A Dynamic and Failure-aware Task Scheduling Framework for Hadoop

Mbarka Soualhia, *Student Member, IEEE*, Foutse Khomh, *Member, IEEE*, and Sofiène Tahar, *Senior Member, IEEE*

Abstract—Hadoop has become a popular framework for processing data-intensive applications in cloud environments. A core constituent of Hadoop is the scheduler, which is responsible for scheduling and monitoring the jobs and tasks, and rescheduling them in case of failures. Although fault-tolerance mechanisms have been proposed for Hadoop, the performance of Hadoop can be significantly impacted by unforeseen events in the cloud environment. In this paper, we introduce a dynamic and failure-aware framework that can be integrated within Hadoop scheduler and adjust the scheduling decisions based on collected information about the cloud environment. Our framework relies on predictions made by machine learning algorithms and scheduling policies generated by a Markovian Decision Process (MDP), to adjust its scheduling decisions on the fly. Instead of the fixed heartbeat-based failure detection commonly used in Hadoop to track active TaskTrackers (*i.e.*, nodes that process the scheduled tasks), our proposed framework implements an adaptive algorithm that can dynamically detect the failures of the TaskTracker. To deploy our proposed framework, we have built, ATLAS+, an AdapTive Failure-Aware Scheduler for Hadoop. To assess the performance of ATLAS+, we conduct a large empirical study on a 100-node Hadoop cluster deployed on Amazon Elastic MapReduce (EMR), comparing the performance of ATLAS+ with those of three Hadoop schedulers (FIFO, Fair, and Capacity).Results show that ATLAS+ outperforms FIFO, Fair, and Capacity schedulers. ATLAS+ can reduce the number of failed jobs by up to 43% and the number of failed tasks by up to 59%. On average, ATLAS+ could reduce the total execution time of jobs by 10 minutes, which represents 40% of the job execution times, and by up to 3 minutes for tasks, which represents 47% of the task execution time. ATLAS+ also reduced CPU and memory usage by 22% and 20%, respectively.

Keywords—Adaptive Scheduling, Failure-Aware Scheduling, Hadoop, MapReduce, ATLAS+.

1 Introduction

MAPREDUCE [1] has become a major programming model for processing large data sets in cloud computing environments. Hadoop [2], an open-source implementation of MapReduce has become the cornerstone technology of many big data and cloud applications. It has been deployed in many leading companies (*e.g.*, Yahoo!, and Facebook) to manage applications ranging from web analytic, web indexing, image and document processing, to high-performance scientific computing and social network analysis [3]. A key advantage of Hadoop over other big data processing frameworks is that it allows for efficient data processing across clusters of commodity servers. Despite the different failure detection and recovery mechanisms integrated within Hadoop cluster, many task failures still occur because of unforeseen events in the cloud environment. In fact, in the cloud, failures are the norm rather than exceptions. Liu *et al.* [4] discusses the impact of different types of failures (correlated and non-correlated) for nodes when processing tasks and jobs in cloud environments. For instance, they claim that tasks may fail because of data loss, which is likely caused by correlated and non-correlated machine failures in storage systems. Studies report that a cluster can experience more than one thousand individual node failures and thousands of hard-drive failures during its first year of service [4] [5]. In addition, several power problems can also happen bringing down between 500 and 1000 nodes for up to 6 hours. The recovery time of these failed nodes being as high as 2 days. These frequent failures in data centres negatively impact the performance of applications running Hadoop [5], [6]. Furthermore, in the current implementations of Hadoop, the nodes send heartbeat messages to the scheduler at fixed time intervals, and the scheduler checks the received heartbeats from the nodes also at fixed interval time [5]. Consequently, when a node failure occurs, the scheduler can detect this failure only within the next interval time. The scheduler considers this node as dead, and the running tasks on it as failed and will restart them from scratch on other nodes. In the meantime, the scheduler can assign tasks to the failed nodes, which would likely increase the task failures rate. Therefore, the early detection of tasks and nodes (*e.g.*, TaskTracker: the node that processes the scheduled tasks, DataNode: the node that manages the data stored in Hadoop) failure in Hadoop clusters is important to improve the performance of Hadoop applications. It is also important to build efficient scheduling strategies that adjust to the unpredictable changes of a cloud environment.

In a previous work, we have shown that it is possible to predict task and jobs failures in a cloud environment [7]. Based on the findings of this work, we have proposed a new scheduler for Hadoop called ATLAS (AdapTive faiLure-Aware Scheduler), that adapts its scheduling decisions to events occurring in the cloud environment. We have also shown that ATLAS can help reduce tasks and jobs failures in Hadoop clusters by up to 39% and 28%, respectively [8]. But, we observe that some jobs and tasks are still failing, because the approach proposed in [8] does not generate efficient scheduling policies. Also, it does not consider the machines' constraints and the dynamic behavior of the environment where the tasks are executed (*e.g.*, availability of resources, reliability of the TaskTrackers, number of running tasks on a machine, network congestion). Moreover, it lacks mechanisms to decide whether it is better to immediately process a task or to wait until its success conditions are met. This can have a large impact on failures rate, since it does not ensure making efficient scheduling decisions. For instance, when a task is waiting for reduce slots and the TaskTracker cannot release these slots, a better decision could be to kill the task and reschedule it on another TaskTracker. This is in order to avoid a long waiting time in the queue on the JobTracker especially when there are multiple jobs running concurrently. Furthermore, the approach proposed in ATLAS [8] does not possess a dynamic mechanism for detecting TaskTracker failures. In this paper, we build on these early findings and propose a dynamic and failure-aware scheduling framework for Hadoop that adapts its scheduling decisions to events occurring in the cloud environment. Using historical information about events occurring in the cloud environment (*e.g.*, resource depletion on a node of the cluster or failure of a scheduled task), a machine learning algorithm (Random-Forest algorithm), and a Markovian Decision Process (MDP), our framework learns scheduling decisions that reduce failures in the cluster. Specifically, we propose to train different machine learning algorithms using past task executions to predict the scheduling outcome of each new task submitted for scheduling. The algorithm providing the best performance (in terms of accuracy, precision, and time) will be used to predict whether a given task will fail or not, based on its collected attributes. We implement the MDP model of our proposed framework using reinforcement learning techniques to guide the decision making process and evaluate the scheduling decisions in the context of adaptive policy-driven scheduling. In addition, our proposed framework uses an adaptive algorithm to control the communication between the JobTracker and the TaskTrackers in a Hadoop cluster and adjust the interval timeout to consider a TaskTracker as dead based on the occurrence of failures in the Hadoop environment. This is using four well known algorithms from the network field in our framework to predict the expected arrival time of the next heartbeat from a TaskTracker node based on information about recently received heartbeats messages.

