Adaptive Failure-Aware Scheduling for Hadoop

Mbarka Soualhia

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy (Electrical and Computer Engineering) at

Concordia University

Montréal, Québec, Canada

February 2018

© Mbarka Soualhia, 2018

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By: **Mbarka Soualhia**

Entitled: **Adaptive Failure-Aware Scheduling for Hadoop**

and submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Dr. Rene Witte

______________________ Dr. Marin Litoiu

______________________ Dr. Juergen Rilling

______________________ Dr. Samar Abdi

______________________ Dr. Nawwaf Kharma

______________________ Dr. Sofiène Tahar

______________________ Dr. Foutse Khomh

Approved by ______________________

Dr. William E. Lynch, Chair of the ECE Department

March 8, 2018______________________

Dr. Amir Asif, Dean, Faculty of Engineering and Computer Science

ABSTRACT

Adaptive Failure-Aware Scheduling for Hadoop

Mbarka Soualhia, Ph.D.

Concordia University, 2018

Given the dynamic nature of cloud environments, failures are the norm rather than the exception in data centers powering cloud frameworks. Despite the diversity of integrated recovery mechanisms in cloud frameworks, their schedulers still generate poor scheduling decisions leading to tasks' failures due to unforeseen events such as unpredicted demands of services or hardware outages. Traditionally, simulation and analytical modeling have been widely used to analyze the impact of the scheduling decisions on the failures rates. However, they cannot provide accurate results and exhaustive coverage of the cloud systems especially when failures occur. In this thesis, we present new approaches for modeling and verifying an adaptive failure-aware scheduling algorithm for Hadoop to early detect these failures and to reschedule tasks according to changes in the cloud. Hadoop is the framework of choice on many off-the-shelf clusters in the cloud to process data-intensive applications by efficiently running them across distributed multiple machines. The proposed scheduling algorithm for Hadoop relies on predictions made by machine learning algorithms trained on previously executed tasks and data collected from the Hadoop environment. To further improve Hadoop scheduling decisions on the fly, we use reinforcement learning techniques to select an appropriate scheduling action for a scheduled task. Furthermore, we propose an adaptive algorithm to dynamically detect failures of nodes in Hadoop.

We implement the above approaches in *ATLAS*: an AdapTive Failure-Aware Scheduling algorithm that can be built on top of existing Hadoop schedulers. To illustrate the usefulness and benefits of ATLAS, we conduct a large empirical study on a Hadoop cluster deployed on Amazon Elastic MapReduce (EMR) to compare the performance of ATLAS to those of three Hadoop scheduling algorithms (FIFO, Fair, and Capacity). Results show that ATLAS outperforms these scheduling algorithms in terms of failures' rates, execution times, and resources utilization. Finally, we propose a new methodology to formally identify the impact of the scheduling decisions of Hadoop on the failures rates. We use model checking to verify some of the most important scheduling properties in Hadoop (*i.e.*, schedulability, resources-deadlock freeness, and fairness) and provide possible strategies to avoid their occurrences in ATLAS. The formal verification of the Hadoop scheduler allows to identify more tasks failures and hence reduce the number of failures in ATLAS.

In Loving Memory of my Mother,

To my Father, my Husband and Baby

ACKNOWLEDGEMENTS

First and foremost, I owe my deepest gratitude to my supervisors, Dr. Sofiène Tahar and Dr. Foutse Khomh, for their guidance and encouragements throughout my Ph.D. studies. I have learned so much from their deep insights about research and strong expertise. I am deeply grateful to the Tunisian Ministry of Higher Education and Scientific Research and the University Mission of Tunisia in North America (MUTAN) for granting me a scholarship to conduct my doctoral studies at Concordia University. I am thankful to Dr. Rilling, Dr. Abdi and Dr. Kharma for serving on my doctoral advisory committee. Their constructive feedback and comments at various stages have been significantly useful in shaping my thesis to completion. I am very pleased that Dr. Marin Litoiu has accepted to be my external Ph.D. thesis examiner.

My sincere thanks go to all my friends in the Hardware Verification Group (HVG) and the SoftWare Analytics and Technologies (SWAT) laboratories for their support and help. I am also thankful to my best friends, Hadhami, Sabrine, and Foued for their support and encouragements. I am deeply grateful to my family for all the love and support they have provided me over the years. Finally, I would like to thank the love of my life, Aymen, for his unconditional love and support. At the end of my Ph.D., I got the most precious gift I could ever ask for; my first baby, Touty. The existence of my little baby has fulfilled my life with joy and has motivated me to work harder and to succeed. Nothing would be possible without you.

TABLE OF CONTENTS

LIST OF TABLES xii
LIST OF FIGURES xiv
LIST OF ACRONYMS xvii

1 Introduction **1**
1.1 Hadoop-MapReduce 1
1.2 Motivation 4
1.3 Proposed Methodology 8
1.4 Thesis Contributions 10
1.5 Thesis Organization 11

2 Related Work **13**
2.1 Failure-aware Scheduling in Hadoop 13
2.2 Adaptive Scheduling in Hadoop 17
2.3 Formal Verification of Hadoop and Cloud 20
2.3.1 Formal Analysis of Hadoop System 20
2.3.2 Formal Analysis of Cloud System 22
2.4 Summary 23

3 Failures Detection and Adaptive Scheduling **25**
3.1 Limitation of Current Schedulers 26
3.1.1 Tasks Failures Detection 26
3.1.2 TaskTrackers Failures Detection 29
3.2 Task Failure Detection 31
3.2.1 Task Failure Detection Methodology 31

General Overview 31
Tasks/Jobs Attributes Extraction 32
Task/Job Failure Profiling 33
Task Failure Prediction 34
3.2.2 Task Failure Detection Evaluation 37
Experimental Design 37
Experimental Results 38
3.3 Adaptive Scheduling 41
3.3.1 Adaptive Scheduling Methodology 42
General Overview 42
Scheduling Policies Modeling 43
Reinforcement Algorithms Training 44
3.3.2 Adaptive Scheduling Evaluation 49
Experimental Design 49
Experimental Results 51
3.4 TaskTracker Failure Detection 54
3.4.1 TaskTracker Failure Detection Methodology 54
General Overview 54
Heartbeats Analysis and Estimation 55
TaskTracker Failure Detection 56
Chen Failure Detector 56
Bertier Failure Detector 57
ϕ Failure Detector 57
Self-Tuning Failure Detector 58
3.4.2 TaskTracker Failure Detection Evaluation 60

Experimental Design 61
Experimental Results 62
3.5 Summary 66

4 ATLAS: AdapTive faiLure-Aware Scheduling 67

4.1 ATLAS Implementation 68
4.2 ATLAS Evaluation 70
4.2.1 Experimental Design 72
Cluster 72
Workload 73
Injected Failures 73
Collected Performance Metrics 74
4.2.2 Performance Analysis Results 74
Number of Finished Jobs/Tasks 75
Number of Failed Jobs/Tasks 78
Execution Times of Jobs/Tasks 80
Resources Utilization of Jobs/Tasks 81
Long-Execution of Jobs/Tasks 82
4.2.3 Scalability Analysis Results 85
4.3 Threats to Validity 89
4.3.1 Construct Validity 90
4.3.2 Internal Validity 90
4.3.3 Conclusion Validity 91
4.3.4 Reliability Validity 92
4.3.5 External Validity 93
4.4 Summary 93

5 Formal Verification of Hadoop **95**
5.1 Preliminaries . . . 96
5.2 Formal Verification Methodology . . . 97
5.2.1 General Overview . . . 97
5.2.2 Hadoop Scheduler Formal Model . . . 98
5.2.3 Hadoop Scheduler Properties . . . 102
5.2.4 Quantitative Failures Analysis . . . 104
5.2.5 Qualitative Failures Analysis . . . 105
5.3 Formal Verification Evaluation . . . 105
5.3.1 Experimental Design . . . 105
5.3.2 Experimental Results . . . 107
Properties Verification and Scalability Analysis . . . 107
Quantitative Failures Analysis . . . 111
Qualitative Failures Analysis . . . 113
5.4 Formal Verification of Hadoop and Refinement of ATLAS . . . 116
5.4.1 Experimental Design . . . 116
5.4.2 Experimental Results . . . 117
5.5 Summary . . . 119

6 Conclusions and Future Work **120**
6.1 Conclusions . . . 120
6.2 Future Work . . . 123

Bibliography **127**

Biography **141**

List of Tables

3.1 Jobs and Tasks Attributes 34
3.2 Accuracy, Precision, Recall (%) and Time (ms): FIFO Scheduler . . . 39
3.3 Accuracy, Precision, Recall (%) and Time (ms): Fair Scheduler 40
3.4 Accuracy, Precision, Recall (%) and Time (ms): Capacity Scheduler . 41
3.5 Normalized Values of Wrong Failure Detection Rate of TT 65

4.1 Amazon EC2 Instance Specifications [1] 72
4.2 Resources Utilization of the FIFO Scheduler 83
4.3 Resources Utilization of the Fair Scheduler 84
4.4 Resources Utilization of the Capacity Scheduler 85
4.5 Benefits of ATLAS Components 86
4.6 Reduction Rates (%) of Proposed Algorithms (30,000 Hadoop Jobs) . . 87
4.7 Reduction Rates (%) of Proposed Algorithms (60,000 Hadoop Jobs) . . 87
4.8 Reduction Rates (%) of Proposed Algorithms (90,000 Hadoop Jobs) . . 87
4.9 Worst-Case Execution Time (Seconds) in ATLAS (30,000 Jobs) 88
4.10 Worst-Case Execution Time (Seconds) in ATLAS (60,000 Jobs) 88
4.11 Worst-Case Execution Time (Seconds) in ATLAS (90,000 Jobs) 89

5.1 Verification Results: Trace for the First Month (1,772,144 Tasks) . . . 108
5.2 Verification Results: Trace for the Second Month (476,034 Tasks) . . . 109

5.3 Verification Results: Trace for the 1-6 Months (4,006,512 Tasks) 110

5.4 Coverage Results(%): Trace for the First Month (1,772,144 Tasks) . . . 112

List of Figures

1.1 An Overview of Job Execution in MapReduce [2] 3

1.2 Overview of the Proposed Methodology 9

3.1 Example of Hadoop Job Failure 28

3.2 TaskTracker Failure Detection Model in Hadoop Framework 30

3.3 Task Failure Detection Methodology 32

3.4 Adaptive Scheduling Methodology 43

3.5 Life Cycle of a Task 47

3.6 Number of Explored Policies 51

3.7 Policy Success Rate 52

3.8 TaskTracker Failure Detection Methodology 55

3.9 Detection Time under 30% Failure rate 63

3.10 Detection Time under 50% Failure rate 64

4.1 Finished Hadoop Jobs 76

4.2 Finished Map Tasks 76

4.3 Finished Reduce Tasks 77

4.4 Failed Hadoop Jobs 79

4.5 Failed Map Tasks 79

4.6 Failed Reduce Tasks 80

4.7 Exection Time of Jobs 81

4.8 Exection Time of Tasks 82

4.9 Number of Failed Job over 3 Days 83

4.10 Number of Failed Task over 3 Days 86

5.1 Formal Analysis of Hadoop Schedulers Methodology 98

5.2 Impact of Adding Resources on Failures Rate 115

5.3 Impact of Verification Guidelines on Failed Hadoop Jobs 118

5.4 Impact of Verification Guidelines on Failed Map Tasks 118

5.5 Impact of Verification Guidelines on Failed Reduce Tasks 119

LIST OF ACRONYMS

ATLAS	Adaptive faiLure-Aware Scheduling
COSHH	Classification, and Optimization based Scheduler for Heterogeneous Hadoop
CPU	Central Processing Unit
CREST	Combination Re-Execution Scheduling Technology
CSP	Communicating Sequential Processes
CTree	Conditional Tree
DF	Detected Failure
EBS	Elastic Block Store
EMR	Elastic MapReduce
ESAMR	Enhanced Self-Adaptive MapReduce scheduling
ESP	Encrypted Storage Protocols
FD	Failure Detector
FIFO	First In First Out
FN	False Negative
FP	False Positive
FRESH	FaiR and Efficient slot configuration and Scheduling algorithm for Hadoop
FV	Formal Verification
GB	Gigabyte
GiB	Gibibyte
GLM	General Linear Model
HDFS	Hadoop Distributed File System

HLF	Highest Level First
IF	Intermediate Format
IT	Information Technology
JT	JobTracker
LATE	Longest Approximate Time End
LP	Linear Programming
LPF	Longest Path First
LTL	Linear Temporal Logic
MCP	Maximum Cost Performance
MDP	Markovian Decision process
MR	Mistake Rate
PAT	Process Analysis Toolkit
QoS	Quality of Service
RAFT	Recovery Algorithm for Fast-Tracking
RAM	Random Access Memory
RF	Random Forest
SAMR	Self-Adaptive MapReduce scheduling
SARS	Self-Adaptive Reduce Start time
SARSA	State-Action-Reward-State-Action
SFD	Self-tuning Failure Detector
SLA	Service Level Agreement
SLR	Systematic Literature Review
TD	Temporal Difference
TN	True Negative
TP	True Positive

TT	TaskTracker
vCPU	Virtual Central Processing Unit
VIF	Variance Inflation Factor
VM	Virtual Machine
WCET	Worst Case Execution Time
WOHA	WOrkflow over HAdoop

Chapter 1

Introduction

In this chapter, we first introduce Hadoop [3], MapReduce [2], and the existing Hadoop schedulers. Then, we present the motivation behind this work followed by our proposed methodology to achieve the main goal of this thesis. Finally, we outline the main contributions and the organization of this thesis.

1.1 Hadoop-MapReduce

Cloud Computing has become fundamental for IT technologies, replacing traditional models, to deliver and manage services over Internet [4]. Customers can access and lease services provided by cloud computing systems through a virtualized environment and pay only costs of the used infrastructure resources. Motivated by the reasonable prices and the good quality of cloud services, the number of cloud users has exponentially increased, resulting in a huge volume of data that can reach up to the gigabytes, terabytes, petabytes, or exabytes levels. As a result, it poses a challenging issue for several large companies to handle this huge amount of data. Many large companies including Google, Facebook, Yahoo or Amazon deploy Hadoop [3] to process these

intensive and huge data in their data centers. Hadoop has been enormously used in several applications ranging from web analytic, web indexing, image and document processing to high-performance scientific computing and social network analysis [5]. Indeed, Hadoop has become the framework of choice on many off-the-shelf clusters in the cloud. It is a simple yet powerful framework for processing large and complex jobs by efficiently running them across distributed multiple machines. Hadoop is the open-source implementation of MapReduce [2] that is a programming model designed to perform parallel processing of large datasets in the cloud using a large number of computers (nodes). MapReduce splits jobs into parallel sub-jobs to be executed on different processing nodes where data is local, instead of sending the data to where the jobs will be executed. This feature is known as data locality [6] in big-data processing frameworks, like Hadoop. In fact, data locality is very important to reduce the time spent to read, write and copy the input data (especially large ones) of jobs through the network compared to execution of non-local-data tasks [6].

A MapReduce job is comprised of map and reduce functions and the input data. The map function is responsible for splitting the input data into a set of intermediate <key, value> pairs. While the reduce function takes the generated values for the same key to produce the corresponding output for each key. The input data denotes the distributed files assigned to the map functions. MapReduce requires a master, a "JobTracker", to control the execution of the job. While "TaskTrackers" are used in MapReduce to control the execution across the mappers (*i.e.*, worker running a map function) and the reducers (*i.e.*, worker running a reduce function) and ensure that their functions are executed and have their corresponding input data as shown in Figure 1.1.

In addition to its MapReduce processing unit, Hadoop has a storage unit called

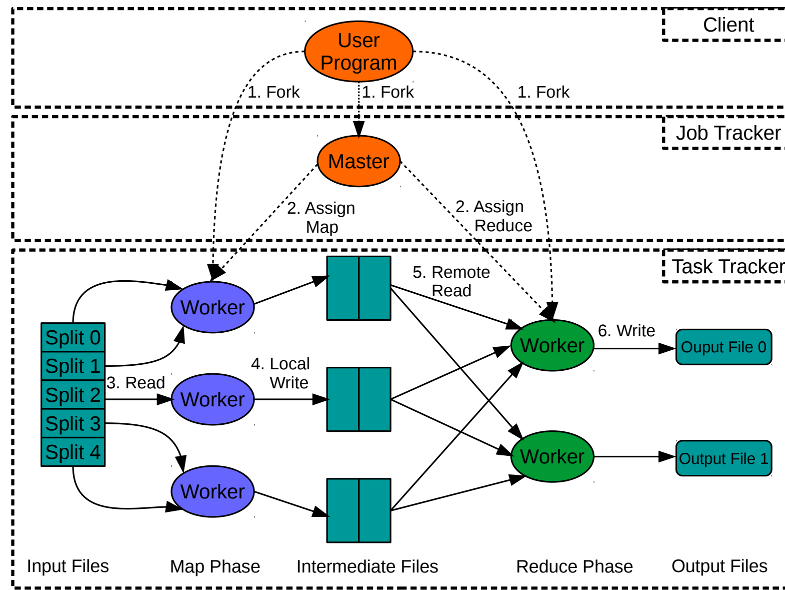

Figure 1.1: An Overview of Job Execution in MapReduce [2]

Hadoop Distributed File System (HDFS). Hadoop follows a master-slave architecture: the master node consists of a JobTracker and a NameNode, while the slave (or worker) consists of a TaskTracker and a DataNode. Hadoop hides all system-level details related to the processing of parallel jobs (such as the distribution to HDFS file store or error handling), allowing developers to write and enhance their parallel programs while focusing only on computation issues rather than the parallelism ones.

The Hadoop framework is equipped with a scheduler responsible for jobs' assignment to the available worker nodes. First In First Out (FIFO) is the default scheduling algorithm used in Hadoop [7]. Facebook and Yahoo! have developed two new schedulers for Hadoop: Fair Scheduler [8] and Capacity Scheduler [9], respectively. The FIFO algorithm grants a full access to the available resources to the scheduled jobs [7]. Facebook proposed the Fair scheduler to ensure a fair distribution of the available resources across the scheduled jobs so that all users of the cluster receive on average the required resources over time. The Fair scheduler can ensure that the minimum

number of slots are assigned to the scheduled jobs to guarantee a high level of service according to the Quality of Service (QoS) requirements [8]. Yahoo! proposed the Capacity scheduler to support multi-user execution within one cluster and to allow a large number of users to fairly execute their jobs over time. This is by dividing the available resources in the cluster across multiple queues given their configurable capacities (*i.e.*, Central Processing Unit (CPU), memory, disk, etc.) [9].

1.2 Motivation

Given the dynamic nature of cloud environments, failures are the norm rather than the exception in data centers powering the cloud. These frequent failures affect the performance of applications running on Hadoop. Studies [10] reveal that a Hadoop cluster can experience more than one thousand individual machine failures and thousands of hard-drive failures on its first year. Furthermore, they show that between 500 and 1000 machines in such cluster can be out of service for up to 6 hours because of power problems, and their recovery time can reach up to 2 days. Indeed, several studies (*e.g.*, [11]) show that a job may encounter multiple failures per day, and they explain the impact of these failures on applications services running on Hadoop [12]. As a result, these failures affect the quality of service delivered to the users. As an example of failures in Hadoop clusters, in 2013, a huge amount of business data, belonging to 5,700 customers of Firstserver Inc., were lost due to the execution of improper operations [13]. Because of invalid configuration changes to their network paths, as another example, in 2013, Amazon Web Services halted their services for approximately 11 hours in one day [14].

Although several failure handling and recovery mechanisms have been proposed to recover from these failures, Hadoop clusters are still experiencing a large number of

failures affecting the quality of service delivered to the users. Dinu et al.[12] analyzed the built-in fault-tolerance mechanisms integrated within the Hadoop framework to recover from failures. More concretely, they evaluated the performance of the Hadoop framework under several types of failures. Overall, they claimed that several tasks failures are due to the lack of sharing information about failures between units responsible for operations' execution in Hadoop, which can lead to poor scheduling decisions. One of these main units is the Hadoop scheduler, which is responsible for tasks and jobs assignment across the available resources. Task scheduling in Hadoop is a crucial problem; the scheduler should satisfy several constraints to guarantee their successful executions and improve the performance of the Hadoop cluster. Indeed, a typical scheduler must distribute received tasks across the available tasks to ensure their successful completions and guarantee that they are finished within their deadlines.

In the event of a node failure, Hadoop is able to restore lost data because it keeps multiple replicas of each data block on different nodes. When this node failure is detected, the Hadoop scheduler reschedules all the running tasks on that node and launches recovery tasks on other nodes with enough resources. Although this re-execution solution is easy to implement, it is not always effective because it can significantly increase the total execution times of the rescheduled tasks. For example, the rescheduling of an almost completed task can add extra time to the overall execution time of the job and use more resources than expected. In addition, these redo solutions cannot guarantee the successful execution of these tasks. Furthermore, several tasks may experience failures because of the same reason, although these failures may be discovered by other Hadoop nodes. This is due to the fact that each computing node in Hadoop handles failures on its own for simplicity and scalability reasons.

Given the importance of task scheduling, several algorithms have been proposed to assign tasks to the available resources in the literature for Hadoop. Even though these proposed algorithms could improve the performance of Hadoop, several tasks failures still occur either when scheduling or executing the received task in Hadoop because of unforeseen events. More precisely, they can occur because of poor scheduling decisions (*e.g.*, resources deadlock, task starvation) or constraints related to the environment where they are executed (*e.g.*, data loss, network congestion).

The Hadoop scheduler should take into account several factors related to the environment where the tasks are processed (*e.g.*, available resources, number of tasks on the queue, the running load on each TaskTracker). However due to the increasing number of received demands over time and the dynamic availability of machines in a Hadoop cluster, it is challenging to take all these factors into account during the scheduling decision process. Consequently, the Hadoop scheduler may fail to meet its requirements and still generates poor scheduling decisions due to different constraints such as unpredicted demands of services or hardware outages. For instance, these failures can be because of different reasons such as resources-deadlock, task starvation, deadline non-satisfaction and data loss, *e.g.*, [15, 16]. For example, different tasks fail because of unexpected resources contentions by long-execution tasks, struggling tasks, etc., *e.g.*, [17, 18].

Moreover, the recovery mechanisms in Hadoop can affect the performance of the running applications and the overall Hadoop cluster utilization. For example, the recovery times of the failed nodes in Hadoop can be long and can lead to longer execution times of the jobs more than expected. This can result in extra delays to the total completion time and leads to resources wastage. Hence, it can significantly decrease the performance of the applications and increase failure rates. For instance,

the Hadoop JobTracker is not able to quickly detect failures of the TaskTrackers due to the fixed heartbeat-based failure detection mechanism adopted in Hadoop to track active nodes. As a result, it is not able to discover the failures of tasks running on the failed nodes and hence, it cannot reschedule them on time. Consequently, this can negatively impact the performance of the jobs running on the Hadoop framework and the overall resources utilization of the cluster.

On the other hand, Hadoop is widely used in several safety and critical applications, such as healthcare [19] and aeronautics [20]. Hence, it poses an open challenge for software engineers to design and test such framework to avoid failure occurrences. Therefore, thorough testing and verification of the Hadoop scheduler is of paramount importance to better analyze the circumstances leading to task failures and performance degradation. Knowing these circumstances upfront would allow Hadoop developers to anticipate these potential issues and propose solutions to overcome them.

Traditionally, simulation and analytical modeling have been widely used to verify and validate the behavior of Hadoop with respect to scheduling requirements, *e.g.*, [15, 16, 17, 18]. However, the size of Hadoop clusters have grown continuously to accommodate the increasing number of demands. As a result, Hadoop systems became very complex and hence their management became very difficult and expensive. Consequently, simulation and analytical modeling are inadequate because they are not efficient in exploring large Hadoop clusters. Moreover, given the complexity and the wide range of constraints in Hadoop scheduler, they are not able to provide accurate results and exhaustive coverage of the Hadoop system especially when failures occur. Overall, they are not able to ascertain a complete analysis of the Hadoop scheduler.

Considering the above facts, failures detection and recovery pose a critical and open challenge for Hadoop developers to design efficient recovery mechanisms to avoid

the occurrence of these failures. Hence, it is of interest to study the existing scheduling strategies of Hadoop in order to propose improvements that increase the success rate of scheduled tasks and jobs, reducing their execution times and resource usage. Therefore, we propose in this thesis to build an *adaptive failure-aware scheduling algorithm for Hadoop* to overcome the aforementioned limitations. Our proposed scheduling algorithm allows to predict task scheduling failures and achieve early rescheduling of the potential failed tasks. Also, it integrates new strategies to make adaptive decisions while scheduling the tasks according to the changes occurring in Hadoop environment. This is to reduce failures rate and improve tasks' execution times and resources utilization. In addition, the proposed scheduling algorithm will be able to early detect failures of TaskTrakers and can early reschedule the potential failed tasks on alive nodes and make better scheduling decisions. We implement the above approaches in *ATLAS*: an AdapTive Failure-Aware Scheduling algorithm that can be built on top of existing Hadoop schedulers. Finally, we propose a formal approach to analyze the impact of the scheduling decisions on the failures rates in Hadoop. This proves to be useful to early identify circumstances and specifications leading to potential failures and prevent their occurrences.

1.3 Proposed Methodology

The main objective of this thesis is to develop an adaptive and failure-aware scheduling algorithm for Hadoop. To this aim, we first propose a dynamic approach for Hadoop schedulers to early identify potential failures of tasks. Second, an adaptive approach is used to allow the proposed scheduling algorithm to generate better strategies to schedule tasks according to changes in the cloud. Moreover, our proposed algorithm integrates a dynamic approach to dynamically detect failures of TaskTrackers in

Hadoop. Finally, the cluster settings and scheduler' design can be upfront adjusted to prevent poor scheduling decisions in Hadoop using a formal approach to analyze the impact of scheduling decisions of Hadoop on the failures' rates. Overall, our proposed scheduling algorithm can help the Hadoop framework reduce tasks' failures and avoid making poor scheduling decisions. Figure 1.2 provides an overview of our proposed methodology.

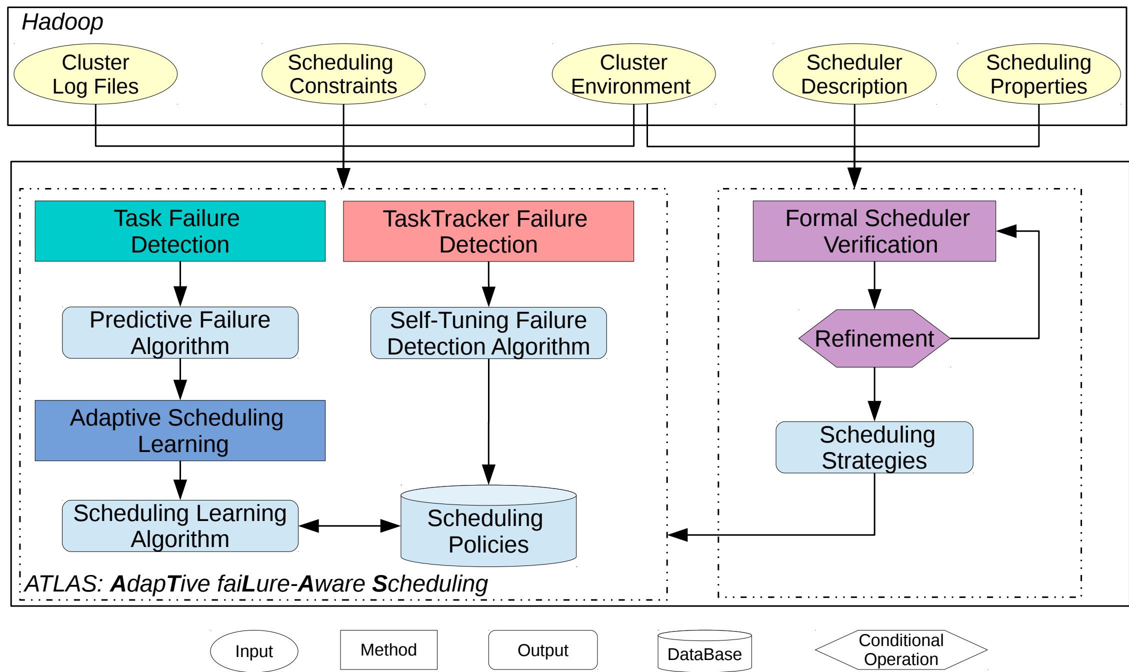

Figure 1.2: Overview of the Proposed Methodology

The proposed methodology addresses the limitations of current schedulers of Hadoop in terms of failures detections and adjusting its scheduling strategies according to the Hadoop environment. Overall, it is comprised of four main approaches:

(1) A dynamic approach to detect tasks failures based on collected data about previous tasks failures and machine learning algorithms [21]. We apply learning algorithms to implement a prediction algorithm allowing the scheduler to identify whether a task will finish or fail.

(2) An adaptive approach to model scheduling strategies using Markovian Decision Process (MDP) models [22] and reinforcement learning algorithms [23] (*e.g.*, Q-Learning [24] and State-Action-Reward-State-Action (SARSA) [25]).

(3) A dynamic approach to quickly detect the failures of the TaskTracker nodes using four well known algorithms from the network field, namely the Chen Failure Detector [26], Bertier Failure Detector [27], ϕ Failure Detector [28] and Self-tuning Failure Detector [29]. This is to produce an algorithm to dynamically adjust the timeout interval at which a TaskTracker node is considered as dead.

