution. Estimation de la taille d'un emploi introduit une erreur: chapitre 3 a été consacrée précisément à étudier l'impact de ces erreurs dans le décisions d'ordonnancement. Nous avons présenté SRPTE, FSPE et FSPE+PS, trois planificateurs qui travaillent avec des tailles estimées, et nous avons montré que tous les trois peuvent tolérer Estimations des erreurs sous certaines conditions d'intérêt pratique. Chaque système qui répond à ces conditions est un candidat à la la mise en œuvre de l'une des politiques d'ordonnancement basés sur la taille au-dessus.

Dans le chapitre 4, nous avons présenté la conception et l'architecture de HFSP, un ordonnanceur basé sur la taille pour Hadoop basé sur FSPE. HFSP soulevé de nombreux défis, par exemple, comment estimer la taille, ou la façon de faire face au vieillissement de travail, ce qui nous résolu à créer un ordonnanceur préemptif basé sur la taille de travail. HFSP conduit à plus petit temps de réponse moyen par rapport à ordonnanceurs existants, tout en être équitable entre les emplois. Nos résultats ont montré que HFSP peut traiter avec des éclats d'emplois mieux que le Salon Scheduler, et donc il peut être utilisé dans les petits groupes.
Bien que notre travail est pour Hadoop, nous croyons que HFSP peut être porté à d'autres biens systèmes tels que Spark ou naïade ou même d'autres systèmes non-disque.
Enfin, dans le chapitre 6, nous avons renforcé les Hadoop MapReduce pour fournir une nouveau mécanisme de préemption de la tâche, appelé Suspend, qui est aussi efficace que le Kill préemption, mais évite tous les problèmes que Kill ne possède. Nous avons également discuté lors Suspend doit être utilisé, et dans lequel le scénario Kill doit être utilisé à la place.
Nous croyons que l'approche de planification basée sur la taille proposée dans ce thèse pourrait être un point de départ pour de futures fonctionne avec tous les avantages que la théorie de l'ordonnancement a montré dans le passé.
Alors que la thèse répond à des questions fondamentales, il soulève inévitablement de nombreux nouveaux ceux qui nécessiteront des études complémentaires dans le domaine.

A.4 Travaux Futurs

Dans cette section, nous présentons quelques-uns des plus idées prometteuses, les deux premières liées à l'estimation de la taille de l'emploi, et la dernière connexes à la tâche suspension.

Complex Jobs

HFSP est un programmateur pour MapReduce et travaille en considérant chaque emploi indépendant des autres. Bien que ce soit la façon dont un planificateur de tâches standard pour MapReduce fonctionne, une tendance de l'élevage dans l'industrie est d'éviter de définir des emplois à Niveau de MapReduce et utiliser les bibliothèques de niveau supérieur, tels que Cascading [115], Scalding [116], Cascalog [117] et Scoobi [118], or languages, such as Pig [105] et Hive [104]. La principale caractéristique des bibliothèques et des langages de haut niveau est de faire abstraction de MapReduce et laissez-le utilisateur de définir son programme de MapReduce dans une voie plus facile. Par exemple, la ruche permet à l' utilisateur de définir un programme de MapReduce dans SQL. Ces " programmes de haut niveau " sont alors compilé dans un ensemble de travaux MapReduce et sont soumis en tant que chaîne de l'emploi, où chaque tâche de la chaîne peut commencer à travailler qu'après la précédente complète. Alors que pour le moteur de MapReduce cette chaîne d'emplois est considérée comme beaucoup emplois indépendants, le planificateur doit prendre en compte que les emplois font partie de la même chaîne et doit les considérer comme un gros travail complexe.
L'évaluation de la taille d'un travail complexe est en effet différente de l'évaluation de la taille d'un travail normal. Haut niveau langues et les bibliothèques offrent des conseils sur les performances de certaines opérations. Ces conseils sont généralement utilisés pour faire l'optimisation basée sur les coûts, mais peuvent également être utilisés pour avoir une meilleure idée de la taille d'un travail complexe. Nous croyons qu'il s'agit d'un direction de la recherche pour notre travail.