To assess the benefits of our proposed solution, we integrate our framework within Hadoop and build ATLAS+, an **A**dap**T**ive fai**L**ure-**A**ware **S**cheduler for Hadoop. We implement ATLAS+ in a 100-node Hadoop cluster deployed on Amazon Elastic MapReduce (EMR) and perform a case study with both single Hadoop jobs (*e.g.*, *WordCount*, *TeraGen*, *Sort*, and *TeraSort*) and chained Hadoop jobs (these jobs are composed of single Hadoop job), to compare the performance of ATLAS+ with those of Hadoop main schedulers (*i.e.*, FIFO, Fair, and Capacity). Each analysis in our case study is repeated 30 times, and we extend the execution of the jobs on a period of 3 days, to ensure that the observations are robust. We also evaluate the impact of continuous learning on the scheduling decisions of ATLAS+. To assess the performance of each scheduler, we compute the total execution times of jobs, the amount of used resources (CPU, memory, disk), the numbers of finished and failed tasks and jobs. Experimental results show that ATLAS+ outperforms FIFO, Fair and Capacity. It can reduce the number of failed jobs by up to 43% and the number of failed tasks by up to 59%. Also, ATLAS+ could reduce the total execution time of jobs by 10 minutes on average; which represents 40% of the job execution times, and by up to 3 minutes for the tasks, which represents 47% of the task execution time. ATLAS+ also reduces CPU and memory usage by 22% and 20%.

The remainder of this paper is organized as follows: Section 2 describes the limitations of the existing Hadoop schedulers. Sections 3 and 4 depict the architecture and implementation of our proposed solution. Section 5 and 6 describe the findings of our work along with a discussion of our approach. Section 7 summarizes the related literature. Section 8 presents our conclusions. Throughout the paper, we will refer to *"JobTracker"* by *JT*, *"TaskTracker"* by *TT*, and *"DataNode"* as *DN*.

2 Limitations of Current Hadoop's Implementation

2.1 Task Failure Detection

In Hadoop, when a component fails (*e.g.*, TT, DN), all tasks running on this node also fail and the recovery time can be long, which can lead to unpredictable execution times and resources wastage. For instance, the average execution time of a job, which is 220 seconds, can reach 1000 seconds under a TT failure and 700 seconds under a DN failure [5]. Dinu *et al.* [5] who analyse the performance of Hadoop clusters report that Hadoop components do not share failure information between JT, TTs, and DNs appropriately. For instance, when a task experiences a failure, information about this failure is not shared with other tasks that depend on the failed task. In fact, when a map task fails, since map and reduce tasks are scheduled separately and there is no exchange of failure information between them, the failure is likely to translate into the failure of the whole job as explained in [8]. This is because reduce tasks may wait for the results of the failed map task until they time out.
Moreover, when a DN fails, this can delay the starting time of speculative execution of some tasks. This is because of the statistical nature of the speculative execution algorithm, which is based on collected data about task progress (*e.g.*, straggling tasks). If a task is making a good execution progress and suddenly a DN experiences a failure, the speculative execution of this task will start

with a delay, since Hadoop expected the same progress from that task. So that task will be speculatively executed later than the time when straggling tasks are usually speculatively executed. Furthermore, the speculative execution of a task could face the same failure. Therefore, a possible solution could be to equip schedulers with mechanisms that enable the early identification of failed tasks and a quick rescheduling of these tasks on available nodes, to reduce the impact of task failures on job execution time.

2.2 TaskTrackers Failure Detection

In [5] [8], it is shown that the JT cannot quickly detect the failures of Hadoop TTs due to the nature of communication over time between the JT and the TTs. As a consequence, it may assign tasks to dead nodes, which could significantly increase the failures rate of tasks. Theoretically, the TTs send heartbeats to the JT every 3 seconds. The JT checks every 200 seconds (3.33 minutes) the timeout condition of the received heartbeats from the TTs. When a TT does not send heartbeats for at least 600 seconds (10 minutes[1]), the JT considers this TT node as dead, and the running tasks on it as failed and will restart them from scratch on other TT nodes, according to their availability [5]. In addition, some heartbeats (may) arrive late to the JT because of network delays or messages losses. In these cases, the JT will consider their corresponding TTs as dead nodes, despite their availability. Furthermore, it will not assign them any load until it receives a new heartbeat from them, which can result in resource wastage.

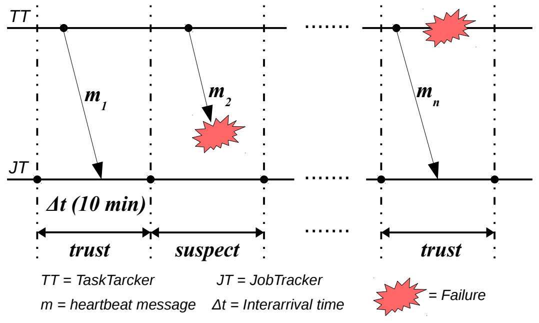


Fig. 1: Failure Detection Model in Hadoop Framework

Figure 1 shows examples of failures that can occur while sending heartbeats between the TTs and the JT. For example, a TT sends a heartbeat message m_1 that arrives before the next arrival time. So, the JT considers this TT as alive. Next, it sends a new heartbeat message m_2 that does not arrive to its destination because of a network problem. The JT considers this node as dead since it does not receive its heartbeat although it is available. This causes resources wastage as this node will not receive new tasks until the next time interval. Another example could be that the TT sends a heartbeat message m_n that arrives before the next time interval. However, this TT experiences a failure and becomes inactive right after sending the message. The JT will consider this node as alive and will keep assigning it new tasks despite the fact that it has failed; which could increase the failure rates of tasks and the execution times of tasks and jobs [5]. Consequently, integrating an adaptive approach to adjust the interval at which the JT considers that a TT is dead can be a possible solution and can help reduce the failure rates of the Hadoop's scheduler.