These three proposed approaches are complementary and can be connected together through a novel algorithm that can be integrated and built with the existing Hadoop schedulers (*e.g.*, FIFO, Fair, Capacity) in order to produce an *A*daptive fai*L*ure-*A*ware *S*cheduling (*ATLAS*) algorithm for Hadoop.

(4) A formal verification approach to identify circumstances leading to tasks' failures in the Hadoop schedulers using model checking [30]. We use the Communicating Sequential Processes (CSP) [31] language to formally model a Hadoop scheduler, and the Process Analysis Toolkit (PAT) [32] model checker to verify its properties. Next, we investigate the correlation between the scheduling decisions and the failures rate to propose and refine possible scheduling strategies for ATLAS.

1.4 Thesis Contributions

The ultimate goal of this thesis is the development of an adaptive and failure-aware scheduling for Hadoop to reduce its failures rates and improve its generated scheduling decisions. We list below the main contributions of this thesis with references to related publications provided in the Biography section at the end of the thesis.

- A Systematic Literature Review (SLR) [33] of task scheduling techniques in Hadoop, Spark [34], Storm [35], and Mesos [36]. [Bio-Jr2]

- A feasibility study on Google clusters to show the effectiveness of our proposed methodology to early identify tasks failures in the cloud. [Bio-Tr2, Bio-Cf3]

- A methodology to early identify the failures of tasks in Hadoop using information about the tasks and machine learning algorithms. [Bio-Tr1, Bio-Cf2]

- A novel method to generate adaptive scheduling decisions to reduce tasks' failures and avoid poor scheduling decisions in Hadoop using reinforcement learning techniques. [Bio-Jr1]

- A dynamic approach to detect the failures of TaskTracker nodes in Hadoop using data about previous heartbeat messages and four well known algorithms from the network field. [Bio-Jr1]

- Implementation of an *AdapTive and faiLure-Aware Scheduling (ATLAS)* algorithm for Hadoop to track failures and adjust the scheduling decisions on the fly. [Bio-Tr1, Bio-Cf2, Bio-Jr1]

- An approach to formally analyze the existing Hadoop schedulers using model checking and propose possible scheduling strategies for ATLAS [Bio-Cf1]

1.5 Thesis Organization

The rest of the thesis is organized as follows: In Chapter 2, we review the state of the art of failure-aware and adaptive scheduling approaches in Hadoop as well as formal verification techniques of Hadoop and cloud systems. In Chapter 3, we present (1) a

failure-aware methodology to early identify failures of tasks, (2) an adaptive approach to generate better scheduling decisions to reduce tasks' failures, and (3) an approach to adjust the communication between the JobTracker and TaskTrackers in order to quickly detect the failures of the TaskTracker nodes. The feasibility and efficiency of these three approaches will be demonstrated on a Hadoop cluster deployed on Amazon Elastic MapReduce (EMR). In Chapter 4, we describe the implementation of ATLAS, which can adjust the scheduling decisions on the fly in Hadoop. To assess the performance of ATLAS, we conduct an empirical study comparing its performance with those of the three existing Hadoop scheduling algorithms. In Chapter 5, we present our approach to formally analyze Hadoop schedulers using model checking. Thereafter, we apply our proposed approach on the scheduler of OpenCloud, a Hadoop-based cluster, and on ATLAS. Finally, Chapter 6 concludes this thesis, presents a description of some challenging aspects of our work, and outlines potential future research directions.

Chapter 2

Related Work

In [Bio-Jr2], we conducted a Systematic Literature Review (SLR) [33] of task scheduling techniques in Hadoop, Spark [34], Storm [35], and Mesos [36], in order to exhaustively identify and classify existing task scheduling techniques in these frameworks. We found that different built-in fault-tolerance mechanisms are integrated within the Hadoop framework. Therefore, in this chapter, we review the most relevant failure-aware and adaptive scheduling approaches in Hadoop. Furthermore, we present existing formal verification techniques of Hadoop and cloud systems.

2.1 Failure-aware Scheduling in Hadoop

Several studies, including [18, 37, 38], report that the Hadoop scheduler is unaware of the failures (*e.g.*, struggling tasks, node failure) encountered by the different components of Hadoop as well as the failures encountered by the scheduled tasks. Therefore, this prevents it from operating correctly and efficiently towards meeting its objectives. To alleviate this issue, Dinu and Ng [37] designed *RCMP* a failure resilience strategy for Hadoop scheduler. RCMP allows the job re-computation upon failures; by recomputing the necessary tasks rather than replicating the data. However, this strategy

was found to be efficient only for I/O intensive jobs and it is not valid for all types of MapReduce workload (*e.g.*, CPU intensive jobs).

Smart replication of intermediate data was also proposed as a solution to improve the performance of Hadoop under TaskTracker failures [39, 40]. The replication of these intermediate data allows a fast re-computation of the generated map output and a fast recovery when one replica is lost. However, it has been shown in [39] that this solution comes at the cost of some overhead when there is no failure, especially that the scheduler is unaware about these failures. Hence, it can significantly aggravate the severity of existing hotspots.

Quiane-Ruiz *et al.* [41] proposed a Recovery Algorithm for Fast-Tracking ($RAFT$) that tracks tasks at different checkpoints to store the execution status of tasks. In the event of task failure, the scheduler reschedules the failed tasks from the last available checkpoint. RAFT shows good results when a job encounters a failure; the scheduler does not need to re-execute the finished tasks belonging to the failed jobs. Also, it allows a fast recovery of the failed tasks since part of their output was stored at some point. Overall, it could reduce the total execution time of the scheduled tasks by 23%. However, this solution comes with an overhead to store these checkpoints, especially for large Hadoop jobs. In addition, the authors did not propose fault-tolerance mechanisms for these checkpoints.

Yuan *et al.* [42] designed a dynamic approach to early detect failures of scheduled tasks by collecting data about these tasks and storing their backups over regular interval times. Upon the detection of failures, the authors proposed that the scheduler would launch the failed tasks on other nodes while using the available information about their stored backups. This approach allows to not lose the intermediate data of the scheduled tasks.

In addition to the above work, Gupta *et al.* [43] designed *Astro* to early identify the most important metrics leading to the failure of scheduled tasks. They used different machine learning algorithms to predict these anomalies. Overall, Astro is able to early detect tasks' failures and notify the scheduler accordingly. It could improve resources usage by 64.23% compared to the existing implementations of Hadoop schedulers. The performance of Astro was improved by integrating mechanisms allowing a better distribution of workloads between the nodes of the cluster. Hence, the execution times of the scheduled tasks were reduced by 26.68% during the time of an anomaly.

A Maximum Cost Performance (*MCP*) algorithm was proposed by Qi *et al.* [44] as a solution to improve the existing speculative execution strategies in Hadoop. Although MCP achieves good performance, it was found to negatively impact the scheduling times of some jobs (batch jobs in particular) [45].

To address the aforementioned limitation, a Combination Re-Execution Scheduling Technology (*CREST*) algorithm [46] was proposed to improve the MCP algorithm. CREST considers data locality when launching the speculative execution of slow running tasks. Furthermore, CREST forces the re-execution of map tasks having local data instead of launching speculative tasks without considering data locality. However, results show that CREST adds an extra cost because of the replication of executed map tasks.

Chronos is a failure-aware scheduling strategy proposed by Yildiz *et al.* [47, 48] as a solution to enable early recovery actions for the failed tasks in Hadoop. Chronos enables a pre-emption technique to allocate the required resources to the recovered tasks. So, it is characterized by a fast action rather than waiting for an uncertain amount of time to recover the failed tasks. In addition, it considers one of the most

important constraint affecting the performance of Hadoop: the data locality while recovering the tasks. Although Chronos could reduce the total completion times of jobs by up to 55%, it is using strategies (*e.g.*, wait and kill) that could lead to resource wastage and degrade the performance of Hadoop clusters.

Despite of its direct impact on the failures' rates of Hadoop clusters, very few work have addressed the problem of early identification of Tasktrackers' failures in Hadoop. Among them, Zhu and Chen [38] implemented an algorithm to control the sending of heartbeats between the JobTracker and the TaskTrackers. To do so, they proposed to adjust the *expiry interval* for the JobTracker to detect the failure of a TaskTracker. The proposed algorithm uses information about jobs' sizes and the number of nodes to adjust the value of the expiry interval. In addition, the authors developed a reputation-based detector to decide whether a worker is failed or not according to a reputation threshold. For instance, they proposed to calculate the reputation of TaskTracker based on the number and time of received heartbeats. When a node is characterized by a reputation lower than a specified threshold, it is considered as dead. This approach can help detect failures of nodes early, and reduce the total execution time of jobs. However, it can generate wrong failures detections, which can negatively impact the performance of Hadoop. Moreover, it does not consider the dynamic nature of Hadoop environments and the different factors leading to the failures of TaskTrackers.

Overall, we observed that the current implementations of Hadoop scheduler lack mechanisms to share information about the failures encountered by both the different components and the scheduled tasks. Therefore, we propose in this thesis to integrate new fault-tolerant mechanisms within the Hadoop scheduler to avoid tasks and TaskTrackers failures.

2.2 Adaptive Scheduling in Hadoop

In the following, we present the most relevant work that describe existing adaptive mechanisms to improve the performance of Hadoop schedulers.

A Longest Approximate Time End (*LATE*) algorithm [49] was proposed as a solution to improve scheduling decisions in Hadoop by prioritizing tasks waiting in the queue according to collected information about running tasks. In addition, LATE considers the progress rate of the running tasks and the cluster's resources availability while scheduling tasks. Overall, LATE was able to reduce the total execution time by a factor of 2 in Hadoop clusters.

A Self-Adaptive MapReduce scheduling (SAMR) algorithm [50] was proposed to improve scheduling decisions of Hadoop by considering collected data about the hardware configurations of machines in a Hadoop cluster. In addition, SAMR is able to estimate the progress of the scheduled tasks by integrating different information about the hardware system. However, it does not consider other important factors about job characteristics (*e.g.*, the task size, data locality).

To overcome these limitations, an Enhanced Self-Adaptive MapReduce scheduling (ESAMR) algorithm [51] was designed to consider extra information about the Hadoop environment (including struggling tasks, job size, and remaining execution time). To do so, ESAMR uses the K-means clustering algorithm to calculate tasks execution times and identify slow running tasks. Overall, it could provide accurate results compared to SAMR and LATE. Furthermore, it allows to early identify the struggling map and reduce tasks and improve the execution times of jobs. However, it does not integrate possible solutions to reschedule these struggling tasks and does not improve the number of the finished tasks.

A Self-Adaptive Reduce Start time (SARS) algorithm [52] was integrated within

Hadoop scheduler to estimate start times of the reduce tasks. This is by using collected information about the completion times of maps and reduce tasks and the job total completion time. SARS uses these information to evaluate the impact of different starting times of reduce tasks on the total execution times of Hadoop jobs. As a result, the total response time of a Hadoop job was reduced on average by 11%.

A FaiR and Efficient slot configuration and Scheduling algorithm for Hadoop (*FRESH*) was proposed by Jiayin *et al.* [53] to identify the matching between submitted tasks and the available slots. FRESH could reduce the makespan between the tasks and provide a better resources distribution strategy across the scheduled tasks. Although each node in Hadoop has a specific number of slots, the scheduler continuously receives different running jobs requiring different slots configurations. the authors proposed to integrate a new management plan to dynamically find the best slot setting. Concretely, FRESH is able to dynamically change the assignment of slots between the map and reduce tasks according to the availability of slots and the requirement of the tasks. Overall, FRESH was able to have a better and fair distribution of the available slots across the scheduled tasks. However, it does not ensure a better memory usage.

FlexSlot [54] was proposed as a task slot management scheme for Hadoop schedulers. FlexSlot allows the identification of struggling map tasks and adjust their assigned slots accordingly. The authors opted for a dynamic strategy to change the number of slots on each node in Hadoop based on collected data about the resources utilization and struggling tasks. As a result, FlexSlot allows to better utilize the available resources in a Hadoop cluster and solves the problem of data skewness by adopting an adaptive speculative execution strategy. Consequently, it achieves good results in terms of total job completion times; reducing it by up to 47.2% compared to

the basic Hadoop scheduler. However, the proposed solution in FlexSlot generates an extra overhead that can impact the processing of Hadoop jobs. Also, it can negatively impact the number of failed tasks because it is using a task-killing-based approach in the slot memory resizing. Furthermore, it forces the killing of tasks multiple time. Hence, it may generate not only extra delays but also may cause the failure of the whole job.

WOrkflow over HAdoop (*WOHA*) was proposed by Li *et al.* [55] to improve workflow deadline satisfaction rates in Hadoop clusters. The authors proposed to use information about job ordering and progress requirements to select the worklflow to be processed first. WOHA selects the workflow that falls furthest from its progress based on the Longest Path First (LPF) and Highest Level First (HLF) algorithms. Hence, it could improve the satisfaction of workflow deadlines by 10% compared to existing scheduling solutions (FIFO, Fair and Capacity schedulers). WOHA collects data from the Hadoop environment to notify the scheduler about the potential deadline of each task (that are unknown ahead of time). However, WOHA does not integrate mechanisms to handle the dynamic nature of Hadoop workload over time.

Classification, and Optimization based Scheduler for Heterogeneous Hadoop (*COSHH*) was proposed by Rasooli *et al.* [56] as a hybrid solution to select the type of scheduling to use in the cluster based on the number of the incoming jobs and the available resources. The authors used Linear Programming (LP) to classify the incoming workloads and determine an efficient resources allocation strategy for Hadoop. According to the obtained results, the authors reported that the FIFO algorithm can be used for under-loaded systems while the Fair Sharing algorithm works well when the system is balanced. However, COSHH can be used when the system is overloaded (*i.e.*, peak hours). In summary, this proposed hybrid solution allows

to improve the performance of Hadoop by reducing the average completion times and improving fairness, locality and scheduling times. Although Rasooli *et al.* [56] specified three different cases when to use each of the schedulers, they do not specify information thresholds upon which one can decide about which scheduler to use.

In summary, we found out that the Hadoop scheduler lacks mechanisms to handle both failures and dynamic changes in cloud environments. Therefore, we propose a new adaptive approach in this thesis to reduce tasks' failures and avoid making poor scheduling decisions in the Hadoop framework.

2.3 Formal Verification of Hadoop and Cloud

The use of formal verification [57] to model and verify Hadoop and cloud systems is recent. In the following, we describe the most relevant studies that have been proposed to show the efficiency of formal verification techniques in the context of Hadoop and cloud systems.

2.3.1 Formal Analysis of Hadoop System

The authors in [58, 59] evaluated the performance of MapReduce using Stochastic Petri Nets and Timed Coloured Petri Nets, respectively. More precisely, they simulated Hadoop jobs using Petri Nets and defined formulas of mean delay time in each time transition. Although the two proposed approaches are able to evaluate the performance of MapReduce, they lack details about scheduling constraints affecting the performance of the executed jobs and tasks.

Su *el al.* [60] used the CSP language to formally model the master, mapper, reducer and file system in MapReduce. They formally modeled these components

while considering basic operations in Hadoop including task state storing, error handling, progress tracking. But, the authors did not verify any properties of the Hadoop framework using the formalized components.

Based on the formalized components proposed in [60], Xie *el al.* [61] formally verified the HDFS reading and writing operations using CSP and the PAT model checker. Some of the properties in HDFS are verified including the deadlock-freeness, minimal distance scheme, mutual exclusion, write-once scheme and robustness. The verification results showed that their proposed approach is able to detect unexpected traces generating errors and verify data consistency in the HDFS. One limitation of this work is that it models the operations to read and write only one file in HDFS, which is not the case in Hadoop where multiple files exist.

Reddy *et al.* [62] used CSP to model the “NameNode”, “DataNode”, task scheduler and cluster setup in Hadoop. They used the PAT model checker to verify some properties including data locality, deadlock-freeness and non-termination in a Hadoop system. In addition, they showed the benefits of these properties among other ones only when using a small workload. Also, the authors did not check their impact on the scheduling strategies of Hadoop.

Although theorem provers are widely used to check the correctness and reliability of several distributed systems, to the best of our knowledge, we only found one theorem prover-based study that verifies the actual running code of MapReduce applications. Ono *et al.* [63] proposed an abstract model to verify the correctness of the application running on Hadoop-MapReduce using the proof assistant Coq [64]. For instance, they modeled the mapper and reducer functions in MapReduce and proved that they satisfy the specification of some applications such as WordCount [65]. However, the presented abstract model lacks several details related to task assignment and resources

allocation, which largely affect the performance of applications running on Hadoop.

2.3.2 Formal Analysis of Cloud System

Jarraya *et al.* [66] introduced a cloud calculus, which is a process algebra based on structural congruence and a reduction relation. They used cloud calculus to model and verify the specification of the migration of virtual machines and security policies in the cloud.

Bansal *et al.* [67] used the *ProVerif* model checker to analyze the security of cloud based Encrypted Storage Protocols (ESP) against attacks. However, ProVerif is a formal tool proposed only for reasoning about security properties in cryptographic protocols.

Armando *et al.* [68] used the *AVISPA* model checker to analyze security issues in the cloud. They expressed these issues based on the Intermediate Format (IF) and a set rewriting as formal foundation. The AVISPA model checker is also limited to the analysis of Internet security protocol and applications.

Another work [13] was proposed to formally verify some vulnerability properties in a cloud computing system using the *NuSMV* model checker [69]. In this work, the authors constructed formal state models for cloud systems and the properties they aim to verify. Then, they translated the obtained models into NuSMV to verify a set of properties within a three-tier cloud system that uses Amazon EC2. Although the NuSMV model checker was used to verify some cloud systems, it does not support the verification of probabilistic properties.

Probabilistic model checking has also been used to analyze models of cloud systems. Kikuchi *et al.* [70] used the PRISM probabilistic model checker to verify some migration properties in concurrent virtual machines (VM) in cloud systems. They

conducted experiments to measure the performance of migrations in the VMs. Then, they used the results of their experiments to construct formal models for migration properties such as having more than 4 migration operations retained in a certain sender server. They used PRISM to check whether the performance models satisfy the migration properties.

Another work was proposed by Naskos *et al.* [71] to model and verify elasticity in cloud computing systems. They used MDP to formally model elasticity properties and PRISM to model and verify several elasticity decision policies including resizing a cluster and dynamically modifying the number and types of VMs. However, the authors of [71] did not report any results about the size of the explored state space in the verified system and the scalability of their approach.

Ishakian *et al.* [72] used the Coq theorem prover to express Service Level Agreements (SLAs) and set theory to analyze the efficiency of co-location in cloud systems. They provide machine verified proofs and showed that the proposed framework is consistent with respect to its semantic. However, Coq is not an automated verification tool and requires interaction with the user. In addition, it does not allow for a fast detection of errors when the verification fails.

Thuraisingham *et al.* [73] proposed to verify information sharing systems in cloud computing environments in terms of soundness, transparency, consistency and completeness. They used the ACL2 theorem prover [74] to verify these properties. However they did not present details about their modeling and verification.

2.4 Summary

In this chapter, we presented the most relevant work that addressed failure-aware and adaptive scheduling in Hadoop and the formal analysis of Hadoop and cloud systems.

Precisely, we described the existing techniques to detect the failures of tasks and TaskTrackers and recover from these failures in Hadoop. In summary, we found out that the Hadoop scheduler lacks mechanisms to share information about the failures encountered by both the different components and the scheduled tasks. Therefore, it is necessary to design efficient recovery mechanisms to avoid the occurrence of these failures in Hadoop. On the other hand, we described the techniques used to analyze Hadoop and cloud systems. Overall, we found that the use of formal methods in the cloud and Hadoop is very recent and only a few work exist in the open literature. Most importantly, there is no work that formally analyze the impact of scheduling decisions on the failures rate of Hadoop.

Chapter 3

Failures Detection and Adaptive Scheduling

In this chapter, we present novel approaches to track failures and generate adaptive scheduling decisions for Hadoop. First, we present our methodology for task failure detection based on machine learning algorithms and data about previously executed tasks in Hadoop to early identify the failures of tasks. Second, we propose a new adaptive approach for modeling scheduling policies using reinforcement learning algorithms (Q-Learning and SARSA) to select an appropriate scheduling action for a scheduled task in Hadoop framework. Instead of the fixed heartbeat-based failure detection, we propose to dynamically adjust the communication between the JobTracker and TaskTrackers in order to quickly detect the failures of the TaskTracker nodes using four well known algorithms from the network field. We demonstrate the feasibility and efficiency of our proposed approaches on a 100-nodes Hadoop cluster deployed on Amazon Elastic MapReduce (EMR).

3.1 Limitation of Current Schedulers

In this section, we first describe the existing mechanisms in Hadoop schedulers to detect failures of tasks and to recover from them. Next, we describe the limitations of the current Hadoop schedulers when detecting active TaskTrackers nodes in a Hadoop cluster.

3.1.1 Tasks Failures Detection

When a Hadoop component fails (*e.g.*, TaskTracker, DataNode), the tasks running on this node will fail and have to be restarted on other nodes. Moreover, the recovery times of the failed nodes can be long and can lead to unpredictable execution times and resources wastage. For instance, Dinu *et al.* [18] report that the average execution time of a Hadoop job, which is 220 seconds, can reach 1000 seconds under a TaskTracker failure and 700 seconds under a DataNode failure. Furthermore, they state that the failure of a DataNode can delay the starting time of speculative execution of some tasks. This is due to the statistical nature of the speculative execution algorithm used in Hadoop to collect data about task progress (*e.g.*, struggling tasks). For example, when a DataNode fails and a task is making good execution progress, the scheduler expects the same progress from that task and hence it will start its speculative execution with a delay. Consequently, the speculative execution of that task will start later than the time when struggling tasks are usually speculatively executed.

Dinu *et al.* [18] claim that Hadoop components do not share failure information appropriately. For instance, they report that Hadoop components do not share information about task failure with other tasks that depend on the failed task on time. For simplicity and scalability, each computing node in Hadoop manages failure detection and recovery on its own and hence failure information are not shared between these

nodes. Therefore, multiple tasks, including the speculative tasks, may experience the same failure although a previous task encounters the same failure. For example, when a map task fails, the failure is likely to translate into the failure of the whole job since map and reduce tasks are scheduled separately and there is no exchange of failure information between them. This is due to the tight dependency between the map and reduce tasks.

In the sequel, we present an example of a Hadoop job failure to better understand its impact on the performance of the Hadoop framework. Let's consider N jobs submitted to a Hadoop cluster. Each job is composed of X map tasks and Y reduce tasks, where each job is using *R(CPU, Memory, HDFS Read/Write)* resources from M machines in a Hadoop cluster. Each new submitted task is assigned to a node to be executed, if it fails, it has to be rescheduled on the same node or on another available node. When a task exceeds its maximum number of scheduling attempts, it is considered to be failed, otherwise it is successfully finished. The failure of one task can cause the failure of the whole job to which the task belongs because of the the dependency between map and reduce tasks. Figure 3.1 shows an example of a Hadoop job failure, *Job3*, because one of its map tasks failed (since it exceeded its maximum number of scheduling attempts). Consequently, all reduce tasks were failed automatically.

To formally describe the final scheduling outcome of an executed job, we consider $S(job)$ the outcome of a job, $S(MapAtt_{ip})$ the outcome of a map_i after the pth attempt and $S(ReduceAtt_{jq})$ the outcome of $reduce_j$ after the qth attempt. We attribute a value of 1 when an attempt is successful and 0 otherwise. We assume that K and L are the maximum numbers of scheduling attempts allowed for map and reduce tasks,

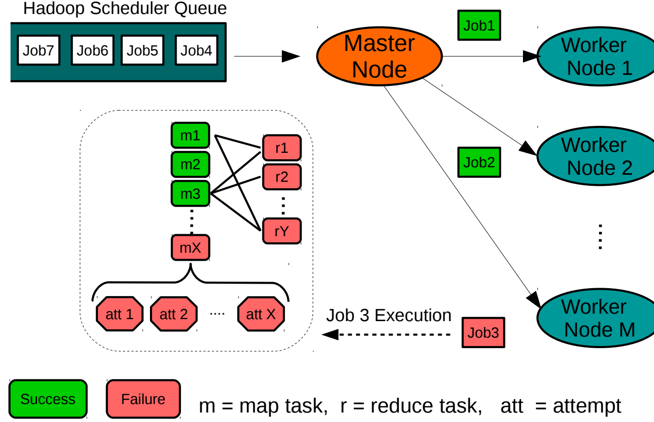

Figure 3.1: Example of Hadoop Job Failure

respectively. The scheduling outcome of a job can be described using Equation 3.1:

$$S(job) = [\prod_{i=1}^{X}(\sum_{p=1}^{K} S(MapAtt_{ip}))] * [\prod_{j=1}^{Y}(\sum_{q=1}^{L} S(ReduceAtt_{jq}))] \tag{3.1}$$

The total execution time of a task is the sum of execution times of all its launched attempts (both the finished and the failed attempts). Therefore, an executed task may have a long execution time when it is characterized by multiple attempts (especially the failed ones). Long execution times of tasks will be translated into a longer execution time of the job to which they belong. For instance, let *T(job)* be the total execution time of a job comprised of $A = \{map_i\}_{i \in X}$ map tasks and $B = \{reduce_j\}_{j \in Y}$ reduce tasks. Let $T(MapAtt_{ip})$ be the execution time of the p attempt when executing map_i and $T(ReduceAtt_{jq})$ be the execution time of the q attempt when executing $reduce_j$. The total execution time of a job be described using Equation 3.2:

$$T(job) = Max_A(\sum_{p=1}^{K} T(MapAtt_{ip})) + Max_B(\sum_{q=1}^{L} T(ReduceAtt_{jq})) \tag{3.2}$$

The job having many failed attempts, even though they are finished at the end, are characterized by the longest execution time compared to jobs having less failure

attempts. Consequently, if the scheduler can reduce the number of failure attempts it allows jobs to reduce their execution times as well. Therefore, by reducing the number of failed task attempts, one will reduce the number of failed tasks and the turnaround time of jobs running in the cluster. In addition, we believe that if one can early identify the circumstances that may lead to a map/reduce task failure, one may be able to reduce the failures rate in a Hadoop framework.

3.1.2 TaskTrackers Failures Detection

In [18], it is shown that the JobTracker is not able to quickly detect the failures of the TaskTracker nodes because of the fixed heartbeat-based failure detection mechanism commonly used in Hadoop to track active TaskTrackers. Consequently, it cannot quickly detect tasks running on failed TaskTrackers and it may assign tasks to dead nodes. Hence, this can significantly increase the number of failed tasks in Hadoop. For instance, active TaskTrackers send heartbeat messages to JobTracker every 3 seconds. While the JobTracker checks every 200 (3.33 minutes) seconds the timeout condition of the received heartbeat messages from the TaskTrackers. The JobTracker declares a TaskTracker as dead only when this TaskTracker node does not send a heartbeat for at least 600 seconds (10 minutes). In addition, it has to reschedule the tasks running on this node on other nodes, according to their availability [18]. On the other hand, because of network delays or messages losses, some heartbeats (may) arrive late to the JobTracker, which can affect the decisions of the JobTracker. For example, the JobTracker can consider their corresponding TaskTrackers as dead nodes, despite their availability. As a result, it will not assign them any load until they are added to the active nodes, resulting in resource wastage.

To better illustrate the TaskTrackers' failures detection approaches in Hadoop,

we present examples of communications between JobTracker and TaskTracker nodes in Figure 3.2. For instance, when JobTracker receives a heartbeat message m_1, sent from a TaskTracker, before the next arrival time, it considers this TaskTracker to be alive. The TaskTracker can send a new heartbeat message m_2 that does not arrive to the JobTracker, because of a network problem or message loss. Then, the JobTracker considers this TaskTracker as dead despite its availability. This can result in resources wastage. When a TaskTracker sends a heartbeat message m_n arriving on time, if

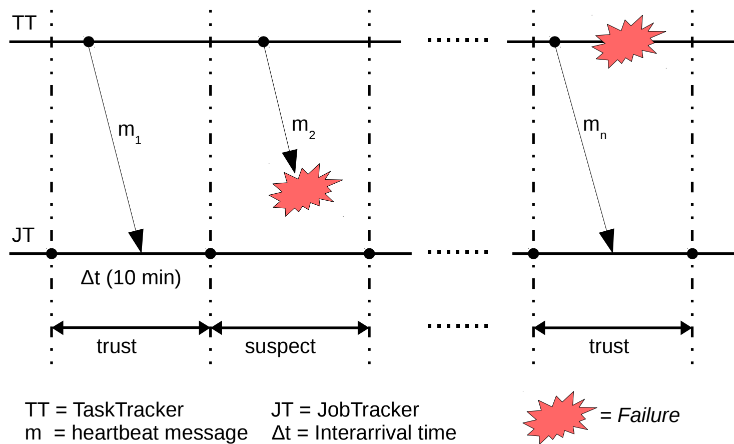

Figure 3.2: TaskTracker Failure Detection Model in Hadoop Framework

this TaskTracker experiences a failure right after sending the message, the JobTracker will consider this node as alive and will assign it new tasks until the next timeout to check the heartbeats messages. As a consequence, this could increase the failure rates of tasks and the execution times of tasks and jobs [18]. Therefore, we believe that it is very important to integrate an adaptive approach adjusting the timeout used to detect the failures of TaskTrackers. In addition, if one can early detect the failures of the TaskTracker nodes, one can reduce the failures rate of tasks in the Hadoop framework.