Recurring Jobs

soumis à la grappe. Par exemple, Agarwal *et al.* [52] spectacle que, dans leur système de production, les emplois récurrents sont le 40 % de l'ensemble des emplois soumis dans leur groupe. À notre avis ce genre d'emplois peut contribuer à une version avancée de HFSP, dans lequel la

taille des tâches sont estimés sur la base non seulement de leurs tailles, mais également sur les dernières pistes. Nous rappelons que le sortie de l'exécution d'un travail est sa taille, qui est exactement ce qu'est une base de taille politique d'ordonnancement doit. Cela s'ajoute le un ordonnanceur peut obtenir une taille très précise pour eux en observant toutes les pistes dernières et puis construire un profil de poste qui regroupe toutes les informations sur le travail. Alors qu'il est facile de recueillir les dimensions réelles d'un travail de ses courses passées, il ya beaucoup de défis posés par cette technique. Tout d'abord, et surtout, la taille du travail dépend de l'entrée. Si l'entrée ne change pas entre deux pistes, ensuite la taille d'un travail est susceptible d'être à peu près identique à la précédente - certaines variations peuvent dé en raison de données localité. Au contraire, si l'entrée est différente, alors l' seule façon d'utiliser les informations sur les pistes dernières est de comprendre la caractéristiques de l'entrée, par exemple, la distribution des clés, et ensuite adapter le passé va information pour la nouvelle entrée.

Extension ordonnanceurs avec tâche suspension

Dans le chapitre 5.4, nous avons montré la performance de HFSP avec ou sans le mécanisme de préemption intégré Hadoop activé. Dans le chapitre 6, nous avons présenté un nouveau mécanisme tâche de préemption appelé SUSPEND qui vise à avoir les mêmes avantages que le haut-KILL primitive sans les inconvénients. Lorsque la suspension d'une tâche d'un emploi, nous avons besoin de sélectionner, parmi les tâches possibles, celles selon certains métrique [2]. Dans la section 4.6 nous avons adopté une stratégie qui est inspiré par l'approche proposée lors de l'utilisation du KILL primitive. Néanmoins, il pourrait y avoir d'autres stratégies, adaptées aux caractéristiques de la SUSPEND primitive, qui peut se traduire par une utilisation plus efficace de la ressources.

[2]Considérons le cas dans lequel, tout en un grand travail avec de nombreuses tâches est en marche, un petit travail qui ne nécessite que quelques tâches arriver: seule une fraction de la tâche de la grande tâche devrait être suspendu à donner les moyens de la petite emploi, tandis que les autres tâches peuvent continuer à travailler.

Un autre aspect intéressant est ce qu'on appelle le Groupe Localité problème: lorsqu'une tâche est mis en suspension, il peut être reprise seulement sur la même machine. Ce problème peut conduire à une mauvaise performance si, par exemple, certaines ressources sont disponibles (par exemple, une autre tâche achevée) sur une machine différente de celle dans laquelle la tâche a été suspendu. La tâche ne peut pas être repris sur l'autre machine, et donc les ressources peuvent être gaspillées. il existe différentes approches pour résoudre ce problème - des approches qui devrait être étudiée en profondeur pour comprendre leur avantages et des inconvénients. Une stratégie qui pourrait être appliqué sans modification Hadoop est d'utiliser l'exécution spéculative, dire, pour exécuter une copie de la tâche suspendue sur la machine avec les ressources disponibles. dans ce cas, il ya deux possibilités: i) la copie spéculative termine le travail et la tâche suspendue peut être tué ii) la tâche suspendue est reprise et complète dans la machine où il a été suspendu et la copie spéculative est tué. Bien que l'exécution spéculative est une stratégie simple, qu'il est fondamentalement consiste à exécuter la tâche plus d'une fois, et par conséquent, les avantages de l'utilisation la suspension peut être perdu. En outre, les données de sortie de la tâche est partiellement dupliqué, et cette fois les déchets disque et réseau des ressources.
Une deuxième stratégie pourrait être la modification de Hadoop pour soutenir tâche check-pointage: quand une tâche est suspendue et les ressources sont libérées sur une autre machine, d'autre part Machine Hadoop commence une tâche spéciale qui continue le travail de la précédente un, sans avoir à refaire tout le travail accompli par le groupe de suspension. Fondamentalement, Hadoop considère la tâche comme divisé en deux tâches, et les traite de façon indépendante. ce stratégie a le problème que, si la tâche a un état interne, comme un accumulateur, cet état doit être migré vers la nouvelle machine.
Comme note finale, nous soulignons que tous les planificateurs de préemption dans Hadoop, comme comme la Foire ou les ordonnanceurs de capacité, pourrait être étendu avec notre suspension mécanisme pour améliorer leur performance.