1. This is the default value in Hadoop, as shown in the "mapred.tasktracker.expiry.interval" property from "mapred-default.xml"

3 FRAMEWORK DESIGN

3.1 General Overview

In this section, we present the architecture of our proposed framework that can predict the scheduling outcomes of tasks (the final output of its execution: either a finished task or a failed task.) using information about the tasks and the cluster environment, and adjust scheduling decisions accordingly. In addition, our proposed framework can adjust the communication between the JT and TTs in order to quickly detect the failures of the TTs nodes. Figure 2 gives a general overview of the structure of our framework. It is comprised of three main components: "*Task Failure Prediction*", "*Dynamic TaskTracker Failure Detection*", and "*Scheduling Policies Modelling*". Specifically, each of the three components is characterized by its own design with respect to its function. Nevertheless, they can be integrated together and built on top of the Hadoop scheduler. The remainder of this section elaborates more on each of these components.

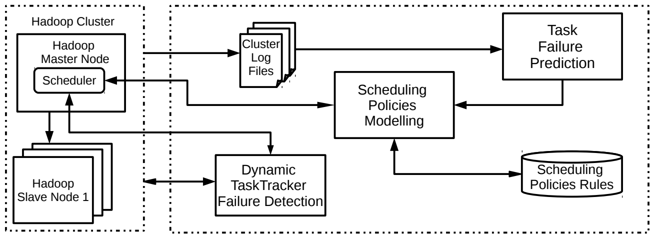


Fig. 2: Architecture of the Proposed Framework

3.2 Task Failure Prediction

This component is responsible for collecting (logs) and processing data about previously executed tasks in a Hadoop cluster. Next, it analyses the correlation between tasks attributes and tasks scheduling outcomes. The results of the correlation analysis are used to identify correlated task attributes, and attributes that are likely to affect a task's scheduling outcome. A machine learning model is trained using these past task executions data. This machine learning model is used to predict the scheduling outcome of each new task submitted for scheduling. The inputs of the machine learning model are the values of the identified predictors (task attributes retained in the model) and the output is the scheduling outcome of a task (either failed or finished). This component of our proposed framework can early identify the tasks that are likely to fail if scheduled on certain nodes, using historical information collected on the Hadoop cluster.

3.3 Dynamic TaskTracker Failure Detection

This component of our proposed framework collects and analyses information about the received heartbeat messages from TTs. Using these information, it can dynamically estimate the arrival time of next heartbeats based on failure occurrences on the TTs. More details about the estimation of next heartbeat arrival time will be presented in Section 4. After this estimation, it calculates the median of the obtained arrival time values. This value will be used to update and adjust the timeout interval at which TT is considered as dead. Specifically, when there are many TT failures, it is expected that the JT will receive heartbeats with a delay. In such case, this component of our framework will notify the scheduler to shorten the interval time of communication between the JT and TTs, in order to quickly detect TTs failures. Following this approach, the scheduler can avoid assigning tasks to dead nodes and resources wastage, and hence reduce task failure rates.

3.4 Scheduling Policies Modelling

To cope with the dynamic nature of cloud environments, our proposed framework requires adaptive scheduling strategies in order to reduce the cost associated with tasks execution. To do so, we propose to consider the scheduling decision process as an MDP [9]. So, the processing of a task can be modelled in terms of a life cycle, where the task progresses through this life cycle and goes from one state to another. The task states in the MDP model are: *submitted, scheduled, waiting, executed, finished, and failed.* The scheduling decision making can be considered as a mapping of states over actions to select a policy according to a derived reward. As shown in Equation 1, the decision process is characterized by two metrics: π^* represents the policy to be applied from one state $S_{(t)}$ to another $S_{(t+1)}$, and R which contains the earned reward by following the selected action $A_{(t)}$. This modelling allows the scheduler to estimate and compare the reward associated with all possible actions to select the scheduling strategy that minimises the risk of failure for each submitted task.

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

More concretely, this component of our framework is responsible for observing the current state of a scheduled task, selecting the possible actions from the current state and observing the derived reward/cost from each transition. The MDP allows to model decision making in environments where the outcomes depend on random factors and are under the control of a decision maker, which corresponds well to the situation of an Hadoop cluster. In fact, MDP models are widely applied to solve decision problems in cloud environment, *e.g.*, resources allocation problems [10], virtual machines scheduling [11], etc. These past successes with MDP models motivated our choice of MDP for Hadoop scheduling.

4 Framework Implementation

In this section, we describe the implementation details of our proposed framework.

4.1 Task Failure Prediction

Using logs collected from previously executed tasks in a Hadoop cluster, we extract job and task attributes to identify correlations between these attributes and the scheduling outcome of a task. To do so, we first propose a formal model to describe task attributes: *Task attributes* = {*Identification, Structure, Execution, Environment*}. The proposed attributes' model can be applied to different processing platforms to collect tasks attributes (to be used later in our proposed framework). In addition, it can give a complete description about the task since it gives an overview about its internal description as well as the way it is executed on its environment. Specifically, we collect the following task attributes according to our proposed model for our proposed framework. 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. Here, we apply the Spearman rank correlation [12] to test the correlation between these attributes and task scheduling outcome.

Next, we train six machine learning models including GLM (General Linear Model), Neural Network, Boost, Tree, Random Forest, and CTree (Conditional Tree), using data collected from past tasks' executions in the cluster [13]. GLM is an extension of linear multiple regression for a single dependent variable. It is extensively used in regression analysis. Neural networks represent interconnected nodes organized in layers. The inputs of the bottom layer represent the predictors, and the outputs of the top layer represent the forecasts of this model. Boost creates a collection of interconnected models iteratively. These successive models are weighted according to their success and their outputs are combined using voting or averaging to create a final model. Decision Tree is extensively used to predict binary outcomes. Random Forest uses a majority voting of decisions for classification and regression results. It offers good out-of-the-box performance and has performed very well in different prediction benchmarks. CTree is an extended implementation of the Decision Tree. The machine learning models aim to predict the scheduling outcome of tasks (*i.e.*, successful completion or failure) based on information about the tasks attributes and Hadoop cluster environment (*i.e.*, availability of resources, failure occurrences in TT, network congestion). The predictors for each model are the task attributes. The output of each model is the scheduling outcome of a task (either failed or finished). For each model, we use the implementation provided in the statistical framework *R* [13]. Next, we compare the performance of the selected models. The model providing the best results will be used to predict whether a given task will fail or not, based on its collected attributes. More details about the collected attributes and the used predictive models can be found in [8]. Algorithm 1 describes the steps followed by the "Task Failure Prediction" component in our framework to predict successes or failures of scheduled tasks.