In the sequel, we present our proposed methodologies to solve the aforementioned limitations of current Hadoop schedulers to early identify tasks failures and to

reschedule them accordingly. Furthermore, we present our approach to dynamically track failures of TaskTrackers.

3.2 Task Failure Detection

In this section, we describe our task failure detection methodology alongside with our evaluation approach and the obtained results.

3.2.1 Task Failure Detection Methodology

To overcome the limitations of Hadoop while tracking tasks' failures, a possible solution could be to equip the Hadoop scheduler with mechanisms that enable the early identification of failed tasks. In the following, we present an overview of our proposed methodology to predict the failure of a task in Hadoop followed by a description of each step.

General Overview

Figure 3.3 presents an overview of our proposed methodology to predict task scheduling outcomes and to adjust scheduling decisions to prevent failure occurrences. First, we collect traces of data about previously executed tasks and jobs in a Hadoop cluster. We parse the obtained log files from the Hadoop cluster to extract the main attributes of jobs and tasks. This step is necessary to analyze the correlations between tasks attributes and tasks scheduling outcomes and profile tasks failures. Finally, we apply statistical predictive learning techniques to build a tasks' failures prediction algorithm. The remainder of this section elaborates more on each of these steps.

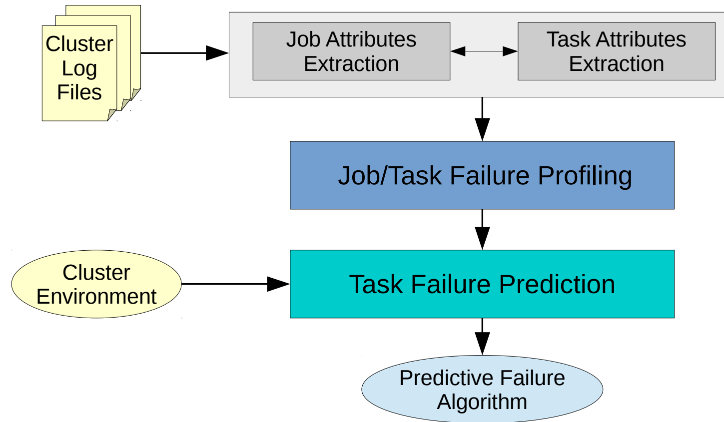

Figure 3.3: Task Failure Detection Methodology

Tasks/Jobs Attributes Extraction

Before starting the analysis of tasks' failures, we collect data about previously executed Hadoop tasks and jobs in a Hadoop cluster. To do so, we first run different workloads including single and chained jobs on Amazon EMR Hadoop clusters. A chained Hadoop job is composed of two or more Hadoop jobs. To get diverse workloads, we run different types of (single) Hadoop jobs including *WordCount*, *TeraGen*, *TeraSort* [75]. Moreover, we run different chained jobs (sequential, parallel and mix chains) composed of *WordCount*, *TeraGen*, *TeraSort* jobs to get a workload similar to the one in a real world cluster. In addition, we vary the size of the running Hadoop jobs by changing the number of map/reduce tasks in a job and the number of jobs in a chained job.

Using the collected log files, we perform the first step to extract attributes of the executed jobs and tasks. This is in order to identify and investigate possible correlations between the extracted attributes and the scheduling outcome of a task. To do so, we first propose to classify task and job attributes into four main categories as follows: *Task/Job attributes* = {*Identification, Structure, Execution, Environment*}. Indeed, the proposed classification of attributes can give a description about the internal structure of the tasks or jobs as well as the way they are executed on their

environment. For instance, the *identification* attributes include ID, priority, and type of a task. The *structural* attributes represent the dependent running/finished/failed tasks belonging to the same job. The *execution* attributes can include the execution time, resources utilization (CPU, memory, bandwidth), execution type (local or non-local), and scheduling outcome (either finished or failed) of the task. The *environment* attributes describe the status of the node where to execute the tasks including the running load (number of running map and reduce tasks), the status of the queue, etc. Given the classification presented above, we collect the attributes of tasks and jobs as described in Table 3.1.

Task/Job Failure Profiling

Given the collected data from the previous step, we analyze the dependencies between job/task scheduling outcome and their attributes to identify a possible correlation. We also carefully investigate the data of the failed tasks and jobs by mapping their outcomes to their attributes. Next, we identify the relevant attributes that impact the final scheduling outcome by applying the Spearman rank correlation [76]. Here, we choose to use the Spearman correlation because it allows to rank and to provide a measure of relationship between random variables; which is the case for our proposed model to predict the failure of a given task.

Table 3.1: Jobs and Tasks Attributes

Attribute	Description	Rationale
Job/Task ID	Immutable and unique identifier for a job/task	Used to identify a job/task
Type	Type of a task	It represents the type of a task: Map or Reduce
Priority	Preemption type of a task/job	Used to capture task/job priority to access resources
Locality/Execution Type	Locality/Execution type of a task	Used to capture the fact that a task was launched locally/speculatively or not
Execution Time	Time between submission date and date when task is finished/-failed	Used to capture the execution time of a task
Number Finished and Failed Tasks	Number of finished and failed tasks	Used to capture the proportion of finished/failed tasks
Number Previous Finished and Failed Attempts	Number of previous finished and failed attempts	Used to capture failure events dependent on a task
Number of Reschedule Events	Number of reschedule events of a failed task	Used to capture the number of times that a failed task was rescheduled
Number of Finished, Failed and Running Task TaskTracker	Number of finished, failed and running tasks on TaskTracker	Used to capture failure events on the same TaskTracker
Available Resources on TaskTracker	Amount of available resources on TaskTracker	Used to capture the availability of resources on TaskTracker
Total Number of Tasks of a Job	Total number of tasks within a job	Used to capture the distribution of tasks within the jobs
Used CPU/RAM/ HDFS R/W	Used CPU, RAM and disk space for a task	Used to capture the usage of resources
Final Status	Final state on a scheduling lifecycle	Used to describe the processing outcome of a task/job

Task Failure Prediction

The next step in our methodology consists in using the collected data from previous steps and machine learning techniques to investigate the possibility to predict a potential task failure, early on its occurrence. To this aim, we use several regression

and classification algorithms in R [77], a programming language used for statistical computing, to build models including the General Linear Model (GLM), Boost, Neural Network, Tree, Conditional Tree (CTree) and Random Forest [78]. We select these algorithms because they showed good performance to predict anomalies in different systems [79]. GLM is an extension of linear multiple regression for a single dependent variable. It is widely used in regression analysis. Boost is a succession of iterative models trained on a data set. In Boost, points misclassified by the previous model are given more weight and the successive models are classified (weighted) according to their success. The outputs of these models are combined using voting or averaging to create a final model. Neural networks denote a set of interconnected layers of nodes where the predictors are the input of the bottom layer and the forecasts are the output of the top layer. The Decision Tree is a widely used classification model to predict binary outcomes. CTree is an extension of the Decision Tree. Random Forest is a set of interconnected Decision Trees that uses the majority voting to provide classification (predicting, often binary, class label) or regression (predicting numerical values) results [77].

The inputs for each model are the task attributes and the output is the scheduling outcome of a task (either failed or finished). To develop the selected models, we use the implementation provided in the statistical framework R and compare their performance. The model having the best results will be used to implement the task failure predictive algorithm. To train these models, we use different training and testing data set collected over a fixed period of time of 10 minutes. Here, we should mention that the training time is related to the steps of training process and not to the complexity of the running jobs. Next, we apply 10-fold random cross validation where each data set is randomly divided into ten folds. Nine folds are used as the

training set, and the remaining fold is used as the testing set. This is to measure the accuracy, the precision, the recall and the error of the prediction models [77]. The accuracy is $\frac{TP+TN}{TP+TN+FP+FN}$, the precision is $\frac{TP}{TP+FP}$, the recall is $\frac{TP}{TP+FN}$, and the error is $\frac{FP+FN}{TP+TN+FP+FN}$, where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives.

Following the previously described steps, the machine learning models can predict the scheduling outcome of tasks (*i.e.*, successful completion or failure) based on information about the tasks attributes and the Hadoop cluster environment (*i.e.*, availability of resources, failure occurrences in TaskTracker, network congestion). The output of the proposed methodology, presented in Figure 3.3, is a predictive failure algorithm. Overall, the task failure detection approach allows to early identify potential failures that could affect the overall performance of the Hadoop scheduler. However, it is highly dependent on the quality and the relevance of the training data over time. Therefore, we decided to update the used data to train the models over regular intervals from the created cluster in order to get new data and update the used models accordingly. Algorithm 3.1 describes the steps followed to predict successes or failures of scheduled tasks. The first step consists of collecting data about previously executed tasks and then analyzing the correlation between tasks attributes and scheduling outcomes (lines 2 to 4). Next, we train the selected machine algorithms on the collected logs and measure their performance in terms of accuracy, precision, recall and time when applying a 10-fold cross validation step (lines 5 to 9). Thereafter, the model giving the best performance results is selected to be integrated within Hadoop (line 10). When there is a new task to be scheduled, data about this task will be collected to serve as input to the predictive algorithm that will predict whether the

task will finish or fail (lines 13 to 16). The collected logs are updated for the proposed predictive algorithm each 10 minutes (lines 1 and 11).

Alg. 3.1. Predictive Failure Algorithm

```
1: for (Each 10 minutes ) do
2:     logs = Collect-logs(Cluster)
3:     /* Analyse correlations between task attributes and scheduling outcome */
4:     Analyse-Correlation(logs)
5:     /* Apply Machine Learning predictive models on collected data */
6:     Machine-Learning(logs, models)
7:     10-fold-Cross-Validation(logs, models)
8:     /* Measure accuracy, precision, recall, error and time of predictive models */
9:     Performance = Measure-Performance(logs, models)
10:    Model = Select-Model(models, Performance)
11:    Update-logs(Cluster, logs)
12:    /** Integrate the predictive model within the scheduler **/
13:    while (There is a new task to be scheduled) do
14:        Attributes = Collect-Attributes(Task, TaskTracker)
15:        /* Selected predictive model will predict if task will be finished/failed */
16:        Predicted-Status = Predict(Model, Task, Attributes)
17:    end while
18: end for
```

3.2.2 Task Failure Detection Evaluation

In this section, we present our approach to evaluate the proposed methodology to detect task failure in Hadoop along with the obtained results.

Experimental Design

We collect logs from the cluster and extract data related to 120,000 jobs and 300,000 tasks. We use the collected task attributes as inputs to the proposed failure prediction algorithms. The output of these algorithms is a binary variable taking the value "True" if a scheduled task succeeds and "False" if it fails. The collected data are used to train and test the predictive models and we evaluate the performance of the

selected machine learning models. This step is performed for the map and reduce tasks separately for the three studied schedulers (FIFO, Fair and Capacity). We evaluate the performance of the selected models by applying a 10-fold random cross validation to select the algorithm able to early identify the failure of a task with the best accuracy, precision and execution time. We use different training rates of 10%, 30%, 50%, 70%, and 90% for the selected algorithms, described in Section 3.2.1. This is to evaluate the performance of the models at different training rates and analyze the impact of the training rate on their performance.

Experimental Results

First, we analyze the correlation between task attributes and the scheduling outcomes of the collected map and reduce tasks. We find that there is a strong correlation between the the number of running/finished/failed tasks on a TaskTracker, the locality of the tasks, the number of previous finished/failed attempts of a task, and the scheduling outcome of a task. Also, we notice that tasks that experience multiple failure events have a high probability to fail in the future. In other terms, the failed tasks are characterized by multiple past failed previous attempts and many concurrent tasks experiencing different failures.

We present the obtained performance results of the studied predictive algorithms when applied to the FIFO, Fair, and Capacity schedulers in Tables 3.2, 3.3, and 3.4, respectively. Overall, we find that Random Forest achieves the best results in terms of precision, recall, accuracy, and execution time for the three studied schedulers, when compared to the other predictive models. These results can be explained by the fact that Random Forest uses the majority voting on decision trees to generate results and hence it is robust to noise and can provide highly accurate predictions [77]. For map

tasks, the Random Forest model outperforms the other algorithms with an accuracy up to 88.5%, a precision up to 87.6%, a recall up to 93.4%, and an execution time of 29.33 ms. For reduce tasks, it achieves an accuracy up to 94.5%, a precision up to 97.4%, a recall up to 96.5% and an execution time up to 38.41 ms. In general, we observed that the results for the studied schedulers follow the same trend.

Table 3.2: Accuracy, Precision, Recall (%) and Time (ms): FIFO Scheduler

Task	Scheduler	FIFO			
Map Task	**Algorithm**	**Accuracy**	**Precision**	**Recall**	**Time**
	Tree	62.8	79.5	70.4	14.43
	Boost	73.5	80.6	71.3	190.4
	GLM	63.4	86.3	74.3	10.94
	CTree	64.3	84.6	68.5	16.14
	R.F.	**85.9**	**87.6**	**93.4**	**25.14**
	N.N.	64.7	86.4	74.3	63.51
Reduce Task	**Algorithm**	**Accuracy**	**Precision**	**Recall**	**Time**
	Tree	74.5	84.1	67.4	15.13
	Boost	75.2	84.5	73.5	314.15
	GLM	63.4	89.2	63.5	19.73
	CTree	80.1	91.2	79.8	19.85
	R.F.	**94.5**	**97.4**	**93.4**	**38.41**
	N.N.	71.4	81.9	74.4	85.14

R.F. = Random Forest, N.N. = Neural Network

While using different training rates, we analyze the performance of the selected predictive algorithms. Overall, we observe that the performance of the FIFO, Fair, and Capacity schedulers are following the same trend in terms of accuracy, precision, and recall. Hence, we only discuss the results of one scheduler: Fair scheduler. Random Forest outperforms the other predictive algorithms for the map and reduce tasks. In addition, we can report that the accuracy, precision, and recall values increase when the training rate increases, and can reach 83.9%, 94.3%, and 94.3%, respectively, under 90% training rate for the map tasks. The obtained results for the reduce tasks show that Random Forest is characterized by the highest accuracy, precision, and

Table 3.3: Accuracy, Precision, Recall (%) and Time (ms): Fair Scheduler

Task	**Scheduler**	**Fair**			
Map Task	**Algorithm**	**Accuracy**	**Precision**	**Recall**	**Time**
	Tree	68.6	75.8	63.4	10.02
	Boost	67.3	84.2	69.7	201.4
	GLM	65.6	89.5	65.4	13.54
	CTree	69.4	84.4	68.3	17.34
	R.F.	**79.9**	**81.8**	**93.5**	**23.9**
	N.N.	64.8	86.3	74.1	63.61
Reduce Task	**Algorithm**	**Accuracy**	**Precision**	**Recall**	**Time**
	Tree	74.5	85.4	74.0	15.23
	Boost	84.4	81.7	74.7	268.77
	GLM	77.2	94.3	71.3	19.19
	CTree	82.4	88.4	79.4	20.52
	R.F.	**94.12**	**92.3**	**96.5**	**29.77**
	N.N.	84.3	85.4	75.6	98.14

R.F. = Random Forest, N.N. = Neural Network

recall values, 93.4%, 97.8%, and 93.9%, respectively, with a training rate of 90% .

Consequently, we can conclude that the Random Forest algorithm depends on the training rate, and the higher is this training rate, the better would be its performance. In light of these results, we select Random Forest for the implementation of our proposed task failure prediction methodology, presented in Section 3.2.1, and retrain its model to collect data over regular interval of times (*e.g.*, 10 minutes).

Summary: To address the limitations of Hadoop scheduler to detect failures of tasks, we presented a methodology for task failure detection using machine learning algorithms. Overall, we found that the Random Forest algorithm outperforms other algorithms in terms of accuracy, precision, and recall. Next, we proposed a predictive algorithm to early identify the failures of tasks and adjust the decisions of the Hadoop scheduler based on collected data about the scheduled tasks and machine learning algorithm (Random Forest). However, given the dynamic nature of cloud

Table 3.4: Accuracy, Precision, Recall (%) and Time (ms): Capacity Scheduler

Task	Scheduler	Capacity			
Map Task	**Algorithm**	**Accuracy**	**Precision**	**Recall**	**Time**
	Tree	64.5	83.3	71.2	62.14
	Boost	75.4	83.7	79.5	295.44
	GLM	65.2	84.7	65.8	19.14
	CTree	64.9	80.9	64.2	20.51
	R.F.	**88.5**	**83.4**	**89.9**	**29.33**
	N.N.	72.1	85.4	79.3	63.84
Reduce Task	**Algorithm**	**Accuracy**	**Precision**	**Recall**	**Time**
	Tree	61.4	74.3	65.4	14.15
	Boost	64.4	85.4	74.7	208.98
	GLM	67.8	83.7	75.3	25.44
	CTree	60.5	81.4	60.4	27.18
	R.F.	**82.3**	**92.5**	**89.4**	**30.15**
	N.N.	77.4	90.6	81.4	90.4

R.F. = Random Forest, N.N. = Neural Network

environments, the Hadoop scheduler can generate poor scheduling decisions that can affect the failures rates and performance of jobs running on Hadoop. Therefore, we present in the next section a methodology that builds on these findings to enable the early identification of failed tasks and a quick rescheduling of these tasks on available nodes.

3.3 Adaptive Scheduling

Building on the results of previous section on task failure detection, we now present an adaptive approach to select scheduling policies and avoid poor scheduling decisions in Hadoop scheduler.

3.3.1 Adaptive Scheduling Methodology

In the sequel, we present an overview of our proposed methodology to generate adaptive scheduling decisions in Hadoop followed by a description of each step.

General Overview

In the previous section, we presented a methodology that enables the early identification of tasks' failures based on collected data about Hadoop environment and machine learning algorithm (*i.e.*, Random Forest). More precisely, our methodology can predict whether a task will eventually finish or it will fail. Tasks predicted to be finished will be executed whereas, tasks predicted to fail will be rescheduled according to the events occurring in the scheduler's environment. However, given the dynamic nature of Hadoop environment and the wide range of constraints and aspects involved in Hadoop scheduler, it is important to propose efficient scheduling strategies to handle the unpredictable changes in Hadoop scheduler environments and avoid poor scheduling decisions.

Figure 3.4 depicts a new methodology to model and train adaptive scheduling algorithms for Hadoop. To this aim, we consider the scheduling decisions in Hadoop as an MDP model where the scheduling actions are the transitions between the different states in this model and each transition from one state to another one is associated with a reward. Second, we propose to use the reinforcement learning algorithms to compare the reward associated with all possible actions, to select a possible scheduling strategy for the submitted task. The output of this methodology is a scheduling learning algorithm and a set of scheduling policies for the Hadoop scheduler. The remainder of this section elaborates more on each of these steps.

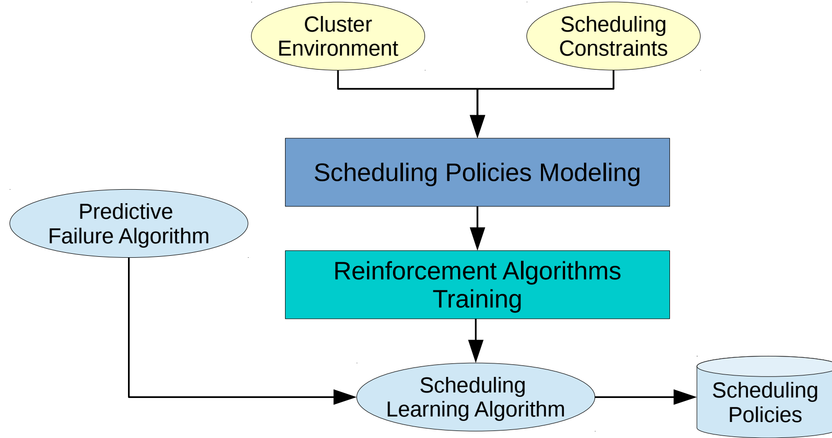

Figure 3.4: Adaptive Scheduling Methodology

Scheduling Policies Modeling

The Hadoop scheduler requires adaptive scheduling strategies to handle the unpredictable events occurring in nodes where tasks are executed and reduce the cost associated with tasks execution. To this aim, we consider the scheduling decision process in the scheduler as an MDP [80]. Indeed, MDPs are widely used to model different decisions procedures where the behavior of the systems depend on random factors and are under the control of a decision maker, which is the case for a Hadoop cluster. In addition, MDPs showed good performance when applied to solve decision problems in cloud environment, *e.g.*, resources allocation [81] or virtual machines scheduling problems [22]. Given the past successes with MDP models in cloud systems, we believe that MDP model can help selecting the possible actions from the current state and observing the derived reward/cost from each transition in order to find a better scheduling decision in Hadoop.

More concretely, we consider the scheduling and execution of tasks as a life cycle in which the task progresses through this life cycle and goes from one state to another. For instance, a task can go through the following states in the MDP model: *submitted, scheduled, waiting, executed, finished*, and *failed*. We consider the mapping between

these states over the possible actions to find an appropriate scheduling policy as the process of the scheduling decision selection in Hadoop scheduler. Indeed, this selection is based on the derived reward for the selected action when applied on a specific state. The modeling of the scheduling decision can be described using Equation 3.3 as follows [22]:

$$\pi^* = arg \max_{\pi} E[\sum_{t=1}^{A} R(S_{(t)}, A_{(t)}, S_{(t+1)})|\pi] \tag{3.3}$$

where π^* represents the policy to be applied from one state $S_{(t)}$ to another $S_{(t+1)}$, and R contains the earned reward by following the selected action $A_{(t)}$. Following this approach, our proposed solution can estimate and compare all possible rewards that are earned when applying the actions from a given task's state. Hence, it can select the action that could reduce the risk of failure for each submitted task.

Reinforcement Algorithms Training

Given the context of adaptive policy-driven scheduling, we opt for reinforcement learning algorithms [82] to implement the proposed MDP model for Hadoop scheduler. Indeed, the reinforcement learning techniques consist of learning from past experiences (*e.g.*, scheduling policies) to predict potential future actions that can improve a given system's performance. Furthermore, they allow to consider the dynamic events occurring in a system's environment and to adjust the decisions making procedures under uncertainty. They showed good performance when applied to solve several problems in cloud computing systems including resource allocation [83], selection of virtual machines [84], job scheduling [85], and virtual machines consolidation [86]. Reinforcement learning has been successfully used to improve performance of such problems similar to the scheduling decisions modeling in the cloud.

Different reinforcement learning algorithms exist in the open literature including the Temporal Difference (TD) learning [80], Q-Learning [24], and SARSA (State-Action-Reward-State-Action) [25] that are mostly frequently used to train several systems in cloud systems [83]. Van *et al.* [87] analyzed the performance of these three algorithms in terms of state space exploration (the number of times the system changes its state after applying an action). They reported that Q-Learning and SARSA algorithms outperform the TD-learning algorithm. This can be explained by the fact that the TD-learning cannot easily exploit particular action sequences because it uses only one state network. Q-Learning is an off-policy reinforcement learning algorithm updating a Q-function according to a random policy that maximizes the expected reward. Q-function is an action-value function to estimate the expected utility of taking a given action in a given state. SARSA is an on-policy reinforcement learning algorithm selecting the next state and action according to a random policy and updating the Q-function accordingly.

The major difference between SARSA and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. The procedural form of the Q-Learning and SARSA algorithms are given in the Algorithms 3.2 and 3.3, respectively, where α represents the learning rate (α in [0,1]) and γ is the discount factor (γ in [0,1]).

For the Q-Learning algorithm (Algorithm 3.2), the value of Q-function (Q(s,a)) is initialized arbitrary (line 1). Then, the algorithm selects an action among the possible ones from the initial state (s) and observe the obtained reward (r) and the new state (s') (lines 3 to 6). Here, the algorithm selects the action maximizing the value of the Q-function that will be updated accordingly (lines 7 to 8). The algorithm repeats steps from line 5 to 8 until reaching the final state.

Alg. 3.2. Q-Learning Algorithm [24]

```
1: Initialize Q(s,a) arbitrary
2: for Repeat for each episode do
3:    Initialize s
4:    for each step of episode until s is terminal do
5:       Choose a from s using policy derived from Q (e.g., ϵ-greedy)
6:       Take action a then observe r and s'
7:       Q(s,a) = Q(s,a) + α * [ r(s,a,s') + γ * max(Q(s', a')) - Q(s,a) ]
8:       s = s'
9:    end for
10: end for
```

For the SARSA algorithm (Algorithm 3.3), the value of Q-function (Q(s,a)) is initialized arbitrary then, the algorithm initializes the first state and selects a possible action (a) (lines 1 to 5). Next, the Q-function, new state (s'), and new action values will be updated accordingly (lines 6 to 9).

Alg. 3.3. SARSA Algorithm [25]

```
1: Initialize Q(s,a) arbitrary
2: for Repeat for each episode do
3:    Initialize s
4:    for each step of episode until s is terminal do
5:       Take action a then observe r and s'
6:       Choose a and s' using policy derived from Q (e.g., ϵ-greedy)
7:       Q(s,a) = Q(s,a) + α * [ r(s,a,s') + γ * Q(s', a') - Q(s,a) ]
8:       s = s'
9:       a = a'
10:   end for
11: end for
```

Given the above facts, we choose to implement the proposed adaptive approach to select scheduling actions for the MDP model in Hadoop based on the Q-Learning and SARSA algorithms. More concretely, we evaluate the performance of these two algorithms in terms of the number of explored policies and successful policies (policies leading to task execution success). Next, we integrate the algorithm providing the best results within Hadoop scheduler. In the sequel, we present in Algorithm 3.4

the different steps followed by our proposed approach to model adaptive scheduling policies for Hadoop to avoid poor scheduling decisions and reduce tasks' failures.

Given a new task to be submitted, the proposed algorithm (Algorithm 3.4) collects data about the current status of the Hadoop cluster, then it selects which action to proceed given its current state (lines 1 to 4). According to the obtained reward, the algorithm selects a candidate policy to be applied on the new scheduled tasks (*e.g.*, process, reschedule, kill) as shown by lines 4 to 6. The obtained scheduling outcome will be stored in a database (lines 8 to 9). Here, we should mention that a submitted task can go through the following states: *submitted, scheduled, waiting, executed, finished*, and *failed* when the scheduler applies one of the following actions: *schedule, wait, reschedule, process, finish, kill*, and *fail* as shown in Figure 3.5.

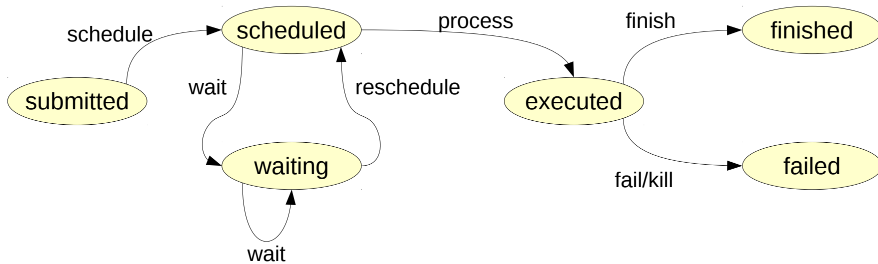

Figure 3.5: Life Cycle of a Task

After collecting the data from the cluster environment, the *Select-Action(Task, State)* function selects a possible action to be applied on a task given its current state. Next, the *MDP-Solver(Task, State, Action)* function does the mapping between the selected action and the current state in order to calculate the obtained reward while the *MDP-Solver(Task, State, Action)* function calculates the associated reward for the submitted task given its current state and the selected action to be applied. Thereafter, the *Select-Policy(Task, State, Action, data)* function will select the appropriate

policy to be processed by the scheduler. In general, a policy is characterized by the following format: *<ID_Task, Type, Locality, Current_State, Action, Next_State, Reward, TaskTracker, Outcome>*.

Alg. 3.4. Scheduling Learning Algorithm

```
1: while (There is a new task to be scheduled ) do
2:    data = Collect-Env(Cluster)
3:    /* Calculate reward associated with action using Q-Learning or SARSA */
4:    Action = Select-Action(Task, State)
5:    Reward = MDP-Solver(Task, State, Action)
6:    Policy = Select-Policy(Task, State, Action, data)
7:    /* Apply the scheduling policy and update the scheduling policies rules */
8:    Outcome = Apply-Policy(Task, State, Action)
9:    Update-Policies-Rules(Task, State, Action, Policy, Outcome)
10: end while
```

In the sequel, we present examples of scheduling policies in order to better illustrate the steps followed in Algorithm 3.4. Given a new submitted map task characterized by *ID_T* that is on the *scheduled state*, the scheduler decides that it is possible to process it on the selected TaskTracker, *Selected_TT*, where its data is located. This is because it has enough resources to process the scheduled task; according to the collected data from the cluster environment. Therefore, the generated scheduling policy will be: *<ID_T, Map, Local, Scheduled, Process, Executed, +1, Selected_TT, Running>*. Here, the MDP-Solver() function attributes a positive reward (+1) since the task is running and not rescheduled on the queue. However, when the scheduler decides to reschedule this task and to send it to the queue because the *Selected_TT* does not have enough resources, the scheduling policy will be: *<ID_T, Map, Local, Scheduled, Wait, Waiting, -1, Selected_TT, Rescheduled>*. Here, the obtained reward has a negative value (-1) because the task will experience more waiting time in the queue, which can affect the total execution time of the job to which the task belongs.

3.3.2 Adaptive Scheduling Evaluation

In this section, we present our approach to evaluate the proposed scheduling methodology and the obtained results.

Experimental Design

We train the SARSA and Q-Learning algorithms while scheduling 22,000 tasks (map and reduce tasks) to evaluate their performances in terms of the number of explored policies and the resulting outcomes when applying policies (a finished task or a failed task). To this aim, we implement a script to submit 1500 different tasks to be scheduled each 5 minutes in Hadoop. Next, we implement another script to compute the number of explored policies and their corresponding outcomes for each algorithm when integrated separately to the Hadoop scheduler.