Algorithm 1 Task Failure Prediction 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, TT)
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
```

4.2 Dynamic TaskTracker Failure Detection

To address the limitations of the current heartbeat-based communication between JT and TTs, we use four well known algorithms from the network field: *Chen Failure Detector (Chen-FD)* [14], *Bertier Failure Detector (Bertier-FD)* [15], *ϕ Failure Detector (ϕ-FD)* [16] and *Self-tuning Failure Detector (SFD)* [17]. We select these algorithms because they have been designed to address message synchronization issues, and have achieved good results when used for detecting failures in network systems. They can significantly reduce the number of false failure detections [18]. We implement (and adapt) these algorithms in our framework to predict the expected arrival time of the next heartbeats from a TT node based on information about recently received heartbeats messages. The four algorithms can adjust the interval timeout at which the JT can consider a TT as dead using collected information about the received heartbeats and TT nodes failure occurrences. More precisely, they use historical information about the arrival time of received heartbeats to estimate the expected arrival times of future heartbeat messages from each TT using the equations described for each algorithm. The selection of these algorithms in our framework is based on their performance in terms of detection time of the TT failures over time. In other terms, we select the algorithm that is able to quickly detect the TT failures when compared to the other algorithms. A formal description of the four algorithms and their equations is presented in Appendix A. The steps followed by our "Dynamic TaskTracker Failure Detection" component to dynamically adjust the sending of heartbeats between the TTs and the JT are presented in Algorithm 2.

Algorithm 2 TaskTracker Failure Detection Algorithm

```
 1: HB-data = Collect-data(TT, heartbeats)
 2: /* Apply the algorithms to control the communication between JT-TTs */
 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, TTs, HB-data)
10:     HB-median= Get-Median(TTs, HB-next-arrival)
11:     /* Update the next interval timeout of the following communication */
12:     Update-Communication(JT, TTs, HB-median)
13:     Notify-Scheduler(JT, TTs, HB-median)
14: end while
```

4.3 Scheduling Policies Modelling

To implement the MDP model of our proposed framework, we opt for reinforcement learning techniques [19], to guide the decision making process and evaluate the scheduling decisions in the context of adaptive policy-driven scheduling. We choose to follow a reinforcement learning approach because it allows learning from past experiences (*e.g.*, scheduling policies) to predict potential future actions that guarantee the successful completion of scheduled tasks. Indeed, the reinforcement learning techniques have been applied to solve several problems in the cloud computing system including resource allocation [20], selection of virtual machines [21], job scheduling [22], virtual machines consolidation [23]. They show good performance when applied to such problems similar to the scheduling decisions modeling in the cloud. Furthermore, reinforcement learning techniques allow to consider the dynamic events occurring in the scheduler environment (availability, resources, size of queue, etc.) and adjust the decisions making procedures under uncertainty.

There exist a multitude of reinforcement learning approaches in the open literature. Among them, the TD-learning (Temporal-Difference learning) [9], Q-learning [24], and SARSA (State-Action-Reward-State-Action) [25] are the most used ones. In [26], the authors analyse these algorithms and find that Q-learning and SARSA algorithms outperform the TD-learning algorithm in terms of state exploration (the number of times the system change its state after applying an action). This is because the TD-learning uses only one state network and hence, it cannot easily exploit particular action sequences. Therefore, in our proposed framework, we use a combination of the Q-learning and SARSA algorithms to implement the action selection procedure for the MDP model of the scheduler. For instance, Q-learning is an Off-Policy algorithm that updates a Q-function according to a random policy that maximizes the expected reward. SARSA is an On-Policy algorithm that selects the next state and action according to a random policy and updates the Q-function accordingly. We integrate these two different approaches within the scheduler to evaluate their impact and identify if the algorithm can allow the scheduler to explore more possible policies and to find policies leading to task execution success. The procedural form of SARSA and Q-Learning are presented respectively in Algorithms 2 and 3 from the Appendix B. We describe the approach followed in our framework to obtain better scheduling policies in Algorithm 3.

Algorithm 3 Scheduling Policies Algorithm

```
1:  while (There is a new task to be scheduled ) do
2:      data = Collect-Env(Cluster)
3:      /* Calculate the 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
```

4.4 ATLAS+

We integrate the three different components of the proposed frameworks, described in the previous sections, into the JT through ATLAS+ (an **A**dap**T**ive fai**L**ure-**A**ware **S**cheduler) algorithm described in Algorithm 4. ATLAS+ builds on top of the Hadoop's existing schedulers to provide adaptive scheduling decisions according to events occurring in the cluster.

This algorithm requires firstly to get the status of the Hadoop cluster (including the number of TTs and their status). Next, the Algorithm 2 dynamically updates the communication interval timeout and notifies the JT based on the algorithm giving the best performance results (as described in Algorithm 2). This proactive approach allows ATLAS+ to quickly detect TTs failures. This part of the algorithm runs in parallel with the rest of ATLAS+ algorithm (*i.e.*, as shown by lines 1 to 5 in Algorithm 4).

For a new submitted task, ATLAS+ collects the attributes of the tasks (map/reduce). Using the values of these attributes, the Algorithm 1 predicts whether the submitted task will be finished or failed if executed (line 11 in Algorithm 4). We implement two different prediction algorithms for the mappers and the reducers (since the mappers/reducers have different input parameters). Next, the Algorithm 3 will be executed to get a candidate policy that can be either a *process*, a *reschedule* or a *kill* policy. Specifically, a process policy is a request to the scheduler to execute the submitted tasks on a given TT. Whereas, a reschedule policy is a request to resubmit the task to the queue and wait until its success conditions are met. The success conditions of a task represent the environment circumstances/specifications that eventually lead to a successful execution of that task (*i.e.*, terms to a finish event at the scheduler level). A kill policy is sent to the scheduler to kill an executed or a waiting task.

When the task is predicted to succeed (line 12 in Algorithm 4), ATLAS+ determines a candidate scheduling policy for this task using Algorithm 3 (line 13 in Algorithm 4). When the candidate policy is a process policy (line 14 in Algorithm 4), ATLAS+ checks the availability of the TT and DN to verify if they are activated or not before applying the policy (line 15 in Algorithm 4). Then, ATLAS+ runs the policy (line 18 in Algorithm 4) when the TT and the DN needed to process the task are available (line 16 in Algorithm 4). It saves the outcome result of the processed policy (*e.g.*, finished/failed task, environment status, used resources) in the scheduling policies database (as shown in our proposed framework in Section 2). If the TT and DN are not available, ATLAS+ resubmits the task to the queue and assigns it a penalty (line 21 in Algorithm 4). When the policy is to reschedule (line 24 in Algorithm 4) or to kill (line 26 in Algorithm 4), ATLAS+ runs the policy, assigns a penalty to the task and stores the scheduling policy in the database rules. This penalty reduces their execution priority, causing them to wait in the queue until enough resources are available to enable their speculative execution on multiple nodes.

If the task is predicted to fail (line 29 in Algorithm 4) and the Algorithm 3 selects a process candidate policy (line 31 in Algorithm 4), ATLAS+ will launch the task speculatively on many nodes that have enough resources (line 33 in Algorithm 4), in order to speed up the execution of the task and increase the chances of its success. When the policy is to reschedule (line 35 in Algorithm 4) or to kill (line 37 in Algorithm 4) the task, ATLAS+ runs the policy and saves its outcome in the database rules. All decisions made by the ATLAS+ are controlled by a time-out metric from Hadoop's base scheduler. Hence, if a task reaches its time-out, its associated attempt will be considered as failed and the task will be rescheduled again but with a lower priority.

5 Framework Evaluation

In this section, we present the setup and results of the experiments performed to assess the effectiveness of the proposed framework.

5.1 Experiment Setup

Cluster: We create a 100-node Hadoop 1.2.0-cluster on Amazon EMR; one node is the master, another node is the secondary master to replace the master when it crashes, and 98 slave nodes. The nodes have different characteristics since we select different types of nodes from Hadoop Amazon EMR list. The selected types are *m3.large* (30 nodes), *m4.xlarge* (30 nodes), and *c4.xlarge* (40 nodes) [27], details about their characteristics are listed in Table 1. We select different types of nodes to obtain a heterogeneous set of nodes as in a real world cluster, and to support different workloads. In addition, we vary the number of map and reduce slots (*e.g.*, 100, 150, 200, 1500) for the nodes in order to obtain Hadoop nodes having different capacities and characteristics.

Scheduler: We evaluate the performance of three different types of task scheduling algorithms in Hadoop including First-In-First-Out (FIFO), Fair, and Capacity [28]

Algorithm 4 ATLAS+ Scheduling Algorithm