Using these collected data, we define a "*policy success rate*" metric as the ratio between the number of policies leading to a finished task over the total number of explored polices. We measure this metric in each interval, when submitting new tasks to be scheduled, in order to assess how good is the integrated algorithm. More concretely, the number of explored policies and the policy success rate metrics allow to identify the algorithm characterized by a higher number of explored policies as well as a higher success rate. This is in order to select the algorithm allowing Hadoop scheduler to increase the number of finished tasks and avoid poor scheduling decisions. We repeat this experiment 30 times in order to evaluate the variance of the two integrated algorithms in Hadoop scheduler over time.

While performing the experiments, we collect and store data about the used scheduling policies and their corresponding outcomes in a database. To characterize the explored policies in the two algorithms, we collect the following attributes: policy

ID, locality/execution type (local or non-local), time to find the policy (time to access the database and find the policy), selected TaskTracker, policy reward (reward collected from the proposed model), number of speculative executions, number of tasks pending in a queue, policy Q-Value (obtained according to the Q-Learning or SARSA algorithm), load (number of finished, failed, killed, struggling and running tasks), available slots on selected TaskTracker, requested slots on selected TaskTracker, used slots on selected TaskTracker, frequency of policy usage, frequency of policy positive usage (policy leading to task success), frequency of policy negative usage (policy leading to task failure) and policy outcome (task final status; finished or failed).

Overall, we select these attributes because they represent the most important metrics describing the used policies that we could collect from the scheduler environment. To better understand the relationship between these metrics, we perform a multi-collinearity analysis, which checks the dependencies between variables in a given model, by computing their corresponding *Variance Inflation Factor (VIF)* values [88]. Next, a threshold value of 5 is used to categorize these metrics into two groups; correlated and non-correlated ones [88]. Specifically, policy attributes with a VIF value higher than 5 are considered as correlated. While the VIF values can vary between 0 and 10, we selected a threshold value of 5 following the recommendations provided in [88] [89]. Furthermore, we checked the obtained results when using different threshold values (*e.g.*, 4, 6 and 7) and we found out that they provide the same correlation results. Moreover, we apply another metric in order to assess the importance of these metrics on the scheduling policies. This is by applying the *MeanDecreaseGini* criteria that estimates the importance of variables on the output of a given model [88]. Metrics with higher values are considered to be the ones having more impact on the scheduling outcome of the policy.

To select the appropriate policy when there is a new submitted task for the scheduler, we deploy a procedure to select the policy to apply when there is a new submitted task. This procedure uses data about this task and the characteristics of the running workload in order to select the policy having the greatest value of positive usage, *i.e.*, the highest probability of success.

Experimental Results

We evaluate the performance of the SARSA and Q-Learning algorithms while scheduling tasks over time. Figures 3.6 and 3.7 present the obtained results of the two algorithms in terms of the number of explored policies and policy success rate, respectively, for 30 experiments with a confidence level of 95%.

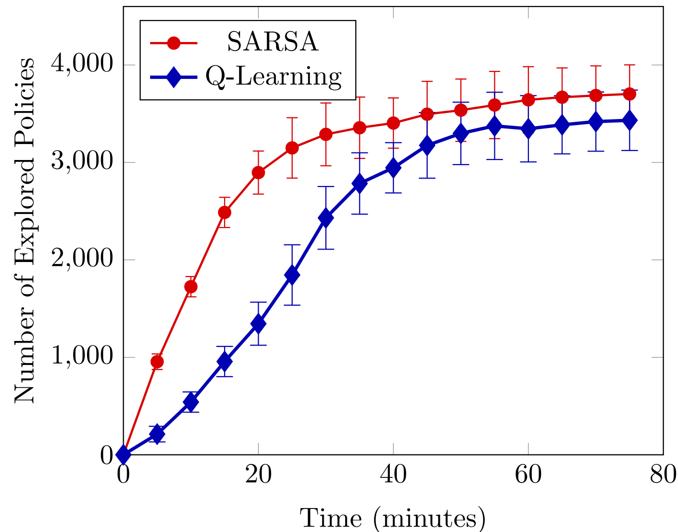

Figure 3.6: Number of Explored Policies

The obtained cumulative performance shows that the SARSA algorithm outperforms the Q-Learning algorithm in terms of the number of generated policies. For instance, Figure 3.6 shows that the SARSA algorithm explores 2896 policies, whereas

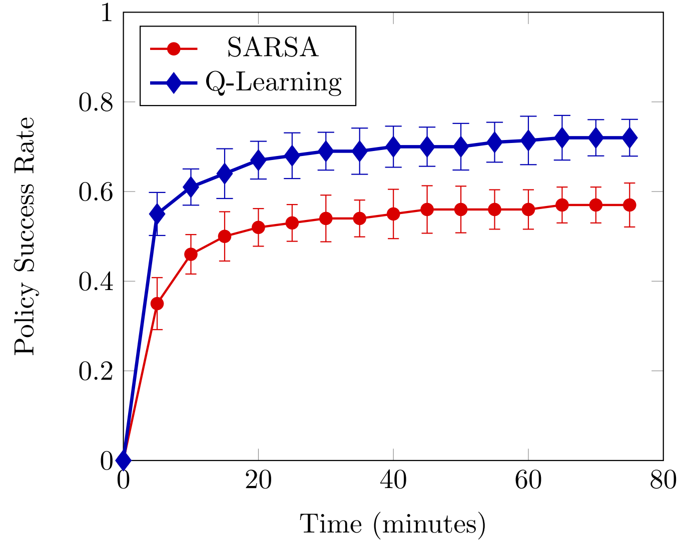

Figure 3.7: Policy Success Rate

the Q-Learning explores 1345 in 20 minutes. Furthermore, we found out that Q-Learning is characterized by a success rate of 0.67%, while SARSA achieves only a success rate of 0.52% in 20 minutes as shown in Figure 3.7. We can explain these expected results by the fact that the SARSA algorithm uses a random function to select the next state and action, and hence it can explore more policies. While the Q-Learning algorithm selects the next possible action using a function that maximizes the reward of the next action, and hence it can increase its success rate compared to the SARSA algorithm. In addition, we observed that the two algorithms provide almost the same performance after 30 minutes, meaning that they explored almost the same number of policies. This is because the two algorithms reach the maximum number of possible policies for the submitted tasks and the scheduler is mostly re-using the previously generated scheduling.

On the other hand, we investigate the correlation between the policy attributes and their corresponding outcomes using the VIF variable values. Here, we should

mention that attributes having VIF values greater than 5 are not considered in the analysis, as discussed earlier [88]. The obtained results show that the following attributes (along with their VIF values) including the policy reward (0.45), selected TaskTracker (1.34), Q-Value (0.83), load (1.41), available slots (2.07), selected slots (2.42), requested slots (2.58), used slots (2.25), frequency of policy positive (3.18)/ negative usage (3.24) have a strong correlation with the policy outcome. Next, the obtained "MeanDecreaseGini" scores of the policy attributes show that the most important attributes affecting the policy outcome (success or failure) can be ordered as follow: load, available/selected slots on selected TaskTracker, policy Q-value, frequency of policy positive/negative usage, policy reward, locality/execution type, and number of tasks in queue. Consequently, we decide to train our proposed scheduling algorithm for Hadoop using these selected attributes (those affecting a policy outcome most).

Given the performance of the two algorithms, we decide to train the proposed scheduling algorithm using the SARSA algorithm at the beginning of the experiments (to explore more scheduling policies) and then to switch to the Q-Learning algorithm (to guarantee that the scheduler explores more policies and selects the policy that gives a maximum reward). Concretely, we use the SARSA algorithm for a given interval of time; because we found that after 30 minutes, the two algorithms have almost the same results in terms of number of generated policies as shown in Figure 3.7. Thereafter, we switch and run the Q-Learning algorithm to allow the Hadoop scheduler selecting policies that can help increasing the number of finished tasks.

Summary: In this section, we presented a methodology for modeling adaptive scheduling decisions for Hadoop to reduce its failures rate and avoid making poor scheduling decisions. Our proposed methodology allows the Hadoop scheduler to select appropriate scheduling strategies that minimize the risk of failures for the submitted tasks. However, given the impact of TaskTrackers' failures on the execution of tasks, the scheduler still generates poor scheduling decisions and cannot reschedule tasks accordingly because of the fixed heartbeat-based approach used in Hadoop to track active nodes. Therefore, we present in the next section a methodology to dynamically control the communication between the TaskTrackers and JobTracker in order to quickly identify nodes' failures.

3.4 TaskTracker Failure Detection

In this section, we describe our TaskTracker failure detection methodology alongside with our evaluation approach and the obtained results.

3.4.1 TaskTracker Failure Detection Methodology

In the sequel, we present our proposed methodology to early identify failures of TaskTrackers in Hadoop followed by a description of each of its steps.

General Overview

Figure 3.8 presents our proposed methodology to dynamically track the failures of TaskTrackers in Hadoop. Given data about recently received heartbeat messages from TaskTrackers, the proposed methodology analyzes these messages and dynamically estimates the arrival time of next heartbeats based on failure occurrences on the TaskTrackers. Specifically, it adjusts the timeout interval at which a TaskTracker

node is considered as dead. When a Hadoop cluster experiences many TaskTracker failures, it is expected that the heartbeats will arrive with a delay to the JobTracker. Hence, our proposed methodology can notify the scheduler to shorten the timeout at which it controls the communication between the JobTracker and TaskTrackers. For this goal, the first step in our methodology is the analysis of previously received heartbeat messages in the JobTracker. This step is necessary to find possible correlations between the time of sending the heartbeats and the failures occurrences in the TaskTrackers. Next, we propose to apply existing algorithms from the network field to adjust the time to detect TaskTrackers failures. Following this methodology, the Hadoop scheduler can quickly detect TaskTrackers failures and avoid assigning tasks to dead nodes and resources wastage, and hence reduce task failure rates.

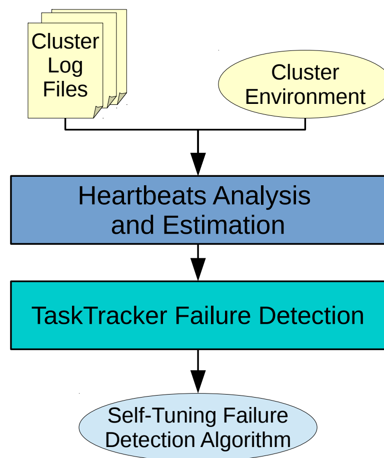

Figure 3.8: TaskTracker Failure Detection Methodology

Heartbeats Analysis and Estimation

The first step in the proposed methodology is to collect data about the previously received heartbeat messages from the cluster log files. Next, we perform an analysis to find a possible correlation between the failures of the TaskTrackers and the time of the received heartbeat messages. For instance, we investigate the impact of TaskTrackers

failures, over time, on the delay to receive the heartbeats by the JobTracker and timeout interval at which TaskTracker is considered as dead.

TaskTracker Failure Detection

The second step in our proposed methodology is to implement an algorithm to dynamically detect the failures of the TaskTrackers and adjust the timeout interval of the JobTracker. For this purpose, we use four well known algorithms from the network field: *Chen Failure Detector (Chen-FD)* [26], *Bertier Failure Detector (Bertier-FD)* [27], *ϕ Failure Detector (ϕ-FD)* [28] and *Self-tuning Failure Detector (SFD)* [29]. We choose to use these algorithms because they have been successfully applied in the network field to dynamically detect failures of nodes. Indeed, they are used to solve the problem of message synchronization between nodes in a network and they showed good performance in reducing the number of false failure detections [29]. In the sequel, we briefly describe these algorithms.

Chen Failure Detector According to Chen *et al.* [26], the expected arrival time of the next heartbeat message from a TaskTracker can be estimated by [26]:

$$EA_{(k+1)} = \frac{1}{n} \sum_{i=k-n-1}^{k} (A_i - \Delta_i \times i) + (k+1)\Delta_i \tag{3.4}$$

where the JobTracker received n heartbeat messages from the TaskTracker denoted by m_1, m_2, ..., m_n. Their actual receiving times are A_1, A_2,..., A_n according to the JobTracker's local clock. Δ_i is the sending interval of heartbeat messages. This technique provides an estimation for the expected arrival time of the next heartbeat message based on a constant safety margin. Typical values of the safety margin are between 0 and 1 [26]. Then, the arrival time of the next heartbeat message $\tau_{(k+1)}$ is

expressed in terms of $EA_{(k+1)}$ and a constant safety margin α as follows:

$$\tau_{(k+1)} = \alpha + EA_{(k+1)} \tag{3.5}$$

Bertier Failure Detector Bertier *et al.* [27] build on the description of the Chen FD approach and propose to update the safety margin α based on the variable error in the last estimation as follows [27]:

$$error_k = A_k - EA_k - delay_k, \tag{3.6}$$

$$delay_{(k+1)} = delay_k + \gamma \times error_k, \tag{3.7}$$

$$var_{(k+1)} = var_k + \gamma \times (|error_k| - var_k), \tag{3.8}$$

$$\alpha_{(k+1)} = \beta \times delay_{(k+1)} + \phi \times var_k, \tag{3.9}$$

and

$$\tau_{(k+1)} = EA_{(k+1)} + \alpha_{(k+1)} \tag{3.10}$$

In the previous equations, γ represents the importance of the new measure with respect to the previous ones, the *delay* represents the estimate margin and the *var* represents the magnitude of errors. The β and ϕ variables are used to adjust the parameter *var*. Based on [27], the β, ϕ and γ variables can be defined by the following typical values: 1, 4 and 0.1, respectively.

ϕ **Failure Detector** Hayashibara *et al.* [28] propose ϕ FD that provides update information for the heartbeat messages sending at continuous interval. The value of

ϕ is calculated as follows [28]:

$$\phi_{(t_{now})} = -\lg P_{later}(t_{now} - T_{last}), \tag{3.11}$$

Here,

$$P_{later}(t) = \frac{1}{\sigma\sqrt{2\pi}} \int_{t}^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = 1 - F(t), \tag{3.12}$$

where T_{last} represents the time of receiving of the recent heartbeat, t_{now} represents the current time, P_{later}(t) represents the probability of receiving a heartbeat more than t times in the future. μ and σ^2 represent the mean and variance parameters of the cumulative distribution function of a normal distribution *F(t).* μ and σ^2 values are estimated based on data collected from a sampling window over time. Finally, the values of ϕ for each TaskTracker will be averaged and the obtained value will be compared with a threshold Φ, which is given by the developer. Finally, the interval of sending heartbeats will be updated accordingly.

Self-Tuning Failure Detector Xiong *et al.* [29] proposed a self tuning failure detector that adjusts the next heartbeat message freshness point $\tau_{(k+1)}$ based on collected information from the environment as follows [29]:

$$\tau_{(k+1)} = SM_{(k+1)} + EA_{(k+1)}, \tag{3.13}$$

where $EA_{(k+1)}$ represents the expected arrival time of the next heartbeat message as defined in Chen-FD [26]. $SM_{(k+1)}$ is the dynamic safety margin that can be defined as follows [29]:

$$SM_{(k+1)} = SM_k + Sat_k \left\{QoS, \overline{QoS}\right\} \times \alpha, \tag{3.14}$$

here, the α variable can have a value between 0 and 1 ($\alpha \in (0,\ 1)$) as defined in Chen-FD [26] and the Sat_k variable can be defined as follows [29]:

$$Sat_k\left\{QoS, \overline{QoS}\right\} = \begin{cases} \pm\beta, & QoS > \overline{QoS}; \\ 0, & QoS \leq \overline{QoS}. \end{cases} \tag{3.15}$$

According to the previous equation, $\mathrm{Sat}_k\left\{QoS, \overline{QoS}\right\}$ can have β, $-\beta$, or 0 value. The parameter β represents the constant adjusting rate and it can be dynamically chosen such that the value of $\beta \in (0,\ 1)$. The procedure used to adjust the parameters of SFD is described in Algorithm 3.5 [29]:

Alg. 3.5. A Method to Adjust Parameters in SFD [29]

1: ***Begin***
2: ***Initialization***
3: $\overline{T_D}$: *Set the detection time*
4: $\overline{MR}$: *Set the mistake rate*
5: *Set the initial safety margin value for* SM_1
6: *Set the constant parameters* α *and* β
7: ***Step 1: Get the relevant data***
8: *get the output* $QoS(T_D,\ MR)$
9: ***Step 2: Get the feedback information***
10: ***If*** $T_D > \overline{T_D}$ *and* $MR < \overline{MR}$ ***then***
11: $Sat_k\left\{QoS, \overline{QoS}\right\} = \beta$;
12: ***endif***
13: ***If*** $T_D < \overline{T_D}$, $MR < \overline{MR}$ ***then***
14: $Sat_k\left\{QoS, \overline{QoS}\right\} = 0$;
15: ***endif***
16: ***If*** $T_D < \overline{T_D}$, $MR > \overline{MR}$ ***then***
17: $Sat_k\left\{QoS, \overline{QoS}\right\} = -\beta$;
18: ***endif***
19: ***If*** $T_D > \overline{T_D}$, $MR > \overline{MR}$ ***then***
20: *this SFD cannot get high QoS requirements and goes to line 25*
21: ***endif***
22: ***Step 3: Adjust parameters***
23: *Send* $Sat_k\left\{QoS, \overline{QoS}\right\}$ *to the SFD*
24: *Adjust SFD relevant parameters based on the value of* $Sat_k\left\{QoS, \overline{QoS}\right\}$
25: ***End***

We adapt the existing implementations of these algorithms in Hadoop scheduler to quickly detect the failures of TaskTrackers. Based on information about recently received heartbeats messages and TaskTracker nodes failure occurrences, the four algorithms can adjust the interval timeout at which the JobTracker considers a TaskTracker as dead. More precisely, these algorithms use the equations for each algorithm to estimate the expected arrival times of future heartbeat messages based on information about the arrival time of previously received ones. Next, we evaluate the performance of the four algorithms in terms of detection time of the TaskTracker failures over time when integrated in the Hadoop scheduler. We select the algorithm providing the shortest time to quickly detect TaskTrackers' failures. Algorithm 3.6 describes the steps followed by our proposed methodology to quickly detect TaskTrackers' failures and adjust the sending of heartbeats between the JobTracker and TaskTrackers. We first start by collecting data about previously received heartbeat messages from TaskTrackers as shown by line 1 in Algorithm 3.6. Next, these collected data will be used as inputs to the four selected algorithm to measure their performance results in terms of detection times and mistake rates over time. The algorithm providing the best results will be integrated within Hadoop to dynamically adjust the timeout interval values (lines 2 to 7). For each new interval of communication, the integrated algorithm estimates the expected arrival times of heartbeat messages from the TaskTrackers and notifies the scheduler about the new value of the timeout interval to decide whether a TaskTracker node is dead or not (lines 8 to 14).

3.4.2 TaskTracker Failure Detection Evaluation

In this section, we present our approach to evaluate the proposed methodology to detect TaskTracker failures in Hadoop along with the obtained results.

Alg. 3.6. Self-Tuning Failure Detection Algorithm

```
1: HB-data = Collect-data(TaskTracker, heartbeats)
2: /* Apply the algorithms to control the communication between JobTracker-
   TaskTrackers */
3: Adaptive-Algorithms(HB-data, algorithms)
4: Performance = Measure-Performance(algorithms)
5: /* Select Algorithm giving best results (detection time and mistake rate) */
6: Algorithm = Select-Model(algorithms, Performance)
7: /** Integrate the adaptive algorithm within the scheduler **/
8: while (For each new interval time of communication) do
9:    HB-next-arrival = Estimate-arrival(Algorithm, TaskTrackers, HB-data)
10:   HB-median= Get-Median(TaskTrackers, HB-next-arrival)
11:   /* Update the next interval timeout of the following communication */
12:   Update-Communication(JobTracker, TaskTrackers, HB-median)
13:   Notify-Scheduler(JobTracker, TaskTrackers, HB-median)
14: end while
```

Experimental Design

We evaluate the performance of the four selected algorithm, described in Section 3.4.1, when integrated within Hadoop and the basic algorithm used in Hadoop to track the failures of TaskTrackers. According to [29], the *detection time* (T_D) and the *mistake rate* (*MR*) represent the most important parameters that can describe the fault-tolerance schemes in distributed systems [29]. In this context, Xiong *et al.* [29] claim that there is a correlation between the T_D and the *MR*. For instance, a shorter T_D is likely to translate into low/high precision. Therefore, we will evaluate the compromise between the T_D and the *MR* in our proposed methodology. More precisely, we inject different types of failures to the TaskTrackers (*e.g.*, slowing down/dropping the network, killing/suspending TaskTrackers) and measure the T_D and the *MR* of the selected algorithms while catching these failures over time. We inject the failures at regular times over the communication intervals between the JobTracker and TaskTrackers (*e.g.*, 2 minutes after the beginning of a new interval). In the following interval at which we inject failures, we implement a procedure to revive dead nodes

while changing the time of the recovery of TaskTrackers (*e.g.*, 1, 2 and 3 minutes after the beginning of a new interval). This is in order to evaluate the impact of different recovery times on the mistake rate (*i.e.*, the number of times that the scheduler considers an alive node as dead). In addition, we vary the failure rates to better assess the performance of the selected algorithms in terms of detection times and mistake rates under different amounts of injected failures. The injected failure rates are 10%, 20%, 30%, 40%, and 50% of TaskTrackers.

Experimental Results

We analyze the performance of the four selected algorithms to detect TaskTrackers failures compared to the basic algorithm used in Hadoop. Figures 3.9 and 3.10 present the obtained performance results of the algorithms. The two figures show the variation of the algorithms in terms of detection time over time when the same number of failures are injected. Here, we only discuss the results of the algorithms under 30%, and 50% TaskTracker failure rates because the selected algorithms follow the same trend for the other training rates (*e.g.*, 10%, 20%, and 40%).

Overall, we report that the *SFD* algorithm outperforms the other algorithms; it is characterized by a smaller detection delay over time for the same number of injected failures. For instance, the basic algorithm is providing the worst performance (8 minutes as detection time). Moreover, we notice that the *Bertier-FD* and the ϕ-*FD* algorithms are characterized by detection times very close to that of the basic Hadoop algorithm, under different failure rates. We can explain these results by the fact that these two algorithms rely and depend on the amount of collected information about heartbeat messages to identify failures of nodes. To compute a more adaptive normal distribution function, the ϕ-*FD* requires a large window size to obtain more data

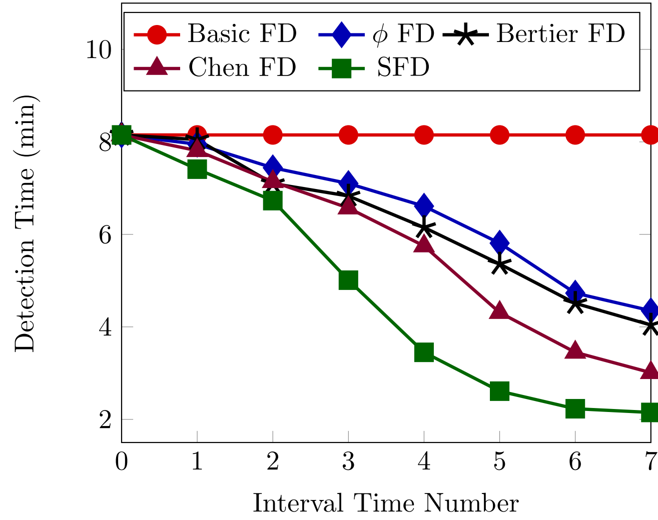

Figure 3.9: Detection Time under 30% Failure rate

for the normal distribution function. However, collecting more data will negatively impact the detection time and the overall performance of Hadoop scheduler. The *Bertier-FD* algorithm is characterized by functions that do not tune its parameters. Hence, the window size of data does not affect the behavior of the algorithm over time and cannot reduce its detection time.

The *SFD* algorithm is providing the best performance when compared to the other algorithms. Furthermore, the performance of the *Chen-FD* algorithm is close to that of *SFD*. This is because these two algorithms are using the same function to estimate the expected arrival time of the next heartbeat as explained in Algorithm 3.5. The main difference between the two algorithms is the function used to update the safety margin. While the *SFD* adjusts and updates the safety margin based on an adaptive function according to the occurrence of failures in the cluster, the *Chen-FD* uses a constant safety margin. As a consequence, the *SFD* algorithm is able to estimate the arrival times for receiving the heartbeat messages in less time compared to the *Chen-FD*.

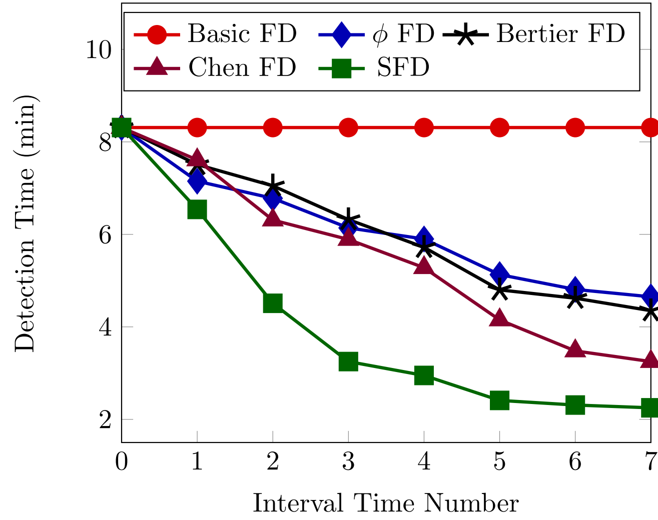

Figure 3.10: Detection Time under 50% Failure rate

Next, we evaluate the performance of the studied failure detection algorithms in terms of mistake rate for different recovery times of TaskTrackers (1, 2, and 3 minutes) and different failure rates (from 10% to 50%). Overall, we observe that the performance of the algorithms are following the same trend for the three different recovery times. Hence, we only discuss the results for a recovery time of 2 minutes in Table 3.5. Here, we report that the *Bertier-FD* and the ϕ-*FD* algorithms identify TaskTrackers' failures while making less errors compared to the *SFD* algorithm, which is making more mistakes over time. Indeed, the intervals timeouts of the *Bertier-FD* and the ϕ-*FD* algorithms are longer than the one of the *SFD*.

Consequently, we can report that there is correlation between the mistake rate and the detection time of the algorithms: the longer the detection time, the less would be the mistakes occurrence and vice versa. With a failure rate of 40% and a recovery time of 2 minutes, the Basic FD is characterized by a constant detection time (8 minutes according to Figures 3.9) and a normalized value of wrong failure detection rate equal to 0.48 (see Table 3.5). Whereas the *SFD* algorithm is characterized by a

Table 3.5: Normalized Values of Wrong Failure Detection Rate of TT

	TT Recovery Time (2 min)				
Failure Rate	**Basic FD**	**ϕ FD**	**Bertier FD**	**Chen FD**	**SFD**
10%	-1.41	-1.17	-1.06	-1.18	-1.27
20%	-0.46	-0.65	-0.62	-0.51	-0.47
30%	0.14	-0.16	-0.42	-0.28	-0.16
40%	0.48	0.71	0.88	0.65	0.56
50%	1.23	1.28	1.23	1.34	1.34

normalized value of wrong failure detection rate equal to 0.56 (see Table 3.5) and a decreasing detection time (which reaches 2 minutes: see Figures 3.9). In this context, we can report that the obtained results are similar for the other algorithms in terms of failure rates, and recovery times. At this level, we should mention that there are some lost heartbeat messages because of network conditions and not because of a failure of a TaskTracker.

In light of these results, we select the ϕ-*FD* and *SFD* algorithms to implement our proposed methodology to detect TaskTracker failures within Hadoop, as described in Section 3.4.1. This is to evaluate the scheduler performance under shorter detection times and lower mistake rates.

Summary: Overall, we presented in this section a methodology for TaskTrackers failures detection to address the limitations of Hadoop scheduler when identifying failures of nodes. Given data about previously received heartbeat messages, our TaskTrackers failures detection methodology can dynamically adjust the timeout interval at the JobTracker level to decide whether a node is dead or not.

3.5 Summary

To address the limitations of Hadoop schedulers to reduce failures of tasks and TaskTrackers, we presented in this chapter novel approaches to track failures and to generate adaptive scheduling decisions for Hadoop. For instance, using data about previously executed tasks in Hadoop, we proposed a predictive algorithm based on machine learning algorithm (Random Forest) to early identify the failures of tasks and adjust the decisions of the Hadoop scheduler accordingly. We also presented a methodology for modeling adaptive scheduling decisions for Hadoop to reduce its failures rate and avoid making poor scheduling decisions using reinforcement learning algorithms (*e.g.*, Q-Learning and SARSA algorithms). To address the limitations of Hadoop scheduler when identifying failures of nodes, we presented a method for TaskTrackers failures detection to dynamically adjust the timeout interval at the JobTracker level to decide whether a node is dead or not. In the next chapter, we will implement the three methods together in order to evaluate the performance of the proposed approaches to reduce failures rates in Hadoop.