```
1: while (Cluster is running) do
2:     Cluster-Status = Get-Status-Cluster(Cluster)
3:     /* Adjust the Communication between JT and TTs */
4:     TaskTarcker-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 = Scheduling-Policies-Modelling(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 Send to Queue + Penalty
25:            end if
26:            if (Policy == "Kill") then Kill(Task)
27:            end if
28:        end if
29:        if (Predicted-Status == "FAILURE") then
30:            Policy = Scheduling-Policies-Modelling(Task)
31:            if (Policy == "Process") and (There are Enough Resources on Nodes)
   then
32:                /* Launch Many Speculative Instance of Task */
33:                Execute-Speculatively(Task,N,Policy)
34:            end if
35:            if (Policy == "Reschedule") then Send to Queue + Penalty
36:            end if
37:            if (Policy == "Kill") then Kill(Task)
38:            end if
39:        end if
40:    end while
41: end while
```

TABLE 1: Amazon EC2 Instance Specifications [27]

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

algorithms. In the FIFO algorithm, the tasks are queued and processed in the order in which they are received, regardless of their types and their sizes [28]. The Fair algorithm ensures that the resources in the cluster are fairly distributed across the received tasks so that all users receive the required resources over time [28]. Finally, the Capacity algorithm splits the Hadoop cluster into different queues with different amounts of resources (*i.e.*, CPU, memory). Next, these queues process the received tasks using FIFO scheduling principles [28].

Workload: We run different workload on Amazon EMR Hadoop [27] clusters. To determine the characteristics of the workload to be executed, we collect data about the Hadoop jobs executed on Google cluster [7] and identify their profiles to obtain a representative workload [7]. The running workload include single jobs (*e.g.*, *WordCount*, *TeraGen*, *Sort*, and *TeraSort* [29]), and chained jobs (sequential, parallel, and mix chains) composed of Hadoop single jobs. To obtain different types of workload, we vary the configurations of the running jobs (*e.g.*, size of the job or the chain, number of map and reduce, size of input file). The log data were collected over a fixed period of time of 10 minutes.

Injected Failures: The AnarchyApe tool [30] is used to inject failures to the created cluster at different rates. For instance, we create different scenarios to inject failures to TTs, DNs, network (drop or slowdown), input data (loss of data), tasks and jobs. Specifically, we kill/suspend TTs, DNs; disconnect/slow/drop network; and randomly kill/suspend threads within the TTs in the running executions. To determine the failure rates to be injected to the Hadoop cluster, we use public Google traces [7] to perform a quantitative analysis about the number of failed jobs and tasks, and identify the typical failure rates to be injected to a typical cluster. The Google traces provide information about previously executed tasks and jobs, including Hadoop jobs, in real world Google clusters. The obtained results reveal that the failure of a real world cluster can be as high as 40%, hence, we vary the failure rates in our experiments from 5% to 40% while injecting different types of failures [7].

5.1.1 Task Failure Prediction

We collect logs from the cluster and extract data related to 120,000 jobs and 300,000 tasks. For each job, we extract: job ID, priority, execution time, number of map/reduce, number of local map/reduce tasks, number of finished/failed map/reduce tasks and the final status of the job. For each task, we extract: job ID, task ID, priority, type, execution time, locality, execution type, number of previous finished/failed attempts of the task, number of reschedule events, number of previous finished/failed tasks, number of running/finished/failed tasks running on the TT, the amount of used resources (CPU, Memory and HDFS (Hadoop Distributed File System) Read/Write) and the final status of the task. The predictors or the input of the predictive models are the collected attributes of the tasks. The proposed failure prediction models use the values of these input attributes to determine whether a task will be finished or failed when executed. More details about this step can be found in [8]. The obtained data from logs of the created Hadoop cluster is used to train and test the predictive models to select the model to implement in ATLAS+. So, we split the data into training data and testing data and we evaluate the performance of the selected machine learning models. We perform this step for the map and reduce tasks separately; predicting the scheduling outcome of these tasks for the three studied schedulers (FIFO, Fair and Capacity). Next, a 10-fold random cross validation is applied on the models to determine the model that can identify the scheduling outcome of a task with the best accuracy, precision and execution time. In the cross validation, each data set is randomly split into ten folds. Nine folds are used as the training set, and the remaining fold is used as the testing set. Furthermore, we vary the training rates for each model to evaluate the performance of the models at different training rates and analyze the impact of the training rate on the performance of the predictive models. This is to test how the performance of the models relies on the training rate. The training rates are respectively: 10%, 30%, 50%, 70%, and 90%.

5.1.2 Dynamic TaskTracker Failure Detection

Different types of failures are injected to the TTs (*e.g.*, slowing down/dropping the network, killing/suspending TTs) in order to evaluate the performance of the four proposed algorithms, presented in Section 4.2 and Appendix A, and the basic algorithm used by Hadoop, in terms of detection time (T_D) over time. This is in order to select the appropriate algorithm to implement in ATLAS+. The injected failure rates are 10%, 20%, 30%, 40%, and 50% of TTs. The injection of failures is regular over the intervals of communications (*e.g.*, 2 minutes after the beginning of a new interval). Next, in the following interval, we implement a procedure to revive dead nodes. Also, we vary the time of the recovery of TTs (*e.g.*, 1, 2 and 3 minutes after the beginning of a new interval) to see 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). For example, if the recovery time is one minute, then the TT has more time to send its heartbeat whereas, if the recovery time is within 3 minutes, then there is a shorter time to send the heartbeat. Here, we specify a limit for adjusting the interval of sending heartbeats to 4 minutes, to reduce the overhead of communication between the TTs and the JT.

5.1.3 Scheduling Policies Modelling

We train the SARSA and Q-learning algorithms proposed to solve the MDP model while scheduling around 22,000 tasks (map and reduce tasks). Specifically, we submit 1500 different tasks each 5 minutes to the Hadoop scheduler. So, we integrate each algorithm separately to ATLAS+ and we compute the number of explored policies and the outcome associated with the used policies (either finished or failed task) for each algorithm. Particularly, we measure the policy success rate that can be defined as the ratio between the number of policies leading to a successful event divided by the total of the explored policies in each interval. This experiments are repeated 30 times in order to measure the variance of the two algorithms when integrated with ATLAS+. Here, our aim is to evaluate the performance of the two algorithms over time to compare them. We perform this step in order to select the algorithm that allows our proposed framework to explore more policies and to select the policy that maximizes the number of finished tasks. Next, we implement two procedures; one to collect and store data about the used scheduling policies within the scheduler, another to select the scheduling policies for the scheduler when there is a new task to be scheduled.

For the scheduling policies, we characterize each policy by the following metrics: policy ID, locality/execution Type (local or non-local), time to find the policy (time to access the database and find the policy), selected TT, 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, straggling and running tasks), available slots on selected TT, requested slots on selected TT, used slots on selected TT, 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). These metrics are selected because they capture the characteristics of the environment where the scheduling policies are applied. In addition, we perform a multi-collinearity analysis to identify correlated metrics. More specifically, we compute the *Variance Inflation Factor (VIF)* of the metrics and use a threshold value of 5 to decide whether the metrics are correlated or not. Metrics having a VIF value greater than 5 are considered to be correlated. To evaluate the importance of the metrics of the scheduling policies, we apply the *MeanDecreaseGini* criteria, selecting metrics with higher values, since they represent the most important ones. Next, the second procedure selects the scheduling policy having the highest probability of success (*e.g.*, having the greatest value of positive usage) based on the characteristics of the workload running on the system.

5.2 Evaluation Results

5.2.1 Task Failure Prediction

When analysing the correlation between task attributes and the scheduling outcome of map and reduce tasks, we find that there is a strong correlation between the number of running/finished/failed tasks on a TT, the locality of the tasks, the number of previous finished/failed attempts of a task, and the scheduling outcome of the task. In other terms, tasks characterized by multiple failure events on its environment (including multiple past failed previous attempts and many concurrent tasks (running on the same TT) that experienced multiple failed attempts) have a high probability to fail in the future. Table 2 presents the performance of our studied predictive models. In general, Random Forest outperforms the other predictive models and achieves the best results in terms of precision, recall, accuracy, error, and execution time for the three studied schedulers. This is because the Random-Forest algorithm uses the majority voting on decision trees to generate results which makes it robust to noise, resulting usually in highly accurate predictions. Here, we discuss only the results of the Fair scheduler, because we find that the three schedulers performance follow the same trend. For map tasks, the Random Forest model can achieve an accuracy up to 88.5%, a precision up to 87.6%, a recall up to 93.4%, an error rate of to 25.1%, and an execution time of 29.33 ms. For reduce tasks, the Random Forest model achieves an accuracy up to 94.5%, a precision up to 97.4%, a recall up to 96.5% and an error up to 15.4%. The total execution time of the evaluation of Random Forest for reduce tasks is 38.41 ms. We also analyse the results of the different models under different training rates in terms of accuracy, precision, and recall. Here, we find that the performance of the FIFO, Fair, and Capacity schedulers is following the same trend and hence, we only discuss the results of only one scheduler: Fair scheduler. We can report that the Random Forest has the highest values for the accuracy, precision, and recall when compared to the other algorithms for the map and reduce tasks. Furthermore, 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. For the reduce tasks, the accuracy, precision, and the recall values are 93.4%, 97.8%, and 93.9%, respectively, when trained with 90%. Consequently, we can claim that the Random Forest is highly dependent on the training rate and can achieve better results when the training rate is large. In light of these results, we select Random Forest for the implementation of the ATLAS+ scheduler (at line 13 of Algorithm 1) and retrain its model to collect data each 10 minutes.

TABLE 2: Accuracy, Precision, Recall, Error (%) and Execution Time(ms) for different Algorithms

	Algorithm	**Acc.**	**Pre.**	**Rec.**	**Err.**	**Time**
Map Task	**Tree**	68.6	75.8	63.4	9.14	10.02
	Boost	67.3	84.2	69.7	34.9	201.4
	Glm	65.6	89.5	65.4	39.9	13.54
	CTree	69.4	84.4	68.3	32.6	17.34
	Random Forest	**79.9**	**81.8**	**93.5**	**23.6**	**23.9**
	Neural Network	64.8	86.3	74.1	31.3	63.61
	Algorithm	**Acc.**	**Pre.**	**Rec.**	**Err.**	**Time**
Reduce Task	**Tree**	74.5	85.4	74.0	29.8	15.23
	Boost	84.4	81.7	74.7	10.9	268.77
	Glm	77.2	94.3	71.3	25.3	19.19
	CTree	82.4	88.4	79.4	25.4	20.52
	Random Forest	**94.12**	**92.3**	**96.5**	**15.4**	**29.77**
	Neural Network	84.3	85.4	75.6	19.6	98.14

5.2.2 Dynamic TaskTracker Failure Detection

Figures 3 and 4 present the performance of the five algorithms used to detect the failures of TT nodes. Specifically, they represent the variation of detection time of these algorithms over time when the same number of failures are injected. We find that the performance of the five algorithms under the different failure rates is following the same trend. Hence, we only present here the results of 30%, and 50% TT failures. The obtained results show that the *SFD* algorithm is characterized by a smaller detection delay over time when compared to the other algorithms for the same number of injected failures. The *Bertier-FD* and the *ϕ-FD* do not give good results as their detection times under different failure rates are close to that of the basic Hadoop algorithm, which is giving the worst performance (8 minutes as detection time). This can be explained by the fact that the *Bertier-FD* and the *ϕ-FD* rely on historical information to identify good failure predictors. For instance, the *ϕ-FD* requires a large window size to obtain more data for the normal distribution function and hence computes a more adaptive normal distribution function. For the *Bertier-FD*, it does not have dynamic parameters to tune, which is why the window size does not affect the behavior of the algorithm over time. For the *Chen-FD*, its performance is close to that of *SFD*. This is because they use the same function to estimate the expected arrival time of the next heartbeat as explained in Appendix A. The main difference between the two is how they update the safety margin. For the *SFD*, it uses an adaptive function to update it according to the occurrence of failures in the cluster. However, the *Chen-FD* uses a constant safety margin. Therefore, the SFD can find the value for sending the heartbeats in less time compared to the *Chen-FD*.

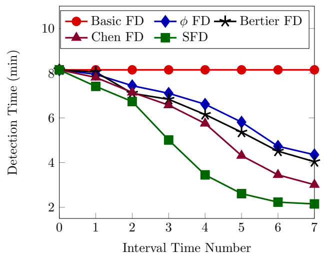


Fig. 3: T_D (30% Failure)

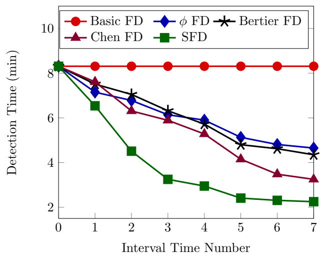


Fig. 4: T_D (50% Failure)

Table 3 presents the results of the failure detection algorithms in terms of mistake rate, for different recovery times of TTs (1, 2, and 3 minutes) and different failure rates (from 10% to 50%). Here, we notice that the performance of the algorithms are following the same trend for the three different recovery times. Hence, we only present the results for a recovery time of 2 minutes. We observe that the *SFD* algorithm is making more mistakes over time. Whereas, the *Bertier-FD* and the *ϕ-FD* makes less errors when identifying failures of nodes. This is because their interval timeout is longer than that of the SFD algorithm. In other words, there is a compromise between the mistake rate and the detection time. The longer is the detection time, the less would be the number of mistakes and vice versa. For instance, the Basic FD is characterized by a constant detection time (8 minutes according to Figures 3), when injecting 40% TT failures and the recovery time is 2 minutes. Overall, it is characterized by a normalized values of wrong failure detection equal to 0.48 (see Table 3). While the SFD is characterized by a decreasing detection time over time (which reaches 2 minutes: see Figures 3) and by a normalized value of wrong failure detection equal to 0.56 (see Table 3). These observations are valid for other algorithms, failure rates, and recovery times. In this context, we should mention also that some heartbeats are lost when sent to the JT. This is due to network conditions and not because of a failure of a TT. For ATLAS+, we select the following algorithms: ϕ-FD and