Chapter 4

ATLAS: AdapTive faiLure-Aware Scheduling

Building on the findings of the previous chapters, we present in this chapter the implementation of our Adaptive faiLure-Aware Scheduling (ATLAS) algorithm for Hadoop that can be integrated with the existing Hadoop schedulers (*e.g.*, FIFO, Fair, and Capacity). To illustrate the usefulness and benefits of our proposed scheduling algorithm, we conduct a large empirical study on 100-nodes Hadoop and 1000-nodes Hadoop clusters deployed on Amazon EMR to evaluate the performance and scalability results of ATLAS. This is done by comparing the performance of ATLAS to those of the three Hadoop schedulers (FIFO, Fair, and Capacity). Results show that ATLAS outperforms FIFO, Fair, and Capacity schedulers, and it could reduce the failures rates for jobs and tasks by up to 49% and 67%, respectively. Furthermore, it could reduce the total execution times of jobs and tasks by 35% and 42%, respectively. Consequently, the CPU and memory usage are reduced by 25% and 24%, respectively.

4.1 ATLAS Implementation

In this section, we present a general overview of ATLAS implementation alongside with an algorithm allowing its integration on Hadoop.

Based on the findings of Chapter 3, ATLAS shall predict the scheduling outcomes of tasks using data about the tasks and the scheduler environment according to the proposed “Task Failure Detection” methodology presented in Section 3.2.1. To cope with the dynamic nature of cloud environment, ATLAS can provide better scheduling decisions on the fly based on the generated policies by the “Adaptive Scheduling Learning” methodology discussed in Section 3.3.1. Instead of the fixed heartbeat-based failure detection approach used in Hadoop to track failures of nodes, ATLAS uses the proposed “TaskTracker Failure Detection” methodology described in Section 3.4.1. Overall, we build ATLAS with the existing Hadoop schedulers to provide adaptive scheduling decisions to reduce the failures rates and improve the applications running on Hadoop. The remainder of this section elaborates more on the procedural description of our proposed ATLAS algorithm.

We present in Algorithm 4.1 the different steps followed by ATLAS: our proposed adaptive and failure-aware scheduling algorithm:

The first step consists of dynamically adjusting the communication between the JobTracker and TaskTrackers according to the “TaskTracker Failure Detection” approach (Section 3.4.1). To do so, ATLAS requires the status of the Hadoop cluster in terms of the number of existing nodes and their status. Next, the “TaskTracker Failure Detection” approach collects data about previously received heartbeat messages and estimate the arrival of the next ones. Then, it uses one of the selected algorithms, presented in Section 3.4.1, to update the communication interval timeout between

the JobTracker and TaskTrackers to quickly detect nodes' failures. This proactive algorithm is running in parallel with the rest of the ATLAS algorithm (lines 1 to 5).

To early identify a potential scheduling outcome of a new submitted task, ATLAS collects attributes of this map/reduce tasks that are considered as inputs for the "Task Failure Detection" approach. Here, we should mention that we collect different data about the map and reduce tasks to generate the predictions. This is due to the difference between their attributes and the impact of their attributes on the scheduling outcome of a task (as explained in Section 3.2.1). Next, the proposed "Task Failure Detection" approach (Section 3.2.1) provides predictions whether this task will finish or fail (line 11).

To select a good scheduling decision, ATLAS uses the proposed "Adaptive Scheduling" approach to obtain a candidate policy that can be either a *process*, a *reschedule* or a *kill* policy. For example, it will select whether to process or delay a task predicted to finish, to reschedule or to kill a task predicted to fail, etc. In this context, we can define a process policy as a request to the scheduler to execute the submitted tasks, while a reschedule policy as a request to delay the execution of the task wait until its success conditions are met. More precisely, it consists of resubmitting the task to the queue until there exist circumstances/specifications in the scheduler environment leading eventually to a successful execution of that task. Finally, a kill policy denotes a request to kill an executed or a waiting task.

ATLAS uses the "Adaptive Scheduling" method to determine a candidate scheduling policy for a task predicted to finish (line 13). In the case of a process policy (line 14), ATLAS will check the availability of the requested TaskTracker and DataNode where the scheduler decides to schedule this task (line 15). When the TaskTracker and DataNode are available (line 16), ATLAS proceeds with the execution

of the task (line 18). Thereafter, ATLAS stores the outcome result after applying the candidate policy (*e.g.*, finished/failed task, environment status, used resources) in a database where the scheduling polices for Hadoop are saved (as shown in our proposed "Adaptive Scheduling" methodology discussed in Section 3.3.1).

In the case of non-availability of the selected TaskTracker and DataNode, ATLAS sends the task to the queue to be waiting for its turn (line 21). ATLAS runs the policy and assigns a penalty to the task if the candidate policy is to reschedule the task (line 25) or to kill the task (line 28). At this level, we should mention that the tasks tagged with penalties can wait in the queue more than others because of their lower execution priority until enough resources are available to enable their speculative execution on multiple nodes. Next, it stores the result of the scheduling policy within the database.

ATLAS will execute a task speculatively when there are enough resources on many nodes (line 33) in the case of a task predicted to fail (line 31) and a process policy (line 32) generated by the proposed "Adaptive Scheduling" approach presented in Section 3.3.1. Indeed, the speculative execution can speed up the execution of the task and increase the chances of its success. However, ATLAS will reschedule or kill a task when the generated policy is either to reschedule (line 38) or to kill (line 41).

While implementing ATLAS, we carefully check that its decisions do not violate its specification, they are controlled by a time-out metric from Hadoop's base scheduler.

4.2 ATLAS Evaluation

In this section, we present the experimental design used to evaluate ATLAS followed by a description of the obtained performance and scalability results.

Alg. 4.1. ATLAS Algorithm

```
1: while (Cluster is running) do
2:    Cluster-Status = Get-Status-Cluster(Cluster)
3:    /* Adjust the Communication between JobTracker (JT) and TaskTrackers (TTs) */
4:    TaskTracker-Failure-Detection(Cluster-Status, JT,TTs)
5: end while
6: /* Lines 1 to 5 run in parallel with the rest of the algorithm */
7: while (There are free slots on TTs) do
8:    while (There is a new task to be scheduled) do
9:       /* Select TT and DN where to execute the task by basic scheduler functions */
10:      TT-DN = Machine-Selection-Basic-Function-Scheduler(Task)
11:      Predicted-Status = Task-Failure-Detection(Task, TT)
12:      if (Predicted-Status == "SUCCESS") then
13:         Policy = Adaptive-Scheduling(Task)
14:         if (Policy == "Process") then
15:            Check-Availability(TT,DN)
16:            if (TT and DN are available) then
17:               /* Execute Task in the TaskTracker TT */
18:               Execute(Task, TT, Policy)
19:            else
20:               /* Resubmit Task since it will fail in such conditions */
21:               Send to Queue + Penalty
22:            end if
23:         end if
24:         if (Policy == "Reschedule") then
25:            Send to Queue + Penalty
26:         end if
27:         if (Policy == "Kill") then
28:            Kill(Task)
29:         end if
30:      end if
31:      if (Predicted-Status == "FAILURE") then
32:         Policy = Adaptive-Scheduling(Task)
33:         if (Policy == "Process") and (There are Enough Resources on Nodes) then
34:            /* Launch Many Speculative Instance of Task */
35:            Execute-Speculatively(Task,N,Policy)
36:         end if
37:         if (Policy == "Reschedule") then
38:            Send to Queue + Penalty
39:         end if
40:         if (Policy == "Kill") then
41:            Kill(Task)
42:         end if
43:      end if
44:   end while
45: end while
```

4.2.1 Experimental Design

In the following, we describe the design of the cluster, workload, and injected failures used to assess the performance of ATLAS, as well as the collected performance metrics.

Cluster

We create two Hadoop clusters to evaluate the performance of ATLAS at two scales. Indeed, we create a 100-nodes Hadoop 1.2.0-cluster and a 1000-nodes Hadoop 1.2.0-cluster on Amazon EMR. Here, the second created cluster is used to show the benefits of ATLAS in a very large cluster where there are enough failures to both learn from and also avoid. In each cluster, one node represents the master, another node is the secondary master node replacing the master in the event of failure, and the rest of nodes are slave nodes. The instantiated nodes in the created clusters have different characteristics to obtain a cluster similar to those in a real cloud cluster. Particularly, we select different types of nodes from Hadoop Amazon EMR machines' list including *m3.large* (30% nodes), *m4.xlarge* (30% nodes), and *c4.xlarge* (40% nodes) [1]. Table 4.1 summarizes the characteristics of each type of node. Furthermore, we use different slot configuration for the nodes to get different Hadoop nodes having different capacities and supporting different workload.

Table 4.1: Amazon EC2 Instance Specifications [1]

Machine Type	**vCPU**	**Memory (GiB)**	**Storage (GB)**	**Network Performance**
m3.large	1	3.75	4	Moderate
m4.xlarge	2	8	EBS-Only	High
c4.xlarge	4	7.5	EBS-Only	High

Workload

We run different workloads on the created cluster to obtain a heterogeneous workload similar to the one running in a real-world cluster. To this aim, we perform a quantitative and qualitative analysis to get data about the Hadoop jobs executed on a real cluster where different Hadoop jobs were executed in a Google cluster [90]. Here, we should mention that the Google traces include scheduling and execution details about different tasks and jobs, including Hadoop jobs. We select two different types of jobs to be running on Hadoop: single jobs (*e.g.*, *WordCount*, *TeraGen*, *Sort*, and *TeraSort* [75]), and chained jobs (sequential, parallel, and mix chains) composed of Hadoop single jobs. Each job has a different number of map and reduce tasks to obtain a heterogeneous workload. Moreover, the number of jobs in the chained jobs is different in order to evaluate their impacts on ATLAS.

Injected Failures

We create different scenarios to evaluate the performance of ATLAS under different types and rates of failures. To do so, we use the AnarchyApe tool [91] to inject several failures to the created cluster. More precisely, we inject failures to the scheduled tasks and to the TaskTracker and DataNode nodes, input data, and network in the cluster. For example, the injected failures include kill/suspend TaskTrackers, DataNodes; disconnect/slow/drop network; and randomly kill/suspend threads within the TaskTrackers in the running executions. On the other hand, we use the public traces of Google to determine a failure rate that may encounter a real Hadoop cluster [90]. This is by determining the number of failed jobs and tasks belonging to these traces. Overall, we found out that a typical cluster in a real environment can experience a failures rate as high as 40%. Therefore, we decided to inject different failures at rates

ranging from 5% to 40% [92].

Collected Performance Metrics

In order to evaluate the performance ATLAS, we measure the number of finished and failed jobs and tasks (map and reduce), the total execution times of jobs and tasks, and the amount of used resources. For the resources, we collected data about the used CPU, memory and HDFS Read/Write for the executed jobs and tasks. We repeated this experiment 30 times to measure and compare the performance of ATLAS using the exact same jobs, tasks and data. Next, we calculate the upper and lower bounds of the obtained results with a confidence level of 95%.

While implementing ATLAS on Hadoop, we integrate our three proposed approaches: (1) task failure prediction, (2) adaptive scheduling, and (3) TaskTracker failure detection. Particularly, we use respectively, the Random Forest (RF) algorithm, the SARSA and the Q-Learning algorithms, and the ϕ-FD and the SFD algorithms. In the sequel, we present the performance results of ATLAS using the following configurations: (1) ATLAS with RF, (2) ATLAS with RF and MDP, (3) ATLAS with RF, MDP and ϕ-FD, and (4) ATLAS with RF, MDP and SFD.

4.2.2 Performance Analysis Results

We evaluate the performance of ATLAS by running 2000 Hadoop jobs (10% single jobs, 30% sequential chains, 30% parallel chains, and 30% mix chains), and around 50,000 map/reduce tasks when integrated respectively with the FIFO, the Fair, and the Capacity schedulers. In the sequel, we present the obtained results in terms of the number of finished and failed jobs/tasks, execution times, and resources utilization.

Number of Finished Jobs/Tasks

Figures 4.1, 4.2, and 4.3 present the performance of ATLAS algorithm when built on top of the FIFO, Fair and Capacity schedulers in terms of the number of finished jobs, map, and reduce tasks, respectively. When evaluating the performance of ATLAS, we found out that ATLAS+RF algorithm could increase the total number of finished jobs, map and reduce tasks compared to the basic implementations of the three schedulers. Moreover, we noticed that ATLAS+RF+MDP is characterized by a higher number of finished jobs and tasks when compared to both the basic scheduling and the ATLAS+RF algorithms. These results were expected because, the ATLAS+RF algorithm allows the scheduler to early identify tasks failures while, the ATLAS+RF+MDP algorithm could identify and quickly reschedule the potential failed tasks accordingly.

Next, we analyze the performance of ATLAS in terms of TaskTracker failures identification, when adding ATLAS+RF+MDP+ϕFD and ATLAS+MDP+SFD algorithms on Hadoop. Overall, we noticed that both ATLAS+RF+MDP+SFD and ATLAS+RF+MDP+ϕFD algorithms outperform the ATLAS+RF+MDP, ATLAS+RF, and the basic algorithms for the FIFO, Fair, and Capacity schedulers.

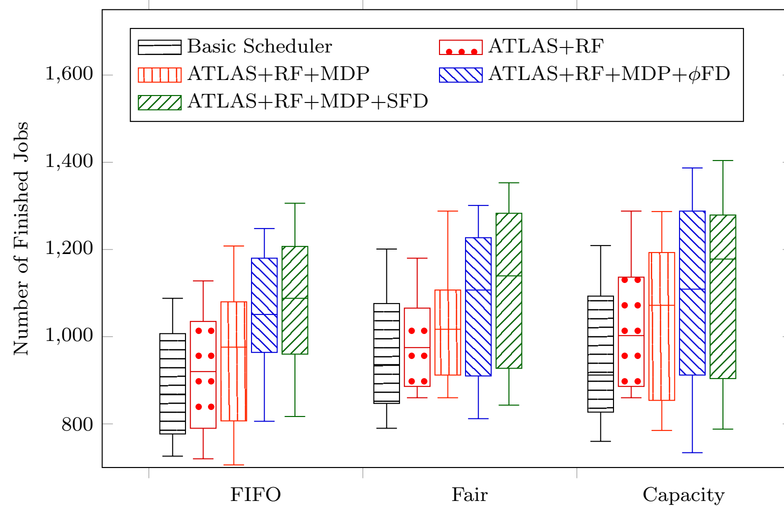

Figure 4.1: Finished Hadoop Jobs

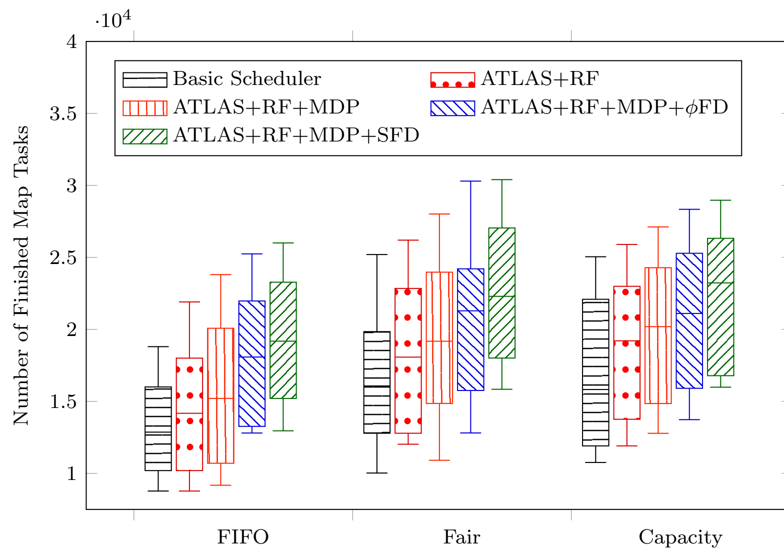

Figure 4.2: Finished Map Tasks

We can explain these results by the fact that Algorithm 3.6 allows to early identify TaskTrcakers' failures and to avoid submitting tasks to potentially dead nodes. Hence, it allows the early rescheduling of tasks on other nodes in order to increase the number of finished jobs and tasks. At this level, we observed also that the ATLAS+RF+MDP+SFD algorithm achieves better performance compared to the

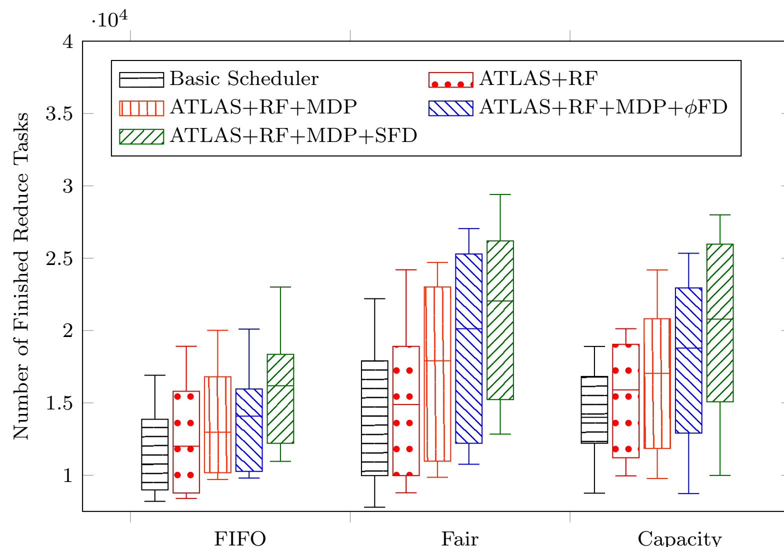

Figure 4.3: Finished Reduce Tasks

ATLAS+RF+MDP+ϕFD algorithm when integrated in Hadoop scheduler. As a result, we can claim that although it can make more wrong failures while detecting TaskTracker failures, the SFD algorithm allows to quickly detect TaskTrackers' failures and dynamically adjust the timeout to detect these failures in comparison with the ϕFD algorithm.

Furthermore, we found out that the different implementations of the FIFO and Fair schedulers achieve better performance in comparison with those of the Capacity scheduler. This is because the Capacity scheduler forces the killing of tasks consuming a large amount of memory more than expected. The obtained results show that the improvement at the task level is higher than at the job level due to the tight dependency between task scheduling and job outcomes. Indeed, the failure of a single task can cause the failure of the whole job.

In summary, ATLAS could increase the number of finished tasks by up to 61% when integrated with the Fair scheduler (see ATLAS+RF+MDP+SFD-Fair in Figure 4.3). At the job level, ATLAS could increase the number of finished jobs by up

to 55% when integrated with the Fair scheduler (see ATLAS+RF+MDP+SFD-Fair in Figure 4.1). In addition, we noticed that ATLAS increases the number of finished single and chained jobs. Precisely, the number of successful single jobs is higher than the number of successful chained jobs (due to the dependency between the jobs belonging to the same chain).

Overall, we can conclude that ATLAS performs better when integrated with the Fair scheduler since it is the "winner" compared to the other implementations of ATLAS (with FIFO and Capacity schedulers).

Number of Failed Jobs/Tasks

Figures 4.4, 4.5, and 4.6 present the obtained results, in terms of the number of failed jobs, map, and reduce tasks for the three schedulers together (as shown in the x-axis: FIFO, Fair, and Capacity), with a confidence level of 95%. Overall, we noticed that ATLAS is characterized by a lower number of failed tasks. Concretely, ATLAS could reduce the number of failed tasks by up to 67% (see ATLAS+RF+MDP+SFD-Capacity in Figure 4.6). Here, we should mention that ATLAS could efficiently reschedule the reduce tasks because they failed due to the failure of their corresponding map tasks. Overall, it could reduce the number of failed jobs by up to 49% (see ATLAS+RF+MDP+SFD-Capacity in Figure 4.4).

In the context of failure-aware scheduling, we can conclude that ATLAS achieves a good performance by sharing failure information between components in a Hadoop cluster. Also, we can claim that ATLAS outperforms the existing scheduling algorithms of Hadoop because of its adaptive scheduling mechanisms to adjust the existing

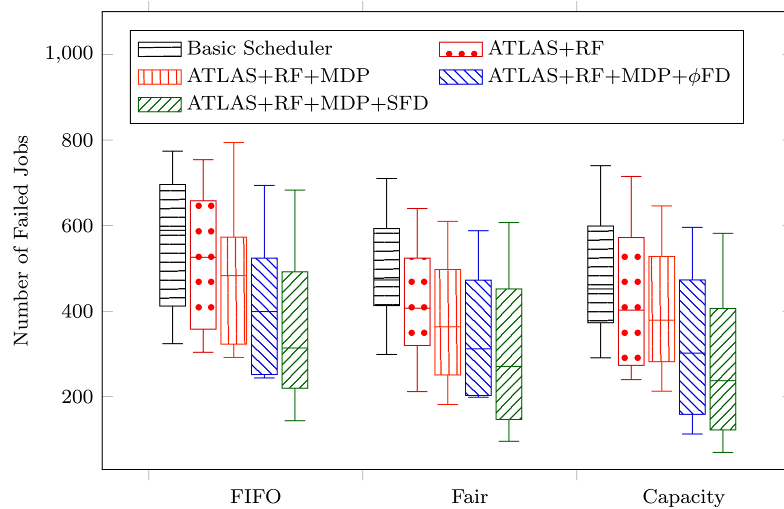

Figure 4.4: Failed Hadoop Jobs

scheduling strategies. In summary, ATLAS outperforms the existing Hadoop scheduling algorithms in terms of number of failed jobs and tasks. Furthermore, it shows better results when integrated with the Fair scheduler.

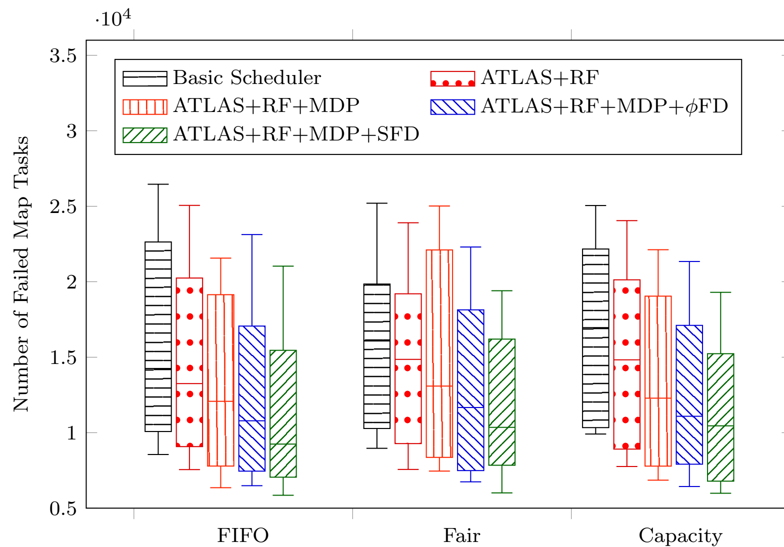

Figure 4.5: Failed Map Tasks

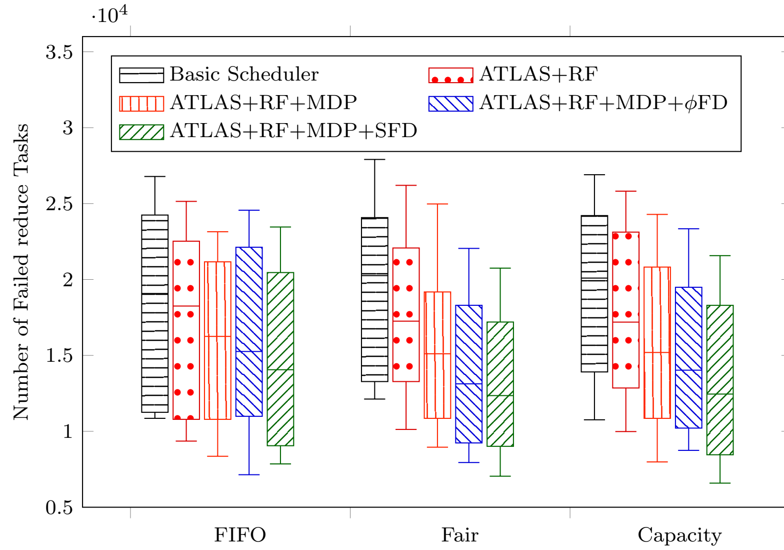

Figure 4.6: Failed Reduce Tasks

Execution Times of Jobs/Tasks

By reducing the number of failed attempts of the scheduled map and reduce tasks, ATLAS can reduce the total execution time of the executed tasks. Figures 4.7 and 4.8 present the execution times of the jobs, tasks (map and reduce), respectively. Here, we observed that ATLAS outperforms the other existing schedulers in terms of total execution time of the map/reduces tasks. Consequently, it could reduce the total execution time of their corresponding jobs.

Overall, we notice that ATLAS can decrease the execution times of tasks by 3 minutes (see ATLAS+RF+MDP+SFD-Capacity in Figure 4.8). While, it can reduce the total execution times of jobs on average by 10 minutes, which represents a 40% reduction on the total execution time of these jobs (see ATLAS+RF+MDP+SFD-Capacity in Figure 4.7). Furthermore, one of the benefits of ATLAS is its ability to reduce the execution times of long-running tasks from 30-40 minutes to less than 20 minutes. This represents approximately a 50% reduction allowing to improve the performance of applications running on Hadoop. One more benefit of ATLAS is that by reducing the number of failures, it can largely compensate its generated overhead

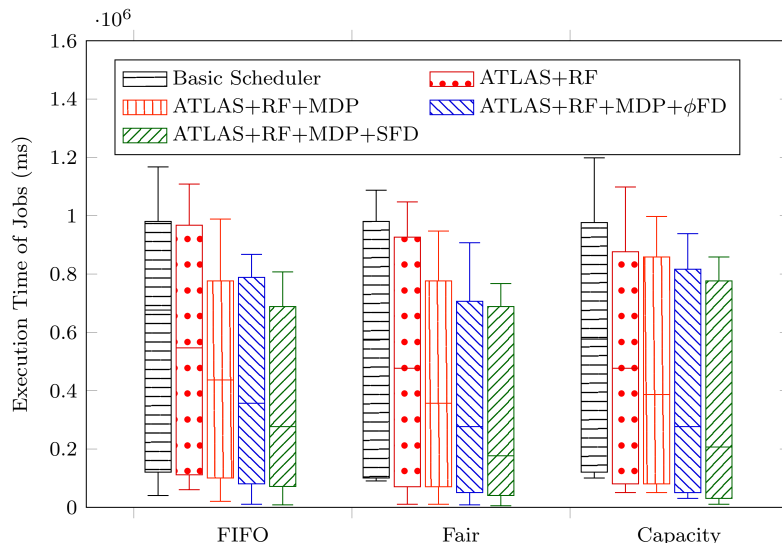

Figure 4.7: Exection Time of Jobs

(*e.g.*, training time, adjusting time, etc). Overall, we can conclude that ATLAS is able to reduce the overall execution times of tasks and jobs in Hadoop when integrated with the three existing Hadoop schedulers.

Resources Utilization of Jobs/Tasks

Another Benefit of integrating ATLAS within Hadoop schedulers is reducing the total amount of resources consumed by its applications. Tables 4.2, 4.3, and 4.4 present the obtained results of ATLAS when built on top FIFO, Fair and Capacity schedulers, respectively, in terms of CPU, memory and HDFS Read/Write usage. These results are collected from the log files of the created cluster, where each task is tagged with the amount of used CPU, memory, and HDFS Read/Write. Overall, we found that the Hadoop scheduler can use less resources using scheduling policies of ATLAS, than those used the existing implementations of the FIFO, Fair and Capacity schedulers. Indeed, ATLAS could reduce the total amount of consumed resources on average by: 25% for CPU, 24% for memory, and 31% for HDFS usage. These results were expected

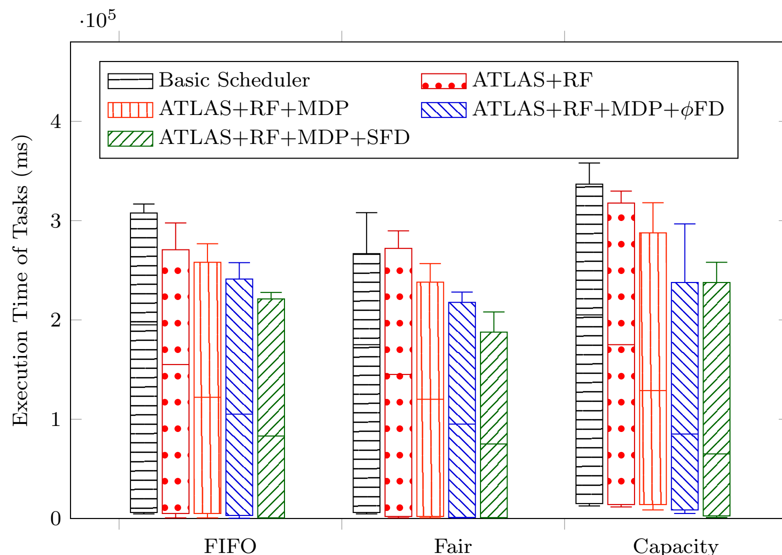

Figure 4.8: Exection Time of Tasks

because the correlation between the number of failed tasks and jobs and the cluster utilization. Overall, we can conclude that by early identifying the failure of tasks and rescheduling them, ATLAS can improve the resource utilization of the cluster.

Long-Execution of Jobs/Tasks

To further show the benefits and practical usefulness of our scheduler ATLAS, we perform an experiment to evaluate its performance after an execution period of 3 days when scheduling 120,000,000 jobs and 350,000,000 tasks. Figures 4.9 and 4.10 present the number of failed jobs, and tasks (map and reduce) of the different implementations of ATLAS. At this level, we found out that the ATLAS+RF+MDP+SFD implementation achieves better performance compared to other implementations of ATLAS. Overall, it could reduce the number of failed jobs by up to 44% and the number of failed tasks by up to 49% for the three schedulers. Consequently, it could reduce the total execution times of the scheduled jobs and tasks, particularly the long running execution ones. Hence, it was able to reduce the amount of used resources in

Table 4.2: Resources Utilization of the FIFO Scheduler

Job/Task	Scheduler	FIFO				
		Basic	**ATLAS RF**	**ATLAS RF MDP**	**ATLAS RF MDP ϕ-FD**	**ATLAS RF MDP SFD**
	Resource	**Avg.**	**Avg.**	**Avg.**	**Avg.**	**Avg.**
Job	**CPU**	15307	14830	13720	13297	12415
	Memory	12571	11784	9964	9451	7730
	HDFS Read	10930	9845	8341	8103	8005
	HDFS Write	9747	9125	8962	8397	8321
Task	**CPU**	4823	4723	4674	4479	4285
	Memory	3340	3308	3269	3017	2642
	HDFS Read	1968	1845	1732	1628	1533
	HDFS Write	1972	1907	1855	1794	1624

Units: CPU (ms), Memory (10^5 bytes), HDFS Read (10^3 bytes), HDFS Write (10^3 bytes)

the created Hadoop cluster.

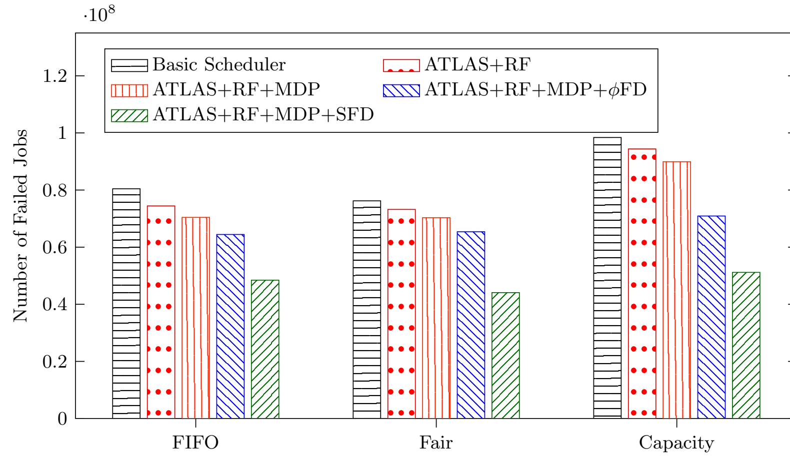

Figure 4.9: Number of Failed Job over 3 Days

We can explain the obtained results by the fact that ATLAS learns more scheduling decisions over time. Indeed, the learning time has a direct impact on the quality of the applied actions in ATLAS; the more data the scheduler collects, the better the scheduling decisions would be. For instance, the ATLAS algorithm learns from its past decisions to improve its selection procedures and obtain new knowledge about when

Table 4.3: Resources Utilization of the Fair Scheduler

Job/Task	Scheduler	Fair				
		Basic	**ATLAS RF**	**ATLAS RF MDP**	**ATLAS RF MDP ϕ-FD**	**ATLAS RF MDP SFD**
	Resource	**Avg.**	**Avg.**	**Avg.**	**Avg.**	**Avg.**
Job	**CPU**	14251	13764	12783	11370	11150
	Memory	9458	9078	8766	8042	7647
	HDFS Read	10568	9478	8615	8339	8257
	HDFS Write	9943	8945	7453	7124	7066
Task	**CPU**	4730	4711	4672	4513	4313
	Memory	3007	3001	2955	2912	2496
	HDFS Read	1954	1934	1834	1809	1783
	HDFS Write	1963	1922	1893	1811	1776

Units: CPU (ms), Memory (10^5 bytes), HDFS Read (10^3 bytes), HDFS Write (10^3 bytes)

and where to apply the selected scheduling decisions. On the other hand, we noticed that ATLAS is associated with a cost in terms of execution time required to access the scheduling rules database and select the appropriate decisions to apply. This cost may have a negative impact on the time spent by ATLAS to select an appropriate action. For instance, the more scheduling rules ATLAS generates, the longer would be the time to decide the action to apply. Therefore, we decide to sort the scheduling decisions by the frequency of usage and the scheduling outcome (finished or failed). In other words, the most used scheduling decisions leading to success events will be on top of the scheduling decisions database of ATLAS. This approach was found to reduce the selection time in ATLAS. However, it penalizes some policies since they are not on top of the scheduling policies database.

To summarize the benefits of each of our proposed approaches in ATLAS, Table 4.5 presents the advantages of the integrated three algorithms presented in Sections 3.2.1, 3.3.1, and 3.4.1. Overall, we found that by early identifying tasks' failures ATLAS can reduce failures of tasks and jobs by up to 26%, and reduce

Table 4.4: Resources Utilization of the Capacity Scheduler

Job/Task	Scheduler	Capacity				
		Basic	**ATLAS RF**	**ATLAS RF MDP**	**ATLAS RF MDP ϕ-FD**	**ATLAS RF MDP SFD**
	Resource	**Avg.**	**Avg.**	**Avg.**	**Avg.**	**Avg.**
Job	**CPU**	16344	15974	15564	15324	14654
	Memory	11632	11025	10986	10845	9786
	HDFS Read	13032	12032	11396	10345	10012
	HDFS Write	11420	10452	9375	8452	8314
Task	**CPU**	5466	5398	5379	5212	4837
	Memory	3997	3904	3801	3775	3381
	HDFS Read	2074	2015	1943	1829	1688
	HDFS Write	2257	1978	1862	1749	1611

Units: CPU (ms), Memory (10^5 bytes), HDFS Read (10^3 bytes), HDFS Write (10^3 bytes)

the total execution time by up to 17%. Also, it could reduce the amount of used CPU and memory by up to 16% and 14%, respectively. By integrating adaptive scheduling policies to the scheduler, ATLAS can improve the performance of Hadoop. This is by reducing the number of failed tasks, the total execution time, the used CPU and memory by up to 19%, 14%, 9% and 8%, respectively. Based on the shared failure information, we affirm that ATLAS is able to catch more failures of TaskTrackers within Hadoop. As a result, it could reduce the number of failed tasks, the total execution time, the used CPU and memory by up to 14%, 8%, 7% and 5%, respectively.

4.2.3 Scalability Analysis Results

We analyze the scalability of ATLAS by executing a larger workload on the 1000-nodes Hadoop cluster. Concretely, we perform experiments to execute a different number of jobs: 30,000, 60,000, and 90,000 jobs composed of 750,000, 900,000, and 2,250,000

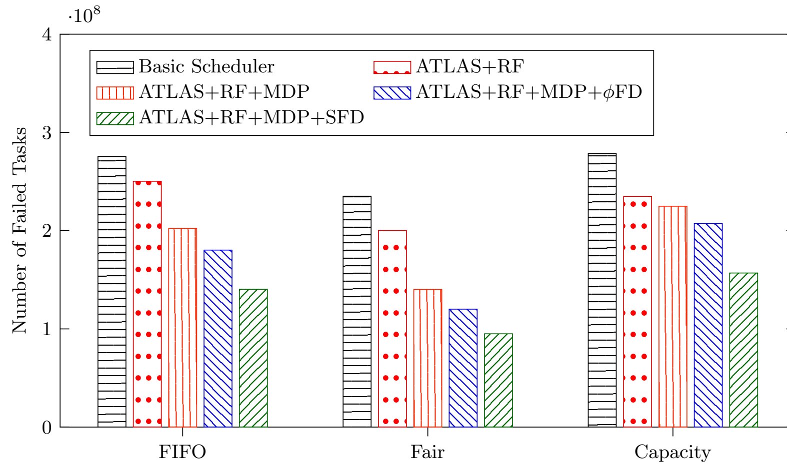

Figure 4.10: Number of Failed Task over 3 Days

Table 4.5: Benefits of ATLAS Components

	Failure Rate	**Execution Time**	**CPU Usage**	**Memory Usage**
Task Failure Prediction	26	17	16	14
Adaptive Scheduling Learning	19	14	9	8
TaskTracker Failure Detection	14	8	7	5

tasks, respectively. The tasks belonging to these executed jobs have different sizes; identified as: small/medium/large tasks. We evaluated the performance of ATLAS under different failures rates going from 5% to 40% by injecting different failures to the nodes and the scheduled tasks. This experiment was repeated 30 times to measure the performance of ATLAS.

Tables 4.6, 4.7, and 4.8 present the reduction rate results of ATLAS in terms of failures rates, execution time, and resources usage for the different workloads; 30,000, 60,000, and 90,000 jobs, respectively. Here, we discuss only the results of the Fair scheduler, because we observed that the three schedulers performances follow the same trend. In a larger cluster, ATLAS is able to identify up to 71% tasks' failures

and reschedule these tasks accordingly. Furthermore, it could reduce the amount of used CPU and memory by up to 53% and 48%, respectively. Hence, ATLAS is able to improve the overall utilization of resources in the deployed cluster.

Table 4.6: Reduction Rates (%) of Proposed Algorithms (30,000 Hadoop Jobs)

Number of Jobs	**30,000 Jobs (750,000 Tasks)**			
Reduction Rate (%)	**Tasks' Failures**	**Execution Time**	**CPU Usage**	**Memory Usage**
Algorithm 3.1	32	19	18	19
Algorithm 3.4	24	21	19	12
Algorithm 3.6	11	15	10	13

Table 4.7: Reduction Rates (%) of Proposed Algorithms (60,000 Hadoop Jobs)

Number of Jobs	**60,000 Jobs (900,000 Tasks)**			
Reduction Rate (%)	**Tasks' Failures**	**Execution Time**	**CPU Usage**	**Memory Usage**
Algorithm 3.1	36	21	19	16
Algorithm 3.4	21	19	18	13
Algorithm 3.6	17	17	14	10

Table 4.8: Reduction Rates (%) of Proposed Algorithms (90,000 Hadoop Jobs)

Number of Jobs	**90,000 Jobs (2,250,000 Tasks)**			
Reduction Rate (%)	**Tasks' Failures**	**Execution Time**	**CPU Usage**	**Memory Usage**
Algorithm 3.1	35	23	20	20
Algorithm 3.4	25	19	18	14
Algorithm 3.6	11	17	15	14

Next, we measure the added overhead when integrating ATLAS within the existing Hadoop schedulers for the created cluster. For this goal, we measure the Worst Case Execution Time (WCET) [93] of each of our proposed algorithms given the running workload. In general, we noticed that the size of the Hadoop cluster and the

number of scheduled jobs and tasks have a direct impact on the overhead of ATLAS. Tables 4.9, 4.10, and 4.11 presents the obtained results for the different algorithms integrated in ATLAS in terms of WCET using different workloads (*e.g.*, 30,000, 60,000, and 90,000 jobs). The WCET can reach up to 117, 258, and 183 seconds for Algorithms 3.1, 3.4, and 3.6, respectively. These results were expected due to the fact the scheduler requires more time to collect data from its environment about failures of tasks and TaskTrackers, the scheduler environment, received heartbeats from 1,000 nodes, etc., to select the appropriate scheduling strategies. Here, we should mention that although Algorithm 3.6 is characterized by a higher WCET, it is able to early catch up to 58% of the failures of TaskTrackers in the new created Hadoop cluster. Hence, it allows ATLAS to better assign tasks to alive nodes and avoid poor scheduling decisions leading to tasks' failures.

Table 4.9: Worst-Case Execution Time (Seconds) in ATLAS (30,000 Jobs)

Number of Jobs	**30,000 Jobs (750,000 Tasks)**		
Type of Task	**Small**	**Medium**	**Large**
Algorithm 3.1	27	63	98
Algorithm 3.4	45	105	143
Algorithm 3.6	211		

Table 4.10: Worst-Case Execution Time (Seconds) in ATLAS (60,000 Jobs)

Number of Jobs	**60,000 Jobs (900,000 Tasks)**		
Type of Task	**Small**	**Medium**	**Large**
Algorithm 3.1	29	74	103
Algorithm 3.4	53	127	159
Algorithm 3.6	234		

It is worth mentioning that all steps of Algorithm 3.6 are off the critical path of the scheduler. This is because they are used to collect data about the received

Table 4.11: Worst-Case Execution Time (Seconds) in ATLAS (90,000 Jobs)

Number of Jobs	**90,000 Jobs (2,250,000 Tasks)**		
Type of Task	**Small**	**Medium**	**Large**
Algorithm 3.1	35	96	117
Algorithm 3.4	71	148	183
Algorithm 3.6	258		

heartbeats and to adjust the expiry interval timeout accordingly. Hence, the integration of Algorithm 3.6 within Hadoop does not impact the execution time of the scheduled tasks; it only impacts the communication time between the JobTracker and TaskTrackers. For Algorithm 3.1, the steps from lines 2 to 11 are required to collect the log files and retrain/select the models, and hence do not generate an overhead to the scheduler. The only steps that are on the critical path of the scheduler are from lines 13 to 17 in Algorithm 3.1. For Algorithm 3.4, all steps are on the critical path of the scheduler.

In light of these results, we can conclude that ATLAS is very useful for very large clusters where there are enough failures to both learn from and also avoid. Indeed, the sizes of the cluster and running workload directly affect the performance of ATLAS. Despite the added overhead of ATLAS, we can affirm that ATLAS could improve the overall performance of the existing Hadoop schedulers. Indeed, the added overhead time by ATLAS was largely compensated by the time saved on the failed tasks that would have been executed otherwise.

4.3 Threats to Validity

In this section, we discuss the threats to validity of our experimental study we performed to analyze the performance of ATLAS, following the guidelines for case study [94].

4.3.1 Construct Validity

Construct validity threat analyzes the relation between the theory and observation. While building the task failure predictive algorithm, we considered that tasks characteristics have the major impact on the scheduling outcome of a task. However, this may not be the case because the amount of allocated resources for the scheduled tasks can also affect their outcomes. For instance, tasks receiving less resources, than their requested ones, have more chances to fail. While performing the correlation between the assigned resources and scheduling outcome of tasks, we found a low correlation between the used resources and the final status of the scheduled task. This means that the resource allocation is more likely to affect resource usages rather than scheduling outcomes. On the other hand, ATLAS is characterized by scheduling procedures that prioritize the processing of tasks on nodes having enough resources. This is done by collecting data from its environment about the available resources and under-loaded nodes. Consequently, this will allow the scheduler to avoid several failures and improve the resources utilization of the available resources.

4.3.2 Internal Validity

Internal validity threat analyzes the used techniques and tools to perform the experiments and evaluate the obtained results. For our methodology to track the failures of TaskTrakers, we adapted the four selected algorithms to adjust the communication between the JobTracker and TaskTracker nodes. Although this approach could early identify failures of nodes and notify ATLAS accordingly, one limitation is that when it shortens the communication interval to small values (*e.g.*, 2 minutes), the TaskTracker nodes can send too frequent messages, which may result in messages congestion on the JobTracker and may add extra time to read them.

Furthermore, we determine the amount of failures to be injected (up to 40%) using information about the tasks failures from Google clusters [92]. However, this failure rate can be high for the created Hadoop clusters (due to the difference between the cluster sizes between Hadoop and Google). For this reason, we perform other experiments to assess the performance of ATLAS under lower failures rates of 1-2%. Overall, we found that ATLAS could reduce the number of failed jobs and tasks by up to 9% and 12%, respectively. These finding confirm the fact that ATLAS highly depends on the training step, which means that the more data the scheduler collects about failures from its environment, the better the scheduling decisions would be and the better it learns from its past experience. Another important observation is that the injected failures scenarios may not represent realistic ones. Therefore, it is of interest to study the performance of ATLAS with a more diverse set of Hadoop clusters and different failure rates and types.

4.3.3 Conclusion Validity

Conclusion validity threat analyzes the relation between the used techniques and the obtained outcomes. The key idea behind ATLAS is to guide the existing Hadoop schedulers by providing better scheduling decisions with a minimal impact on the execution time. For this reason, while implementing the different components in ATLAS, we checked the overhead in terms of added time to overall execution times. Indeed, we have verified that the new integrated scheduling mechanisms do not introduce a large overhead. We performed this verification step by measuring the WCET of each added component to ATLAS separately for different scenarios (different workload and cluster sizes). Overall, the obtained results showed that the times spent to

train the model, to find the appropriate scheduling action, and to adjust the communication between the JobTracker and TaskTrackers did not negatively affect the overall execution times of scheduled tasks. This is due to the fact that ATLAS could save the execution times that would be spent to execute the failed tasks otherwise. We also check that ATLAS does not violate any property of Hadoop such as time-out expiration and task priority in the queue.

4.3.4 Reliability Validity

Reliability validity threat analyzes approaches to replicate our proposed work on other environments. Indeed, our proposed scheduling algorithm can be built within other cloud platforms like Microsoft Azure [95], or Google platform. In order to replicate ATLAS, one needs to collect data about previously executed tasks from these platforms to train and validate the machine learning algorithms. Next, these data and the built predictive algorithms can be used to adjust the proposed MDP-based model in ATLAS. Furthermore, our proposed approach to early detect failures can be adapted to other cloud platforms (*e.g.*, Spark [34], Storm [35], Microsoft Azure [95]) by collecting data about nodes failures in order to adjust the communication between the master and the nodes in such platform. Finally, we can mention that our proposed methodologies can be integrated either separately or combined on top of any cloud scheduler to reduce task failure rates and provide better resources utilization and execution time.

4.3.5 External Validity

External validity threat analyzes methods to generalize the results of our study. While evaluating the performance of our proposed scheduling algorithm, we created 100-nodes and 1000-nodes Hadoop clusters deployed on Amazon EMR. However, more studies can be done on larger scales using different types of machines (with different slot configurations) to validate the obtained results of our empirical study. In order to generalize the findings of ATLAS, we can implement the proposed methodologies on Spark [34], a novel in-memory computing framework for Hadoop. This is in order to evaluate their performance when used on Spark and compare it with those obtained from the Hadoop framework.

4.4 Summary

In this chapter, we presented ATLAS (AdapTive faiLure-Aware Scheduling) algorithm for Hadoop, to show the benefits of the proposed methodologies presented in Chapter 3. ATLAS is built using the proposed algorithms to early detect failed tasks and TaskTrackers and the adaptive algorithm to adjust the scheduling decisions of tasks on the fly. To assess the performance of the proposed algorithm, we performed an empirical study comparing ATLAS's performance with those of the three existing Hadoop schedulers (FIFO, Fair, and Capacity). The obtained results show that ATLAS achieves better results compared to the basic three common schedulers of Hadoop. More concretely, it can reduce the number of failed jobs by up to 49% and the number of failed tasks by up to 67%. By early identifying failed tasks and dynamically rescheduling them, ATLAS can reduce the total execution time of jobs by 9 minutes on average, and by up to 15 minutes for long running jobs; representing 35%

of the job execution times. ATLAS also reduced CPU and memory usages by 25% and 24%, respectively. However, given the complexity and the wide range of constraints in Hadoop, our failures detection and adaptive scheduling approaches were not able to provide an exhaustive coverage of the Hadoop system functionalities and to ascertain a complete analysis of its scheduler. Therefore, we present in the next chapter a methodology to formally analyze Hadoop schedulers to enable the early identification of circumstances leading to failures and hence reduce the failures' rates in ATLAS.

Chapter 5

Formal Verification of Hadoop

In this chapter, we propose a new methodology to formally analyze Hadoop schedulers and identify the impact of the scheduling decisions of Hadoop on the failures rates of tasks. Our methodology can early identify circumstances leading to potential failures and hence it can provide possible strategies to avoid their occurrences in ATLAS. Towards this goal, we propose to verify some of the most important scheduling properties in Hadoop including the schedulability, resources-deadlock freeness, and fairness [96]. Firstly, we construct a formal model of the Hadoop scheduler using CSP language. Next, we analyze the three mentioned properties within the Hadoop schedulers using the PAT model checker. We investigate the correlation between the adopted scheduling strategies and the failures rate. Finally, we apply our proposed methodology on the scheduler of OpenCloud, a Hadoop-based cluster [97] in order to illustrate its usability and benefits. Then, we evaluate the benefits of the proposed methodology on ATLAS given the provided scheduling strategies.

5.1 Preliminaries

In this section we briefly present some basic concepts of the CSP language and the PAT tool, to better understand the different steps of our proposed methodology.

CSP is a formal language used to model and analyze the behavior of processes in concurrent systems. It has been practically applied in modeling several real-time systems and protocols [31]. In the sequel, we present a subset of the CSP language, which will be used in the latter content, where P and Q are processes, a is an event, c is a channel, and e and x are values:

`P , Q ::= Stop | Skip | a → P | P ; Q | P || Q | c!e → P | c?x → P`

- `Stop`: indicates that a process is in the state of deadlock.
- `Skip`: indicates successfully terminated process.
- `a → P`: means that an object first engages in the event a and then behaves exactly as described by P.
- `P ; Q`: denotes that P and Q are sequentially executed.
- `P || Q`: denotes that P and Q are processed in parallel. The two processes are synchronized with the same communication events.
- `c!e → P`: indicates that a value e was sent through channel c and then process P .
- `c?x → P`: indicates that a value was received through channel c and stored in a variable x and then process P .

PAT [32] is a CSP-based tool used to simulate and verify concurrent, real-time systems, etc. It implements different model checking techniques for system analysis and

properties verification (*e.g.*, deadlock-freeness, reachability) in distributed systems. Different advanced optimizations techniques in PAT, such as partial order reduction, symmetry reduction, etc., have been proposed to achieve better performance in terms of the number of explored states and time.

5.2 Formal Verification Methodology

In this section, we present an overview of our methodology to early identify circumstances leading to potential tasks' failures, followed by a description of each step.

5.2.1 General Overview

Figure 5.1 presents the key idea of our methodology to formally verify the Hadoop scheduler using model checking techniques. Our proposed methodology takes as inputs (1) the description of the Hadoop scheduler, (2) the specification of the properties to be verified, and (3) the cluster configuration (*e.g.*, type of scheduler, number of nodes, workload and failure distributions, schedulability rate). The output of our methodology a set of possible scheduling strategies to avoid these failures.

The first step in our methodology is the construction of a formal model for the Hadoop scheduler and its properties. To this aim, we use the CSP language to formally model the scheduler because it enables the modeling of synchronous and concurrent systems. Specifically, it enables to model the behavior and communication of multiple processes and parallel components for different distributed systems [31]. We also use the Linear Temporal Logic (LTL) to provide a description of the properties we aim to verify.

The next step in our methodology is to identify a potential failures rate that a Hadoop cluster may experience. We use the PAT model checker to perform the formal

quantitative analysis of failures in Hadoop scheduler. Our choice of PAT is motivated by the fact that PAT is based on CSP and it showed good results to simulate and verify concurrent, real-time systems, etc. [98].

Finally, we propose to use the generated verification traces to perform qualitative analysis of these failures. Given the generated results from PAT, we propose to determine the circumstances and specifications leading to tasks' failures in the scheduler. Consequently, our proposed methodology can provide scheduling strategies to reduce the number of failed tasks, avoid poor scheduling decisions, and improve the overall cluster performance (resources utilization, total completion time, etc.). The remainder of this section elaborates more on each of the steps of our methodology.

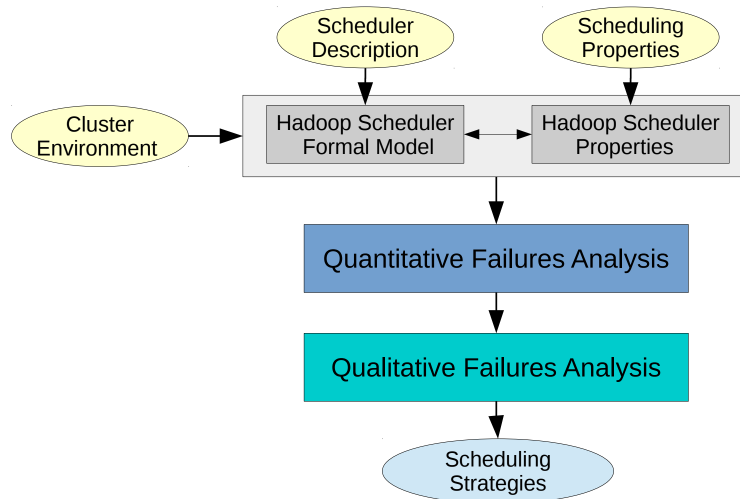

Figure 5.1: Formal Analysis of Hadoop Schedulers Methodology

5.2.2 Hadoop Scheduler Formal Model

The goal of this step is to build a formal model using the CSP language to describe the main components responsible for the scheduling of tasks in Hadoop. Concretely, we construct a model to formally describe the Hadoop master node (including the

JobTracker and NameNode) and the Hadoop slave/worker node (including the TaskTracker and DataNode). At the master level, we model the scheduler and the components responsible for task assignment and resources allocation in Hadoop. At the worker level, we model the components responsible for the task execution in the TaskTracker and the DataNode. Data locality, data placement, and speculative execution are among the most important scheduling constraints due to their direct impact on the scheduling strategies and the performance of executed tasks [17]. Therefore, we decided to integrate these three scheduling constraints within the proposed model of Hadoop scheduler.

In the sequel, we describe the required steps to create a Hadoop cluster following our propsoed model. Thereafter, we present examples of implemented CSP processes[1] to illustrate our formal description of the Hadoop scheduler, TaskTracker activation and task assignment, where "Cluster()" is the main process:

```
Cluster() = initialize(); NameNode_activate() || JobTracker_activate() ||
            (|| i:{0..(N-1)}@DataNode_activate(i)) ||
            (|| i:{0..(N-1)}@TaskTracker_activate(i))||
            Hadoop_Scheduler();
```

The *"Hadoop_Scheduler()"* process represents our proposed formal description of the steps followed by the scheduler to verify the constraints to schedule a map or reduce task. For instance, it checks the availability of the resources in the cluster (*slotTT[i]>0*). Next, it checks the type of received task, either a map (*Queue[index] == 1*) or reduce (*Queue[index] == 2*) task. After, it verifies whether the received task should be speculatively executed or not (*e.g.*, map:

[1]The CSP script is available at: `http://hvg.ece.concordia.ca/projects/cloud/fvhs.html`

Queue[index] == 3 or reduce: *Queue[index] == 4*).

```
Hadoop_Scheduler() = {
   if( (slotTT[i]>0) )
   { while( (found==0) && (index < maxqueue ) ){
     if (Queue[index] == 1)  //it is a map task
      {  schedulable = 1; found =1; location = index;
         type = MapTask; IDjob_ task = IDJob[index];}
     if((Queue[index] == 2))  //it is a reduce task
      { if(FinishedMap[IDJob[index]] == Map[IDJob[index]]  )
       { schedulable = 1; found =1; location = index;
         type = ReduceTask; IDjob_ task = IDJob[index];}    }
      if(Queue[index] == 3)  //it is a speculated map task
       { schedulable = 1; found =1; location = index; type = MapTask;
         IDjob_ task = IDJob[index]; SpeculateTask[location] = 1;}
      if(Queue[index] == 4)  //it is a speculated reduce task
      { schedulable = 1; found =1; location = index; type = ReduceTask;
         IDjob_ task = IDJob[index]; SpeculateTask[location] = 1;}
        ...
 }}} →  signedtask?i→  signedtask_i→Task_Assignment(location, type);
```

The *"TaskTracker_activate(i)"* process represents our formal description to activate a TaskTracker in a Hadoop cluster. To do so, it checks first whether the JobTracker was already activated ($JobTracker == ON$). Thereafter, it activates the input number of slots for each TaskTracker (*slotsnb*). As a result, the activated TaskTracker is ready to execute up to *slotsnb* tasks.

```
TaskTracker_activate(i) = activate_jt_success  → ifa(TaskTracker[i] ==
                      OFF && JobTracker == ON) {activate_tt.i{
                      TaskTracker[i] = ON; trackercount++;}
                       → atomic{activate_tt_success.i →
                      (|| j:{1..(slotsnb)}@TaskTracker_sendready(i))}};
```

The following *"TaskTracker_execute(i)"* process gives an example of task execution procedure while checking its data locality. First, it checks the availability of the slot assigned to a given task by the scheduler (if slot is free then *task_running[nbTT][k]* is equal to 0, where $nbTT$ is the ID of the TaskTracker and k is the ID of the assigned slot). Second, it checks whether the input data of the task to be executed is local or not. This is done by checking whether the node where to execute the task ($selectedTT$) is the same node where its data is located (*Data-LocalTT[idtask]*).

```
TaskTracker_execute(i) = {
 var nbTT =i; var found = 0; var k = 0;
 while((k<slotsnb) && (found == 0) ){
  if(task_ at_ tasktracker[nbTT][k]==1 && task_running[nbTT][k]== 0)
   {selectedslot =k; found = 1; }
  k++; } ...
  if(Data-LocalTT[idtask] == selectedTT) //check locality of the task
   {locality = locality + 1;  Locality[idtask] = 1;}
    else {nonlocality = nonlocality + 1;  Locality[idtask] = 0;   }
 ...}
```

```
}→ if(pos== -1) {TaskTracker_ execute(i)}
            else {execute(i,selectedslot)};
```

5.2.3 Hadoop Scheduler Properties

Given the proposed formal model of the Hadoop scheduler, we aim to verify three properties that are the *schedulability*, *fairness* and *resources-deadlock* freeness. Cheng *et al.* [96] claim that these properties represent some of the main critical properties that can affect the performance of a scheduler in distributed systems. In addition, they explain their impact on the scheduling decisions in terms of task outcomes, delays, resources utilization.

The schedulability checks whether a task is scheduled and satisfies its expected deadline when scheduled according to a scheduling algorithm. The fairness checks whether the submitted tasks are served (*e.g.*, receiving resources slots that ensure their processing) and there are no tasks that will never be served or will wait in the queue more than expected. The resources-deadlock checks whether two tasks are competing for the same resource or each is waiting for the other to be finished. To better explain these three properties, we present in the following how each property is defined.

For the schedulability, the life cycle of a task can be presented as a set of actions to go from state: *submitted* to *scheduled* to *processed* then to *finished-within-deadline* or *finished-after-deadline* or *failed*. Let X be the total number of scheduled tasks and Y be the number of tasks finished within their expected deadlines. The schedulability rate can be defined as the ratio of X over Y. For instance, the property *"schedulabilityrate > 80"* means checking whether the scheduler can have a total of

80% of tasks finished within their deadlines.

The fairness property is used to evaluate how fair is a Hadoop scheduler to distribute tasks and assign them resources to be finished. This property evaluates how long a submitted task can be waiting or running in the scheduler. In other terms, tasks go from state *submitted*, *waiting*, *scheduled* to *finished/failed*. We consider a variable W as the percentage of tasks that have been served on time by the scheduler over the total number of scheduled tasks. Here the fairness property can be expressed for example as: *"fairnessrate = 90"*. This means checking the probability that the scheduler will eventually have 90% of tasks served on time.

A job or a task can be characterized by deadlock-occurrence when, for example, task *t1* holds resource *r1* and task *t2* holds resources *r2*. The resources *r1* and *r2* are locked by these two tasks. Task *t2* requires *r1* to be finished and task *t2* requires *r1* to be finished as well. As a result, both tasks experience a resource-deadlock and can be killed by the scheduler. We consider a variable Z as the percentage of tasks that experience deadlock-occurrence while being processed over the total number of scheduled tasks. In other terms, these tasks go from state *submitted*, *scheduled* to *waiting-resources*. Here the property can be expressed for example as: *"resourcedeadlockrate = 60"]*. This means checking that the scheduler will eventually have 60% of tasks characterized by deadlock occurrence.

To verify above properties in PAT, we need to provide their description in LTL. For example, the following LTL formulas check the schedulability and resource-deadlock freeness of given tasks. The first example checks whether a given task eventually goes from the state submitted to the state finished within the deadline. The second example checks whether a given task should not go to a state of waiting-resources. Here, $\Diamond$, $\models$, and $\neg$ represent *eventually*, *satisfy*, and *not*,

respectively, in the LTL logic.

```
#assert ◊ (task ⊨ (submitted -> finished-within-deadline) );
#assert ¬ (task ⊨ (submitted -> waiting-resources) );
```

5.2.4 Quantitative Failures Analysis

Given the CSP model of the Hadoop scheduler and the LTL description of the three properties, we use the PAT model checker to perform the formal analysis of the scheduler performance. This is by verifying the three selected properties. Specifically, we can vary the rates of these properties to evaluate their impact of Hadoop performance and the failures rates that may encounter the created Hadoop cluster. To better illustrate this step, we provide here some examples of properties verification using PAT. For instance, *"goal0"* represents a goal to check whether all submitted tasks (*"workload"*) are successfully scheduled. The verification of this goal using PAT can identify if the modeled Hadoop cluster, *"cluster1"*, can reach this goal or not. Another example could be to check if *"cluster1"* meets *"goal0"* with a *"schedulabilityrate"* of 80%. The following examples present some of the properties that can be verified using our approach.

```
#define goal0 completedscheduled  == workload && workload >0;
#assert cluster1 reaches goal0;
#assert cluster1 reaches goal0  && schedulabilityrate >80;
#define goal1 fairnessrate ==50;
#assert cluster1 reaches goal0  && goal1;
#define goal2 resourcedeadlockrate ==50;
#assert cluster1 reaches goal0  && goal1  && goal2;
```

5.2.5 Qualitative Failures Analysis

Given the generated traces by the PAT model checker, we propose in the last step of our methodology to parse these traces and extract data about possible relations between the scheduling strategies and the failures rates. To this aim, we propose to check the states where the scheduler does (not) meet a given property and correlate these states to the obtained scheduler performance and to the input cluster configurations. As a result, we can investigate the correlation between the cluster settings, the applied scheduling strategies, and the failures rate. Based on the obtained correlation results, our proposed methodology can provide/suggest possible scheduling strategies to overcome these failures. For instance, it can help Hadoop developers by providing recommendations to change the scheduling decisions (*e.g.*, delay long tasks, wait for a local task execution). Furthermore, it can guide Hadoop customers by suggesting guidelines to change and adjust their cluster configurations (*e.g.*, number of nodes, number of allowed speculative executions).

5.3 Formal Verification Evaluation

Given the different steps of our proposed methodology, we formally analyze the scheduler of OpenCloud, a Hadoop-based cluster [97].

5.3.1 Experimental Design

Before performing the analysis, we investigated Hadoop schedulers available in the open literature to select a case study to evaluate our proposed approach. The main public case studies are Google [99], Facebook [100], and OpenCloud [97] traces that provide most of the inputs required by our methodology. For Google traces, we

found out that they do not contain information about the cluster settings, which are among the important factors affecting the verification results. When checking the Facebook traces, we noticed that there are not enough data describing the cluster settings, capacity of nodes, failures rate, etc., which are essential for our approach. The OpenCloud traces contain enough information that match the required inputs of our verification approach (*e.g.*, # nodes, capacity of nodes, workload). Furthermore, they provide public traces of real-executions of Hadoop workloads over a period of more than 20 months. Consequently, we select the OpenCloud traces to formally verify the scheduler of this Hadoop cluster. OpenCloud is an open cluster for research purposes used in different areas including machine learning, astrophysics, biology, cloud computing, etc. It is managed by Carnegie Mellon University.

We start the verification of the scheduler of OpenCloud cluster by extracting from the traces the required input information for our proposed methodology. To this aim, we analyze the description of the workload given in the first month trace, the second month trace, and the first six months traces. This step is required in order to evaluate the performance of our proposed methodology in terms of the number of visited states and execution time, using different input traces. While analyzing the traces, we observed that they do not provide any information about the type of scheduler used in the cluster. Therefore, we decided to verify the properties of the modeled cluster for the three existing schedulers (FIFO, Fair and Capacity). This is due to the direct impact of the used scheduler on the performance of the cluster. We performed the verification of the OpenCloud scheduler considering different property requirements to evaluate their impact on the failures rate. In the experiments we conduct in PAT, we use the search engine "First Witness Trace with Zone Abstraction" [32] for the analysis with symmetry reduction. The workstation used to perform

these experiments is an Intel i7-6700HQ (2.60GHz*8) CPU with 16 GB of RAM.

5.3.2 Experimental Results

In the following, we discuss the results of our proposed methodology when applied to the OpenCloud scheduler. Precisely, we present the obtained scalability results along with results of the quantitative and qualitative analyses steps in our methodology.

Properties Verification and Scalability Analysis

We present the obtained verification results of our approach when applied to the traces of the first month in Table 5.1. Indeed, these trace files provide a description about 1,772,144 scheduled tasks. According to the obtained results, we found out that the Fair scheduler meets the two schedulability rates of 50% and 80%. Therefore, we can claim that up to 80% of tasks are scheduled and finished within their deadlines. For the Capacity scheduler, we observed that it does not satisfy the schedulability property for a rate of 80%. However, the FIFO scheduler does not meet the two schedulability rates of 50% and 80%, which means that up to 50% of the scheduled tasks are exceeding their deadlines when being executed. At this level, we should mention that tasks that finished after their expected deadlines can use their assigned resources more than expected. Hence, they affect the overall performance of the Hadoop cluster.

When analyzing the results for the fairness property, we noticed that the Fair scheduler meets the two fairness rates of 50% and 80%. This means that up to 80% of submitted tasks get served and scheduled on time. Whereas, the FIFO and Capacity schedulers do not satisfy the fairness property for the two give values. This means that more than 80% of the submitted tasks, from the trace files of the first month,

Table 5.1: Verification Results: Trace for the First Month (1,772,144 Tasks)

Property	Scheduler	Valid?	#States	Time(s)
Schedulability = 50%	**FIFO**	No	742 K	1648
	Fair	Yes	742 K	1597
	Capacity	Yes	742 K	1604
Schedulability = 80%	**FIFO**	No	742 K	1650
	Fair	Yes	742 K	1614
	Capacity	No	742 K	1602
Fairness = 50%	**FIFO**	No	742 K	1594
	Fair	Yes	742 K	1615
	Capacity	No	742 K	1612
Fairness = 80%	**FIFO**	No	742 K	1675
	Fair	Yes	742 K	1642
	Capacity	No	742 K	1619
Resources-Deadlock = 10%	**FIFO**	Yes	742 K	1602
	Fair	No	742 K	1610
	Capacity	Yes	742 K	1632
Resources-Deadlock = 30%	**FIFO**	No	742 K	1596
	Fair	No	742 K	1618
	Capacity	Yes	742 K	1623

are characterized by a longer waiting time in the queue before getting served. Hence, this may lead to the task starvation problem and hence, decreasing the overall cluster performance.

For the resource-deadlock property, the Capacity scheduler satisfies the two rates of 10% and 30%, which means that at least 30% of tasks experienced a resources-deadlock issue when being executed. However, the FIFO and the Fair schedulers are characterized by smaller numbers of tasks facing resources-deadlock as presented in Table 5.1. The FIFO scheduler shows that only 10% of the scheduled tasks experience the problem of resources-deadlock. While, the Fair scheduler shows that less than 10% of tasks may experience an issue of resources-deadlock. This is because it does not meet the two resources-deadlock rates of 10% and 30%. For the Capacity scheduler, the problem of miscalculation of resources (headroom calculation) [101] can be one of the main reasons behind these results. Indeed, the Capacity scheduler assumes that

there are enough slots to reschedule the failed maps but, this may not be the case because it assigns these resources to the reduce tasks. In this case, the map tasks will be waiting for the reduce tasks to release their resources and the reduce tasks will wait for the expected outputs from these map tasks.

In summary, we can affirm that our proposed methodology allows us to verify the three properties for each scheduler. The results of our methodology were obtained by exploring on average 742 K states in 1619 seconds, as shown in Table 5.1.

When applied to the second traces of the OpenCloud cluster, our proposed methodology could formally analyze the three properties for the three schedulers using raw data of 476,034 tasks. We observe that the FIFO, Fair and Capacity schedulers achieve a schedulability rate of 30% and a fairness rate of 20%, whereas they all do not meet a rate of 90% of schedulability and fairness, as shown in Table 5.2.

Table 5.2: Verification Results: Trace for the Second Month (476,034 Tasks)

Property	Scheduler	Valid?	#States	Time(s)
Schedulability = 30%	**FIFO**	Yes	581 K	1198
	Fair	Yes	581 K	1136
	Capacity	Yes	581 K	1205
Schedulability = 90%	**FIFO**	No	581 K	1149
	Fair	No	581 K	1116
	Capacity	No	581 K	1183
Fairness = 20%	**FIFO**	Yes	581 K	1239
	Fair	Yes	581 K	1167
	Capacity	Yes	581 K	1193
Fairness = 90%	**FIFO**	No	581 K	1188
	Fair	No	581 K	1104
	Capacity	No	581 K	1246
Resources-Deadlock = 35%	**FIFO**	No	581 K	1135
	Fair	No	581 K	1106
	Capacity	Yes	581 K	1219
Resources-Deadlock = 50%	**FIFO**	No	581 K	1246
	Fair	No	581 K	1159
	Capacity	No	581 K	1217

For the resources-deadlock, we notice that at least 35% of tasks can experience resource-deadlock when scheduled with the Capacity scheduler. In a comparison with the first trace results, our approach explores 581 K states in (up to) 1246 seconds. This is expected due to the huge difference in the size between the two input traces.

To assess the scalability of our methodology, we conduct the formal analysis of a cumulative workload from the Opencloud traces. To this aim, we start by analyzing the traces of the first month and we incrementally add the workload of each month to the cumulated trace to be analyzed. Table 5.3 summarizes the analysis results of the first six months traces together. Indeed, these traces provide a description about 4,006,512 scheduled tasks.

Table 5.3: Verification Results: Trace for the 1-6 Months (4,006,512 Tasks)

Property	**Scheduler**	**Valid?**	**#States**	**Time(s)**
Schedulability = 50%	**FIFO**	Yes	17692K	4350
	Fair	Yes	17692K	4362
	Capacity	Yes	17692K	4359
Schedulability = 90%	**FIFO**	No	17692K	4346
	Fair	No	17692K	4341
	Capacity	No	17692K	4367
Fairness = 30%	**FIFO**	Yes	17692K	4377
	Fair	Yes	17692K	4312
	Capacity	Yes	17692K	4335
Fairness = 90%	**FIFO**	No	17692K	4352
	Fair	No	17692K	4328
	Capacity	No	17692K	4369
Resources-Deadlock = 10%	**FIFO**	Yes	17692K	4322
	Fair	No	17692K	4360
	Capacity	Yes	17692K	4354
Resources-Deadlock = 50%	**FIFO**	No	17692K	4338
	Fair	No	17692K	4342
	Capacity	No	17692K	4328

Quantitative Failures Analysis

Based on the verification results given by the PAT model checker, we use the generated traces to identify the states where tasks failed. Also, we compare the scheduling outcomes of tasks from the PAT traces to those of the executed tasks in the real OpenCloud cluster. Furthermore, we check the verified properties for these failed tasks. This step is required to examine possible connections between the scheduling decisions and the failed tasks. To this aim, we apply this step on the trace of the first month since it provides a large number of scheduled tasks (*i.e.*, 1,772,144 tasks). First, we check the scheduling outcomes of tasks from both PAT and cluster traces and classify the observations into four main categories: *True Positive (TP)*, *True Negative (TN)*, *False Positive (FP)*, and *False Negative (FN)*. *TP* denotes the amount of finished tasks according to our approach traces and characterized by success outcomes using the simulation traces. *TN* is the number of failed tasks according to both the traces from our approach and the simulation ones. *FP* represents the failed tasks, based on the simulation traces, that are identified as finished tasks using our methodology. While *FN* denotes the number of identified failed tasks that were finished during the real simulation.

Our proposed methodology could determine up to 56.82% of the finished tasks (*TP*, Fair, schedulability = 80%), and up to 4.64% of the failed tasks (*TN*, Fair, fairness = 80%) as presented in Table 5.4. Overall, we noticed that the *TN* values are small compared to the *TP* ones because the analyzed trace of the first month include more than 94% of successful tasks.

Thereafter, we calculate the *Detected Failures (DF)* that can be defined as the number of detected failures using our methodology among the total number of failed tasks in the given trace files. This step is fundamental in order to quantify the number

Table 5.4: Coverage Results(%): Trace for the First Month (1,772,144 Tasks)

Property	Scheduler	TP	TN	FP	FN	DF
Schedulability = 50%	**FIFO**	47.29	2.47	3.41	46.83	42.00
	Fair	55.38	3.82	2.06	38.74	64.96
	Capacity	46.09	2.85	3.03	48.03	48.46
Schedulability = 80%	**FIFO**	47.98	2.79	3.09	46.14	47.44
	Fair	56.82	4.21	1.67	37.3	71.59
	Capacity	42.18	3.03	2.85	51.94	51.53
Fairness = 50%	**FIFO**	47.84	2.92	2.96	46.28	49.65
	Fair	49.63	3.86	2.02	44.49	65.64
	Capacity	44.11	3.04	2.84	50.01	51.70
Fairness = 80%	**FIFO**	49.77	3.32	2.56	44.35	56.46
	Fair	50.14	4.64	1.24	43.98	78.91
	Capacity	48.29	3.48	2.4	45.83	76.18
Resources-Deadlock = 10%	**FIFO**	46.21	3.19	2.69	47.92	54.25
	Fair	51.44	3.63	2.25	42.68	61.73
	Capacity	43.77	2.84	3.04	50.35	48.29
Resources-Deadlock = 30%	**FIFO**	48.54	2.91	2.97	45.58	49.48
	Fair	55.26	3.75	2.13	38.86	63.77
	Capacity	49.71	3.19	2.69	44.41	54.25

of failures that our methodology could identify when compared to the real-execution simulation. Hence, we can answer the question of how many failures could be identified by our formal verification approach before the application is deployed in a Hadoop cluster.

In this context, we observe that our methodology is able to early detect between 42% (*DF*, FIFO, schedulability = 50%) and 78.91% (*DF*, Fair, fairness = 80%) of the failures. Furthermore, results show that our proposed methodology is characterized by a higher rate of *FN*, which is in the order of 40%. In other words, our approach indicates that more than 40% of tasks are failed however, they are finished in the simulation traces. We can explain this result by the internal recovery mechanisms used in Hadoop to reschedule and recover these tasks in the event of failures. For instance, Hadoop is characterized by a pausing and resuming strategy to allow tasks with higher priorities to be executed with enough resources without killing tasks with

lower priority [16]. Although these internal mechanisms could reduce the number of failed tasks, they can affect the total completion times and resources utilization of the scheduled tasks.

Qualitative Failures Analysis

After determining the potential tasks failures rate that a Hadoop cluster can experience, we checked the circumstances and specifications in the scheduler that may lead to these failures. To do so, we check the states of the failed tasks in the generated PAT traces and analyze the factors (*e.g.*, scheduling decisions, cluster configurations) leading to the failures of these tasks. For instance, we observed that there exist multiple failures, up to 32%, because of tasks waiting or executed for a long time that exceeds the property *"mapred.task.timeout"*[2] defining the maximum timeout for a task to be finished. When checking the states of these tasks, we found out that these long delays are caused mainly by the delay to copy input data from another node to the node where the task is executed; *data locality constraints* [17]. Delays caused by this constraint can reach 10 minutes (for small and medium tasks). Furthermore, we noticed that several jobs are failed because of *struggling tasks* (40% of scheduled tasks). struggling tasks are characterized by long execution times and resources contention for a longer time more than expected hence, they can decrease the overall performance of a Hadoop cluster.

Another important observation is that several tasks failures were cascaded from one job to another, especially for the long Hadoop chains (*e.g.*, composed of more than 2 jobs). We can explain these cascaded events between the jobs because of the unawareness of these failures and the lack of information sharing about these failures in

[2]This property is taken from "https://hadoop.apache.org/docs/r2.7.2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml"

Hadoop. When tasks exceed the property *"mapred.map.tasks.speculative.execution"*[2] defining the maximum number of allowed speculative execution, they have likely more chances to fail. Precisely, we found out that 26% of the tasks failed because they exceeded this maximum number. On the other hand, we checked the states of tasks characterized by long waiting times in the queue (Fairness property). We noticed that these delays can be caused by the scheduling of long-execution tasks that occupy their assigned resources for a long time (*e.g.*, large input file to be processed or a job composed of more than 1000 tasks).

In light of these observations, we can conclude that one can adjust the scheduler settings or the cluster configurations of Hadoop to avoid poor scheduling decisions and avoid tasks failures, by early knowing these circumstances and factors. For instance, a Hadoop developer can change the scheduler design to allow more time for tasks before being terminated; this is by changing the value of the timeout of scheduled tasks. One possible strategy could be also to change the type of the adopted scheduling principles in the cluster (*e.g.*, FIFO, Fair, Capacity, etc.). Another strategy can be to change the configuration of the created cluster by changing the number of available nodes to add more resources. Indeed, the added new resources can give other chances for the tasks that did not get enough resources. Hence, they can solve the fairness and resources-deadlock issues.

To better illustrate the benefits of our proposed approach, we evaluate the performance of the modeled OpenCloud cluster using a different cluster setting. This is in order to show the impact of an integrated scheduling strategy or a guideline to adapt the scheduling decisions and to reduce the failures rate. To do so, we adjust the cluster settings of the modeled OpenCloud cluster by changing the number of nodes from 64 to 100. We evaluate the impact of adding more resources in the cluster

considering a failures rate of 5.88% (the identified failure rate from the first trace file). The obtained results of the cluster in terms of failures rates for the three schedulers are presented in Figure 5.2. Given an OpenCloud cluster composed of 100-nodes, the number of failed tasks is reduced by up to 2.34% (Fair scheduler), which represents a total reduction rate of 39.79%. These results were expected because our verification methodology showed that several failures occur because of tasks struggling for more than 10 minutes; waiting for other resources to be released.

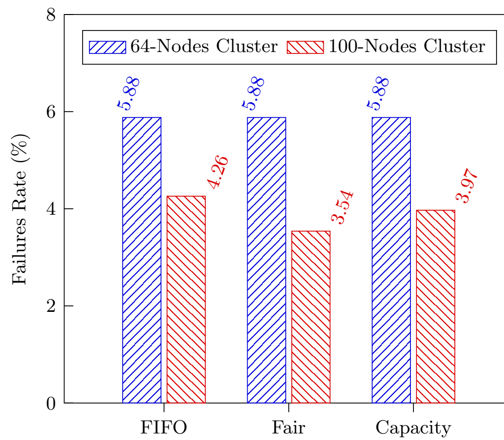

Figure 5.2: Impact of Adding Resources on Failures Rate

In light of these results, we affirm that the feasibility study, which we performed on the OpenCloud cluster (Section 5.3), allows us to evaluate the benefits of our approach to early identify circumstances leading to failures and to propose strategies to overcome them. Therefore, we will apply our approach to formally analyze the Hadoop schedulers of the created cluster for ATLAS (Section 4.2.1) in order to check whether it can identify tasks failures before their occurrences. Next, we will evaluate the impact of scheduling strategies on ATLAS performance in terms of the number of failed jobs and tasks.

5.4 Formal Verification of Hadoop and Refinement of ATLAS

In this section, we present our approach to formally verify Hadoop schedulers of the created cluster for ATLAS (Section 4.2.1) and the obtained results when integrating the scheduling strategies within ATLAS in terms of the number of failures' rates.

5.4.1 Experimental Design

Given the formal analysis methodology described in Section 5.2, we can formally analyze tasks failures in Hadoop scheduler (*e.g.*, FIFO, Fair, Capacity) to early identify circumstances and specifications leading to potential failures and prevent their occurrences. To this aim, we use the description of the created 100-nodes Hadoop cluster (*e.g.*, number of nodes, capacity of nodes), description of the executed workloads (obtained from the cluster log files), and the specification of properties to be verified as inputs for our formal analysis approach as presented in Section 4.2.1. We performed the verification of Hadoop schedulers considering different property requirements to evaluate their impact on the failures rate. The workstation used to perform these experiments is the same described in Section 5.3.1.

Upon the identification of possible tasks failures, we parse the traces provided by the PAT model checker to identify potential scheduling strategies for ATLAS. Based on the obtained scheduling strategies, we can either adjust the cluster and scheduler design before instantiating the Hadoop cluster or adjust and refine the existing scheduling strategies. Next, we measure the new failures' rates, in terms of the number of failed jobs, map and reduce tasks, in ATLAS when integrating the scheduling strategies.

5.4.2 Experimental Results

From the performed analysis of Hadoop schedulers in the 100-nodes cluster, we found out that different tasks experienced several failures because of the values of maximum number of allowed speculative execution and maximum timeout for a task before being killed. Therefore, we adjust these two parameters values in the scheduler and evaluate their impacts on the failures rates. In order to have a fair comparison when integrating the new strategies, we keep the same cluster, scheduler designs and the same workload description. We only adjust the used scheduling strategies for the ATLAS+RF+MDP+SFD scheduling algorithm to obtain an "ATLAS+RF+MDP+SFD-Refined" scheduling algorithm. Overall, we use, for the performed experiments, the implementations of (1) the basic Hadoop scheduling, (2) ATLAS+RF+MDP+SFD scheduling, and (3) ATLAS+RF+MDP+SFD-Refined scheduling algorithms when built with FIFO, Fair, and Capacity.

Figures 5.3, 5.4, and 5.5 present the obtained results of the basic Hadoop, ATLAS+RF+MDP+SFD, and ATLAS+RF+MDP+SFD-Refined scheduling algorithms in terms of the number of failed jobs, map tasks, and reduce tasks, respectively. When integrating the generated scheduling strategies from the formal analysis approach, we found out that the ATLAS+RF+MDP+SFD-Refined algorithm achieves better performance than the ATLAS+RF+MDP+SFD and the basic Hadoop scheduling implementations.

Overall, we noticed that ATLAS+RF+MDP+SFD-Refined algorithm could reduce the number of failed jobs and tasks by up to 7% and 12%, respectively, when compared to the ATLAS+RF+MDP+SFD algorithm. As a result, it could reduce the execution times by up to 9% and hence the CPU and memory usage are used by up to 4% and 3%, respectively. Consequently, we can claim that the two selected

scheduling strategies identified from the formal analysis methodology could improve the performance of ATLAS.

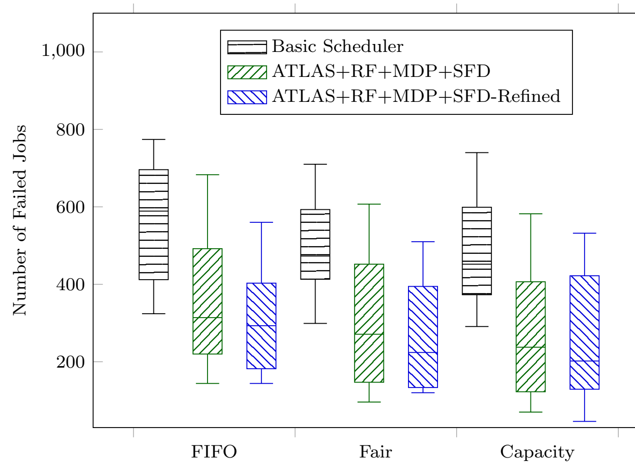

Figure 5.3: Impact of Verification Guidelines on Failed Hadoop Jobs

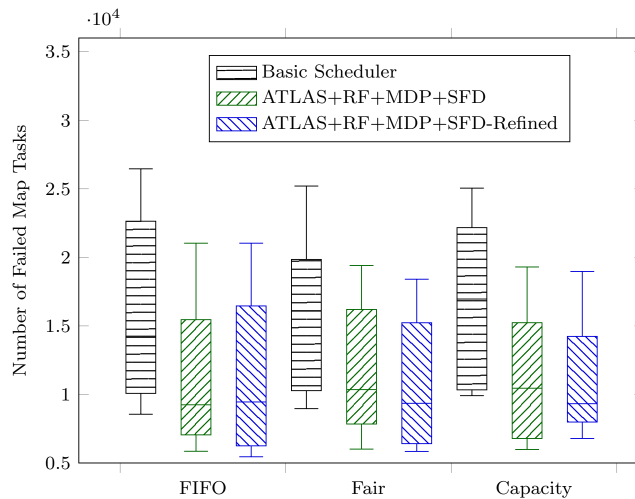

Figure 5.4: Impact of Verification Guidelines on Failed Map Tasks

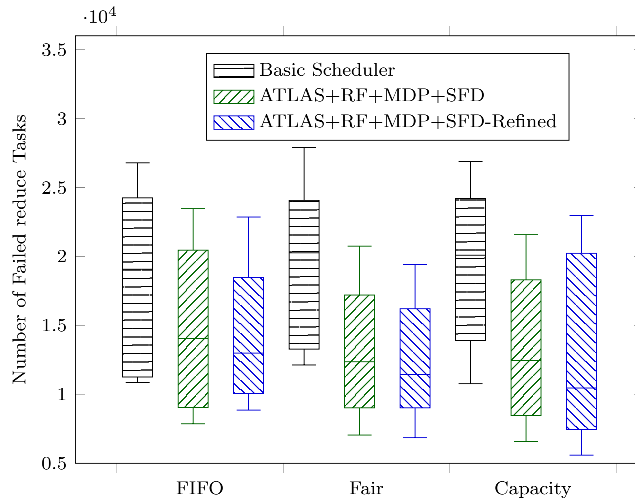

Figure 5.5: Impact of Verification Guidelines on Failed Reduce Tasks

5.5 Summary

In this chapter, we described a new methodology to formally analyze existing Hadoop schedulers using model checking. The aim of the proposed methodology is to investigate the relationship between the scheduling decisions and the failures rates in a Hadoop cluster and to propose possible scheduling strategies to avoid tasks' failures. Particularly, it allows to verify three important scheduling properties, namely schedulability, resources-deadlock freeness, and fairness. We used CSP to model the Hadoop schedulers, and the PAT model checker to verify the mentioned properties. We applied our methodology on the scheduler of OpenCloud, a real Hadoop-based cluster and on Hadoop schedulers to illustrate its usability and benefits. The analysis results showed that it is possible to early identify failures of tasks in a Hadoop application before deploying it. Then, we evaluate the failures' rates in ATLAS given the provided scheduling strategies.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Hadoop has become the *de facto* standard for processing large data in today's cloud environment. The performance of Hadoop has a direct impact on many important applications running in cloud environments. However, because of the scale, complexity and dynamic nature of the cloud, failures are common and these failures often impact the performance of jobs running in Hadoop. Despite the diversity of failures detection and recovery mechanisms integrated in Hadoop, several scheduled jobs still fail because of unforeseen events in the cloud environment such as unpredicted demands of services or hardware outages. This is due to the fact that the Hadoop scheduler may generate poor scheduling decisions leading to several jobs' failures. Also, it lacks mechanisms to share information about failures between the different Hadoop components. Moreover, the Hadoop scheduler is not able to quickly detect the failures of TaskTrackers due to the fixed heartbeat-based failure detection commonly used in Hadoop. On the other hand, simulation and analytical modeling have been widely used to identify the impact of scheduling decisions on the failures rates in Hadoop.

However, it was found that they cannot explore large clusters and provide accurate results and exhaustive coverage about the failures in Hadoop.

To alleviate these issues, we presented in this thesis new approaches for modeling and verifying an adaptive failure-aware scheduling algorithm to adjust the scheduling decisions in Hadoop. The proposed approaches can help reduce the failures rates of tasks and TaskTrackers in the Hadoop framework according to changes in the cloud environment. In the sequel, we present a description of each of the main contributions of this thesis.

The first contribution of this thesis is the development of a methodology for task failure detection to allow the early prediction of the scheduling outcomes of tasks using information about the tasks and machine learning algorithms. Our methodology allows to investigate the correlation between tasks attributes and its scheduling outcomes. To deploy the proposed methodology, we implement a predictive method to use information about previously executed tasks and machine learning algorithms to predict possible outcomes of scheduled tasks. The obtained results show that the Random Forest algorithm achieves the best results in terms of precision, recall, accuracy, and execution time for the three Hadoop schedulers.

Second, we propose a novel method to generate adaptive scheduling decisions to reduce tasks' failures and avoid making poor scheduling decisions in the Hadoop framework. To do so, we model the life cycle of a task as an MDP in which each action is associated with a reward. To solve the MDP model, we use reinforcement learning techniques to select the scheduling strategy that minimizes the risk of failure for each submitted task. We implement an adaptive algorithm to select scheduling actions for the MDP model in Hadoop based on the Q-Learning and SARSA algorithms and evaluate their performance. The obtained results showed that SARSA algorithm

can explore more policies whereas Q-Learning algorithm can select policies giving maximum rewards for Hadoop scheduler

The third contribution of this thesis is the implementation of a dynamic algorithm to adjust the communication between the JobTracker and TaskTrackers in order to quickly detect the failures of the TaskTracker nodes, instead of the fixed heartbeat-based failure detection approach. For this purpose, we used four well known algorithms from the network field: Chen Failure Detector, Bertier Failure Detector (Bertier-FD), ϕ Failure Detector (ϕ-FD) and Self-tuning Failure Detector (SFD). We adapt the existing implementations of these algorithms in Hadoop scheduler to quickly detect the failures of TaskTrackers based on collected information about previously received heartbeats. The evaluation of these algorithms under different failures rates showed that the ϕ-*FD* and *SFD* algorithms outperform the other algorithms in terms of detection time and error rate.

To illustrate the usability and benefits of these proposed methodologies, we implemented an AdapTive and faiLure-Aware Scheduling (*ATLAS*) algorithm to early track failures and adjust the scheduling decisions on the fly. Concretely, we used the failure prediction algorithm based on the Random Forest to early identify the failure for ATLAS. To improve the Hadoop scheduling decisions on the fly, we trained ATLAS using both SARSA and Q-Learning algorithms to generate decisions minimizing tasks' failures. To adjust the communication between the JobTracker and TaskTrackers in ATLAS Hadoop, we used the SFD based algorithm to dynamically track failures of TaskTrackers.

We conducted a large empirical study on a 1000-nodes Hadoop cluster deployed on Amazon Elastic MapReduce (EMR) to compare the performance of ATLAS to those of three Hadoop scheduling algorithms (FIFO, Fair, and Capacity). Results

show that ATLAS outperforms FIFO, Fair, and Capacity scheduling algorithms, and it could reduce the failures rates for jobs and tasks by up to 49% and 67%, respectively. Furthermore, it could reduce the total execution time of jobs and tasks by 35% and 42%, respectively. Consequently, the CPU and memory usage was reduced by 25% and 24%, respectively.

Finally, to identify circumstances leading to potential failures in Hadoop, we implemented a new methodology to formally analyze the Hadoop schedulers. The proposed methodology allows to identify the impact of the scheduling decisions of Hadoop on the failures rates. Towards this goal, we verified some of the most important scheduling properties in Hadoop including schedulability, resources-deadlock freeness, and fairness using the CSP language and the PAT model checker. We applied our methodology on the scheduler of OpenCloud, a real Hadoop-based cluster. Then, we applied our formal analysis to early identify failures of tasks in a Hadoop cluster for ATLAS.

6.2 Future Work

Given the challenges and the learned lessons we gained while doing this thesis, we can propose several research directions that can be pursued to improve the performance of ATLAS.

In our task failure detection methodology, we used supervised learning algorithms to train the failure predictive algorithm. The evaluation of the used algorithms showed that they can early identify tasks' failures with a good accuracy and precision. Indeed, the performance of the supervised algorithms highly depends on the size of the training data: the larger the training data is, the better is the algorithms performance. Here, one can train these algorithms using different sizes of data sets and

evaluate the correlation between them and the accuracy of the algorithms. In addition, we can build the proposed failure predictive algorithm by training unsupervised algorithms and evaluate their impacts on ATLAS and Hadoop schedulers.

When building the failure predictive algorithm, we did not use information about the requested resources because these information were not available in the collected logs from the Hadoop cluster. However, it was found that several tasks may fail because of lack of resources or unexpected resources contentions by larger tasks more than expected. So, our predictive algorithm would hardly predict such failures. This limitation may be solved by building a model to predict the requested resources for the scheduled tasks based on the task characteristics, received workload, running tasks on TaskTrackers, etc. Characteristics of workload to be scheduled over time using ATLAS are important factors that can affect the processing of the tasks and the failures rates of tasks. Therefore, it is of interest to study these characteristics and their impacts on the scheduling decisions and one can build an algorithm to predict the arrival of workload over time to adjust the predictions of tasks failures in ATLAS.

In our task adaptive scheduling methodology, we can extend the scheduling decisions of ATLAS using procedures to optimize the resources allocation across tasks. Indeed, the existing scheduling strategies in ATLAS consider the policies that improve the number of finished tasks while using the available resources slots. Although these strategies could reduce the number of failed tasks and reduce the overall cluster utilization, it was found that ATLAS is not using the available resources efficiently. One important direction to improve the proposed scheduling learning algorithm is the training of the selected reinforcement learning algorithms as a function of the size of the cluster where ATLAS would be deployed.

The performance of the four selected algorithms from the network field highly

depend on the cluster size and the injected failures. Consequently, we can improve their performance by training them in different cluster sizes and different failures rates. While building our TaskTracker failure detection methodology, we observed that the SFD algorithm is able to quickly the failures of nodes in Hadoop. However, it was found to make mistakes because of network delays or messages loss. Therefore, it is necessary to implement a procedure to analyze these mistakes and to learn how to avoid them. Similar to our task failure detection methodology, one future work can be to implement a procedure to predict the failure of TaskTracker using machine learning algorithms and train it over time.

The proposed method to perform the formal analysis of Hadoop scheduler is efficient in the sense that it enables the early identification of tasks failures in a Hadoop application before deploying it. However, it is designed to only include these three important scheduling constraints: data locality, data placement, and speculative execution. An important direction of future work can be to add more features, to our proposed approach, that can impact the failures rate in Hadoop, like the resource assignment and load balancing. Furthermore, we noticed that our formal analysis approach is characterized by a higher rate of wrong identification of failed tasks (up to 40%), which impacts the performance of our scheduler. This is because of the built-in recovery mechanisms in the real cluster that are not involved in our approach. Therefore, we propose to model important internal recovery mechanisms of Hadoop and evaluate their impact of the performance of our formal approach as well us the failures rates. Moreover, our proposed formal analysis is limited to early identify failures of tasks. However, we can extend this work to formally analyze the failures of TaskTrcakers and evaluate their impact on the scheduling decisions and the failures rates of scheduled tasks.

Finally, to generalize the findings of this thesis, we propose to extend and evaluate our proposed approach on Spark [34], a novel in-memory computing framework for Hadoop. One can adapt the proposed four methodologies according to the architecture of Spark.

Bibliography

[1] Amazon EC2 Instances. http://aws.amazon.com/ec2/instance-types/, 2018.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Datfa Processing on Large Clusters. *ACM Communications*, 51(1):107–113, 2008.

[3] Apache Hadoop, http://hadoop.apache.org/, 2018.

[4] I. Foster, Z. Yong, I. Raicu, and L. Shiyong. Cloud Computing and Grid Computing 360-Degree Compared. In *International Workshop on Grid Computing Environments*, pages 1–10, 2008.

[5] H. Mohamed and S. Marchand-Maillet. Enhancing MapReduce Using MPI and an Optimized Data Exchange Policy. In *International Conference on Parallel Processing*, pages 11–18, 2012.

[6] Y.C. Kao and Y.S. Chen. Data-locality-aware MapReduce Real-time Scheduling Framework. *Journal of Systems and Software*, 112:65 – 77, 2016.

[7] H. Chen, Y. Lu, and D. Swanson. Matchmaking: A New MapReduce Scheduling Technique. In *International Conference on Cloud Computing Technology and Science*, pages 40–47, 2011.

[8] J. Yuting, T. Lang, H. Ting, T. Jian, L. Kang-won, and Z. Li. Improving Multi-job MapReduce Scheduling in an Opportunistic Environment. In *International Conference on Cloud Computing*, pages 9–16, 2013.

[9] A. Raj, K. Kaur, U. Dutta, V.V. Sandeep, and S. Rao. Enhancement of Hadoop Clusters with Virtualization Using the Capacity Scheduler. In *International Conference on Services in Emerging Markets*, pages 50–57, 2012.

[10] J. Dean. Experiences with MapReduce, an Abstraction for Large-scale Computation. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 1–1, 2006.

[11] K.V. Vishwanath and N. Nagappan. Characterizing Cloud Computing Hardware Reliability. In *ACM Symposium on Cloud Computing*, pages 193–204, 2010.

[12] F. Dinu and T. Ng. Hadoop Overload Tolerant Design Exacerbates Failure Detection and Recovery. In *International Workshop on Networking Meets Databases*, pages 1–7, 2011.

[13] S. Kikuchi and T. Aoki. Evaluation of Operational Vulnerability in Cloud Service Management Using Model Checking. In *International Symposium on Service Oriented System Engineering*, pages 37–48, March 2013.

[14] Amazon, Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region, 2013.

[15] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace. WOHA: Deadline-Aware Map-Reduce Workflow Scheduling Framework over Hadoop Clusters. In *IEEE International Conference on Distributed Computing Systems*, pages 93–103, 2014.

[16] J.A. Quian-Ruiz, C. Pinkel, J. Schad, and J. Dittrich. RAFTing MapReduce: Fast Recovery on the RAFT. In *IEEE International Conference on Data Engineering*, pages 589–600, 2011.

[17] M. Soualhia, F. Khomh, and S. Tahar. Task Scheduling in Big Data Platforms: A Systematic Literature Review. *Journal of Systems and Software*, 134:170 – 189, 2017.

[18] F. Dinu and T.S.E. Ng. Understanding the Effects and Implications of Compute Node Related Failures in Hadoop. In *Symposium on High-Performance Parallel and Distributed Computing*, pages 187–198, 2012.

[19] D. Chen, Y. Chen, B. N. Brownlow, P. P. Kanjamala, C. A. G. Arredondo, B. L. Radspinner, and M. A. Raveling. Real-Time or Near Real-Time Persisting Daily Healthcare Data Into HDFS and ElasticSearch Index Inside a Big Data Platform. *IEEE Transactions on Industrial Informatics*, 13(2):595–606, 2017.

[20] Applying Apache Hadoop to NASA's Big Climate Data. http://events.linuxfoundation.org/sites/events/files/slides/apachecon_nasa_hadoop.pdf, 2018.

[21] T. Chen and R. Bahsoon. Self-Adaptive and Online QoS Modeling for Cloud-Based Software Services. *IEEE Transactions on Software Engineering*, 43(5):453–475, 2017.

[22] V. Di Valerio and F. Lo Presti. Optimal Virtual Machines Allocation in Mobile Femto-Floud Fomputing: An MDP Fpproach. In *IEEE Wireless Communications and Networking Conference Workshops*, pages 7–11, 2014.

[23] X. Bu, J. Rao, and C. Z. Xu. Coordinated Self-Configuration of Virtual Machines and Appliances Using a Model-Free Learning Approach. *IEEE Transactions on Parallel and Distributed Systems*, 24(4):681–690, 2013.

[24] F. Alexander and H. Matthias. Improving Scheduling Performance Using a Q-learning-based Leasing Policy for Clouds. In *International Conference on Parallel Processing*, pages 337–349, 2012.

[25] Csaba Szepesvari. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010.

[26] W. Chen, S. Toueg, and M. K. Aguilera. On The Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.

[27] M. Bertier, O. Marin, and P. Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In *IEEE Conference on Dependable Systems and Networks*, pages 354–363, 2002.

[28] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The ϕ; Accrual Failure Detector. In *IEEE International Symposium on Reliable Distributed Systems*, pages 66–78, 2004.

[29] N. Xiong, A. V. Vasilakos, J. Wu, Y. R. Yang, A. Rindos, Y. Zhou, W. Z. Song, and Y. Pan. A Self-tuning Failure Detection Scheme for Cloud Computing Service. In *IEEE International Parallel Distributed Processing Symposium*, pages 668–679, 2012.

[30] C. Baier and J-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[31] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating Specification and Programs for System Modeling and Verification. In *IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 127–135, 2009.

[32] Process Analysis Toolkit. http://sav.sutd.edu.sg/pat/, 2018.

[33] B. Kitchenham. Procedure for Performing Systemic Reviews. Technical report, Keele University and NICTA, Australia, 2004.

[34] W. Huang, L. Meng, D. Zhang, and W. Zhang. In-Memory Parallel Processing of Massive Remotely Sensed Data Using an Apache Spark on Hadoop YARN Model. *Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, pages 1–17, 2016.

[35] J. Xu, Z. Chen, J. Tang, and S. Su. T-Storm: Traffic-Aware Online Scheduling in Storm. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, pages 535–544, 2014.

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

[37] F. Dinu and T.S.E. Ng. RCMP: Enabling Efficient Recomputation Based Failure Resilience for Big Data Analytics. In *International Parallel and Distributed Processing Symposium*, pages 962–971, 2014.

[38] H. Zhu and H. Chen. Adaptive Failure Detection via Heartbeat under Hadoop. In *IEEE Asia-Pacific Services Computing Conference*, pages 231–238, 2011.

[39] S.Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making Cloud Intermediate Data Fault-tolerant. In *ACM Symposium on Cloud Computing*, pages 181–192, 2010.

[40] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *USENIX Conference on Operating Systems Design and Implementation*, pages 265–278, 2010.

[41] J.A. Quiane-Ruiz, C. Pinkel, J. Schad, and J. Dittrich. RAFTing MapReduce: Fast recovery on the RAFT. In *International Conference on Data Engineering*, pages 589–600, 2011.

[42] Z. Yuan and J. Wang. Research of Scheduling Strategy Based on Fault Tolerance in Hadoop Platform. In *Geo-Informatics in Resource Management and Sustainable Ecosystem*, volume 399 of *Communications in Computer and Information Science*, pages 509–517. Springer, 2013.

[43] C. Gupta, M. Bansal, Tzu-Cheng Chuang, R. Sinha, and S. Ben-romdhane. Astro: A Predictive Model for Anomaly Detection and Feedback-based Scheduling on Hadoop. In *International Conference on Big Data*, pages 854–862, 2014.

[44] C. Qi, L. Cheng, and X. Zhen. Improving MapReduce Performance Using Smart Speculative Execution Strategy. *IEEE Transactions on Computers*, 63(4):954–967, 2014.

[45] T. Shanjiang, L. Bu-Sung, and H. Bingsheng. DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters. *IEEE Transactions on Cloud Computing*, 2(3):333–347, 2014.

[46] L. Lei, W. Tianyu, and H. Chunming. CREST: Towards Fast Speculation of Straggler Tasks in MapReduce. In *Proceedings of International Conference on e-Business Engineering*, pages 311–316, 2011.

[47] O. Yildiz, S. Ibrahim, T. A. Phuong, and G. Antoniu. Chronos: Failure-aware Scheduling in Shared Hadoop Clusters. In *IEEE International Conference on Big Data (Big Data)*, pages 313–318, 2015.

[48] O. Yildiz, S. Ibrahim, and G. Antoniu. Enabling Fast Failure Recovery in Shared Hadoop Clusters: Towards Failure-aware Scheduling. *Future Generation Computer Systems*, 74:208 – 219, 2017.

[49] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX Conference on Operating Systems Design and Implementation*, pages 29–42, 2008.

[50] C. Quan, Z. Daqiang, G. Minyi, D. Qianni, and G. Song. SAMR: A Self-Adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment. In *International Conference on Computer and Information Technology*, pages 2736–2743, 2010.

[51] S. Xiaoyu, H. Chen, and L. Ying. ESAMR: An Enhanced Self-Adaptive MapReduce Scheduling Algorithm. In *International Conference on Parallel and Distributed Systems*, pages 148–155, 2012.

[52] T. Zhuo, J. Lingang, Z. Junqing, L. Kenli, and L. Keqin. A Self-Adaptive Scheduling Algorithm for Reduce Start Time. *Future Generation Computer Systems*, 4344(C):51 – 60, 2015.

[53] W. Jiayin, Y. Yi, M. Ying, S. Bo, and M. Ningfang. FRESH: Fair and Efficient Slot Configuration and Scheduling for Hadoop Clusters. In *International Conference on Cloud Computing*, pages 761–768, 2014.

[54] Y. Guo, J. Rao, C. Jiang, and X. Zhou. Moving MapReduce into the Cloud with Flexible Slot Management and Speculative Execution. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):798–812, 2017.

[55] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace. WOHA: Deadline-Aware Map-Reduce Workflow Scheduling Framework over Hadoop Clusters. In *International Conference on Distributed Computing Systems*, pages 93–103, 2014.

[56] A. Rasooli and D.G. Down. A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems. In *International Conference on High Performance Computing, Networking Storage and Analysis*, pages 1284–1291, 2012.

[57] O. Hasan and S. Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology*, pages 7162–7170, IGI Global Pub., 2015.

[58] S.T. Cheng, H.C. Wang, Y.J. Chen, and C.F. Chen. Performance Analysis Using Petri Net Based MapReduce Model in Heterogeneous Clusters. In *Advances in Web-Based Learning*, volume 8390 of *Lecture Notes in Computer Science*, pages 170–179. Springer, 2015.

[59] M.C. Ruiz, J. Calleja, and D. Cazorla. Petri Nets Formalization of Map/Reduce Paradigm to Optimise the Performance-Cost Tradeoff. In *IEEE Trustcom/BigDataSE/ISPA*, volume 3, pages 92–99, 2015.

[60] W. Su, F. Yang, H. Zhu, and Q. Li. Modeling MapReduce with CSP. In *IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 301–302, 2009.

[61] W. Xie, H. Zhu, X. Wu, S. Xiang, and J. Guo. Modeling and Verifying HDFS Using CSP. In *IEEE Annual Computer Software and Applications Conference*, volume 1, pages 221–226, 2016.

[62] G.S. Reddy, F. Yuzhang, L. Yang, S.D. Jin, J. Sun, and R. Kanagasabai. Towards Formal Modeling and Verification of Cloud Architectures: A Case Study on Hadoop. In *International World Congress on Services*, pages 306–311, 2013.

[63] K. Ono, Y. Hirai, Y. Tanabe, N. Noda, and M. Hagiya. Using Coq in Specification and Program Extraction of Hadoop Mapreduce Applications. In *Software Engineering and Formal Methods*, volume 7041 of *Lecture Notes in Computer Science*, pages 350–365, 2011.

[64] The Coq Proof Assistant, https://coq.inria.fr/, 2018.

[65] WordCount Example. http://wiki.apache.org/hadoop/wordcount, 2018.

[66] Y. Jarraya, A. Eghtesadi, M. Debbabi, Y. Zhang, and M. Pourzandi. Formal Verification of Security Preservation for Migrating Virtual Machines in the Cloud. In *Stabilization, Safety, and Security of Distributed Systems*, volume 7596 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2012.

[67] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *Principles of Security and Trust*, volume 7796 of *Lecture Notes in Computer Science*, pages 126–146. Springer, 2013.

[68] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[69] NuSMV: A New Symbolic Model Checker, http://nusmv.fbk.eu/, 2018.

[70] S. Kikuchi and Y. Matsumoto. Performance Modeling of Concurrent Live Migration Operations in Cloud Computing Systems Using PRISM Probabilistic Model Checker. In *International Conference on Cloud Computing*, pages 49–56, 2011.

[71] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas. Cloud elasticity using probabilistic model checking. Technical report, Aristotle University of Thessaloniki, Greece, 2014.

[72] V. Ishakian, A. Lapets, A. Bestavros, and A. Kfoury. Formal Verification of SLA Transformations. In *World Congress on Services*, pages 540–547, July 2011.

[73] B. Thuraisingham, V. Khadilkar, J. Rachapalli, T. Cadenhead, M. Kantarcioglu, K. Hamlen, L. Khan, and F. Husain. Towards the Design and Implementation of a Cloudcentric Assured Information Sharing System. Technical report, UTDCS-27-11, The University of Texas at Dallas, USA, 2012.

[74] M. Kaufmann and J.S. Moore. ACL2: an industrial strength version of Nqthm. In *International Conference on Computer Assurance, Systems Integrity, Software Safety, Process Security*, pages 23–34, 1996.

[75] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang. Hadoop Performance Modeling for Job Estimation and Resource Provisioning. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):441–454, 2016.

[76] D. Liu, S. Y. Cho, D. M. Sun, and Z. D. Qiu. A Spearman Correlation Coefficient Ranking for Matching-score Fusion on Speaker Recognition. In *IEEE Region 10 Conference TENCON*, pages 736–741, 2010.

[77] The R Project for Statistical Computing, http://www.r-project.org/, 2018.

[78] M. Fokkema. pre: An R Package for Fitting Prediction Rule Ensembles. *ArXiv e-prints, http://adsabs.harvard.edu/abs/2017arXiv170707149F*, 2017.

[79] L. An, F. Khomh, and B. Adams. Supplementary Bug Fixes vs. Re-opened Bugs. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 205–214, 2014.

[80] B. Jeannet, P. D'Argenio, and K. Larsen. Rapture: A Tool for Verifying Markov Decision Processes. In *International Conference on Concurrency Theory*, pages 84–98, 2002.

[81] G. Oddi, M. Panfili, A. Pietrabissa, L. Zuccaro, and V. Suraci. A Resource Allocation Algorithm of Multi-cloud Resources Based on Markov Decision Process. In *IEEE International Conference on Cloud Computing Technology and Science*, pages 130–135, 2013.

[82] N. Mastronarde and M. van der Schaar. Online Reinforcement Learning for Dynamic Multimedia Systems. *IEEE Transactions on Image Processing*, 19(2):290–305, 2010.

[83] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang. A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning. In *IEEE International Conference on Distributed Computing Systems*, pages 372–382, 2017.

[84] M. Duggan, K. Flesk, J. Duggan, E. Howley, and E. Barrett. A Reinforcement Learning Approach for Dynamic Selection of Virtual Machines in Cloud Data Centres. In *International Conference on Innovative Computing Technology*, pages 92–97, 2016.

[85] Z. Peng, D. Cui, Y. Ma, J. Xiong, B. Xu, and W. Lin. A Reinforcement Learning-Based Mixed Job Scheduler Scheme for Cloud Computing under SLA Constraint. In *IEEE International Conference on Cyber Security and Cloud Computing*, pages 142–147, 2016.

[86] F. Farahnakian, P. Liljeberg, and J. Plosila. Energy-Efficient Virtual Machines Consolidation in Cloud Data Centers Using Reinforcement Learning. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 500–507, 2014.

[87] M. van der Ree and M. Wiering. Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 108–115, 2013.

[88] R.M. Obrien. A Caution Regarding Rules of Thumb for Variance Inflation Factors. *Quality and Quantity*, 41(5):673–690, 2007.

[89] Richard Goldstein. Conditioning Diagnostics: Collinearity and Weak Data in Regression. 35:85–86, 2012.

[90] Traces of Google workloads, http://code.google.com/p/googleclusterdata/, 2018.

[91] F. Faghri, S. Bazarbayev, M. Overholt, R. Farivar, R. Campbell, and W. H. Sanders. Failure Scenario As a Service (FSaaS) for Hadoop Clusters. In *Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, pages 5:1–5:6, 2012.

[92] M. Soualhia, F. Khomh, and S. Tahar. Predicting Scheduling Failures in the Cloud: A Case Study with Google Clusters and Hadoop on Amazon EMR. In *IEEE High Performance Computing and Communications*, pages 58–65, 2015.

[93] S. Jimnez Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean. Open Challenges for Probabilistic Measurement-Based Worst-Case Execution Time. *IEEE Embedded Systems Letters*, 9(3):69–72, 2017.

[94] R.K. Yin. *Case Study Research: Design and Methods*. SAGE Publication, 2002.

[95] Microsoft Azure, http://azure.microsoft.com/en-gb/, 2018.

[96] A.M.K. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification*. John Wiley & Sons, Inc., 2002.

[97] OpenCloud. http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html, 2018.

[98] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *Computer Aided Verification:CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714, 2009.

[99] Google Traces. https://github.com/google/cluster-data, 2018.

[100] Facebook Traces. https://github.com/swimprojectucb/swim/wiki/workloads-repository, 2018.

[101] J. Fang, Headroom Miscalculation can lead to Deadlock in Hadoop Two, http://johnjianfang.blogspot.ca/2014/09/headroom-miscalculation-can-lead-to.html, 2018.

Biography

Education

- **Concordia University**: Montreal, Quebec, Canada

  Ph.D. degree, Dept. of Electrical & Computer Engineering, (May 2013 - April 2018)

- **École de Technologie Supérieure**: Montreal, Quebec, Canada

  Master's degree in Engineering: Information Technology, (January 2011 - April 2013)

- **École Nationale Supérieure d'Ingénieurs de Tunis**: Tunis, Tunisia

  Licence degree in Computer Science, (September 2007 - June 2010)

Awards

- Concordia University Conference and Exposition Award, Concordia University, Canada (2015).

- Tunisia National Bursary for Ph.D study in Canada, (2013-2016).

- Tunisia National Bursary for M.A.Sc study in Canada, (2011-2012).

Work History

- **Research Assistant**, Hardware Verification Group, Dept. of Electrical and Computer Engineering, Concordia University, Montreal, Quebec, Canada (2013-2018)
- **Teaching Assistant**, Design Pattern, Empirical Software Studies (LOG6306), Dept. of Software & Computer Engineering, École Polytechnique Montréal, Montreal, Quebec, Canada (2016)
- **Research Assistant**, Dept. of Software & Computer Engineering, École de Technologie Supérieure, Montreal, Quebec, Canada (2011-2013)

Publications

- **Journal Papers**
  - **Bio-Jr1** M. Soualhia, F. Khomh, and S. Tahar, A Dynamic and Failure-aware Task Scheduling Framework for Hadoop; IEEE Transactions on Cloud Computing, pp. 1-16, 2018, To Appear.
  - **Bio-Jr2** M. Soualhia, F. Khomh, and S. Tahar, Task Scheduling in Big Data Platforms: A Systematic Literature Review; Journal of Systems and Software, vol. 134, pp. 170-189, Elsevier, 2017.
  - **Bio-Jr3** N. Kara, M. Soualhia, F. Belqasmi, C. Azar and R. H. Glitho, Genetic-based Algorithms for Resource Management in Virtualized IVR Applications; Journal of Cloud Computing, vol. 3, pp. 1-15, Springer, 2014.

- **Bio-Jr4** F. Belqasmi, C. Azar, R. H. Glitho, M. Soualhia and N. Kara, A Case Study on IVR Applications' Provisioning as Cloud Computing Services; IEEE Network, vol. 28, no. 1, pp. 33-41, 2014.

- **Refereed Conference Papers**

  - **Bio-Cf1** M. Soualhia, F. Khomh, and S. Tahar, Formal Analysis of Hadoop Scheduler. [(Submitted), NASA Formal Methods Symposium (NFM'18), April 2018, pp. 1-15].
  - **Bio-Cf2** M. Soualhia, F. Khomh, and S. Tahar, ATLAS: An Adaptive Failure-Aware Scheduler for Hadoop. [Proc. IEEE International Performance Computing and Communications Conference (IPCCC'15), Nanjing, China, December 2015, pp. 1-8].
  - **Bio-Cf3** M. Soualhia, F. Khomh, and S. Tahar: Predicting Scheduling Failures in the Cloud: A Case Study with Google Clusters and Hadoop on Amazon EMR. [Proc. IEEE High Performance Computing and Communications (HPCC'15), New York, USA, August 2015, pp. 58-65].
  - **Bio-Cf4** F. Belqasmi, C. Azar, M. Soualhia, N. Kara and R.H. Glitho, A Virtualized Infrastructure for IVR Applications as services. [Proc. ITU Kaleidoscope 2011: The Fully Networked Human? -Innovations for Future Networks and Services, (ITU'11), Cape Town, South Africa, December 2011, 12-14].

- **Technical Reports**

  - **Bio-Tr1** M. Soualhia, F. Khomh, and S. Tahar, ATLAS: An Adaptive Failure-Aware Scheduler for Hadoop, Technical Report, Department

of Electrical and Computer Engineering, Concordia University, November 2015. [24 Pages]. https://arxiv.org/abs/1511.01446

- **Bio-Tr2** M. Soualhia, F. Khomh, and S. Tahar, Predicting Scheduling Failures in the Cloud, Technical Report, Department of Electrical and Computer Engineering, Concordia University, July 2015. [26 Pages]. https://arxiv.org/abs/1507.03562