SFD and integrate them within our proposed scheduler. This is to evaluate the scheduler performance under shorter detection times and lower mistake rates.

TABLE 3: 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

5.2.3 *Scheduling Policies Modelling*

Figures 5 and 6 present the cumulative performance of the two algorithms Q-learning and SARSA in terms of number of explored policies and policy success rate for 30 experiments such that the confidence level is 95%. We observe that the SARSA algorithm shows better results on the generation of policies. It can explore more scheduling policies than the Q-learning algorithm. For example, it can explore 2896 policies while the Q-learning selects only 1345 in 20 minutes (see Figure 5). This is expected since the SARSA algorithm uses a random policy to select the next state and action, which allows it to select more policies. However, we find that Q-learning achieves a success rate of 0.67% while the SARSA only achieves a success rate of 0.52% . This is because the Q-learning algorithm has a function to select the next possible action that maximizes the reward of the next action for each policy (as explained in Appendix B). Furthermore, we notice that after 30 minutes, the two algorithms have almost the same performance. This is because SARSA and Q-learning explored most of the possible policies for the submitted tasks, and the scheduler is mostly using the previously generated scheduling policies.

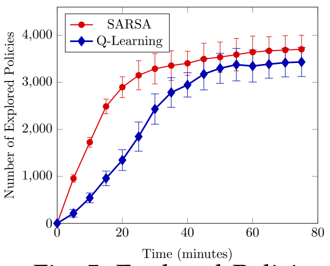


Fig. 5: Explored Policies

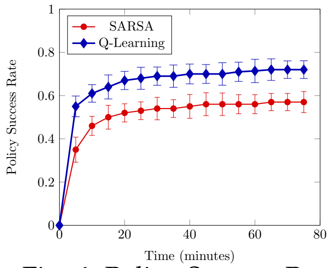


Fig. 6: Policy Success Rate

We analyse the relation between the policy attributes and their outcomes using the VIF, and obtain a strong correlation between the policy reward (0.45), selected TT (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)and the policy outcome. The other attributes have a VIF greater than 5 and hence, we do not consider them in the analysis. Next, we measure the importance of the obtained attributes using the Random Forest model. According to the "MeanDecreaseGini" score, the most important attributes affecting the policy outcome (success or failure) are ordered as follows: load, available/selected slots on selected TT, policy Q-Value, frequency of policy positive/negative usage, policy reward, locality/execution Type, and number of tasks in queue.

Given these results, we decide to train our scheduler using these selected attributes and the SARSA algorithm, at the beginning (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). For ATLAS+, we run the SARSA algorithm for a given interval of time; 30 minutes (since the two algorithms have the same trend starting at 30 minutes as shown in Figure 5) and then switch to the Q-learning algorithm (at lines 10 and 30 of Algorithm 4).

5.2.4 *ATLAS+*

In the sequel, we first discuss the performance of our proposed scheduler ATLAS+. Next, we evaluate the scalability of our proposed framework.

5.2.4.1 Performance Analysis: We compare the performance of our proposed scheduler ATLAS+ when integrated respectively with the FIFO, the Fair, and the Capacity schedulers. All comparisons are done using the exact same jobs, tasks and data. Specifically, we run 2000 Hadoop jobs (10% single jobs, 30% sequential chains, 30% parallel chains, and 30% mix chains), and around 50,000 map/reduce tasks. The performance of each Hadoop's scheduler is measured using the total execution times of jobs, the amount of used resources (CPU, memory, HDFS Read/Write), the numbers of finished and failed tasks and jobs. We calculate the upper and lower bounds of these values, obtained for 30 runs, with a confidence level of 95%. We implement ATLAS+, for these evaluations, using: (1) the Random Forest algorithm, (2) our proposed MDP-model, (3) and an algorithm to control the communication between the TTs and the JT (the ϕ-FD and the SFD algorithms). In summary, we consider the following configurations of our ATLAS+ scheduler: (1) ATLAS+ with MDP, (2) ATLAS+ with MDP and ϕ-FD, and (3) ATLAS+ with MDP and SFD, such that ATLAS+ is built on top of these three existing schedulers.

Figures 7, 9, and 11 present, respectively, the number of finished jobs, map, and reduce tasks, with a confidence level of 95%, for the three schedulers together (as shown in the x-axis: FIFO, Fair, and Capacity). Overall, we observe that ATLAS+MDP increases the number of finished jobs, map, and reduce tasks when compared to the FIFO, Fair, and Capacity schedulers. These results are expected since the early identification of failures allows ATLAS+ to quickly reschedule the potential failed tasks accordingly. Also, we notice that the numbers of finished jobs and tasks (map and reduce) are higher for the ATLAS+MDP+SFD-based algorithm in comparison to the ATLAS+MDP+ϕFD-based algorithm, ATLAS+MDP-based algorithm, and the basic algorithms for the FIFO, Fair, and Capacity schedulers. Hence, despite the fact that it can make more wrong TT failure detections, the SFD-based algorithm can quickly detect the failures of the TTs, in comparison to the ϕ-FD, and dynamically adjusts the interval to detect TTs' failures. The FIFO and Fair schedulers show good performance when compared to the Capacity scheduler because the Capacity scheduler forces the killing of tasks that consume large amounts of memory. Moreover, we observe that the number of finished jobs is lower than the improvement observed on the number of finished tasks. This can be explained by the fact that a single task failure can cause the failure of the whole job. Overall, the number of finished tasks is improved by up to 54% when using ATLAS+ instead of the Fair scheduler (see ATLAS+MDP+SFD-Fair in Figure 11), and the number of finished jobs increased by 41% when using ATLAS+ instead of the Fair scheduler (see ATLAS+MDP+SFD-Fair in Figure 7). Overall, we find that ATLAS+MDP+SFD-based algorithm (particularly, when integrated with the Fair scheduler) is the "winner" compared to the other ATLAS+ implementations.

Figures 8, 10, and 12 present, respectively, the number of failed jobs, map, and reduce tasks with a confidence level of 95%, for the three schedulers together (as shown in the x-axis: FIFO, Fair, and Capacity). The number of failed tasks is decreased by up to 59% (see ATLAS+MDP+SFD-Capacity in Figure 12) and the number of failed jobs is decreased by up to 43% (see ATLAS+MDP+SFD-Capacity in Figure 8). Moreover, we notice that ATLAS+ can reschedule the reduce tasks more efficiently since most of their failures are caused by the failure of their corresponding map task. ATLAS+ is able to achieve better scheduling decisions thanks to the shared failure information in the cluster. This is expected because ATLAS+ can quickly detect the failures of the TTs and update the list of dead nodes, so that the scheduler does not assign new tasks to them. Moreover, ATLAS+ enables the successful processing of single and chained jobs because of the dependency between the jobs within the chained jobs. Specifically, we observe that the number of successful single jobs is higher than the successful chained jobs because of the dependency between the jobs composing these chains. The obtained results show that the number of failed jobs is reduced by up to 43% and that the failure rates of tasks (map and reduce) are also reduced by up to 59% for FIFO, Fair, and Capacity schedulers, respectively. In addition, the ATLAS+MDP+SFD-based algorithm (particularly, when integrated with the Capacity scheduler) outperforms the other implementations of the scheduler and achieves the best performance.

The execution time of ATLAS+ is lower compared to other existing schedulers. ATLAS+ can reduce the number of failed attempts of map/reduce tasks and consequently it can reduce their execution times. Figures 13 and 14 present the execution times of the jobs, tasks (map and reduce), respectively. We observe that the execution times of tasks are decreased on average by 3 minutes (see ATLAS+MDP+SFD-Capacity in Figure 14). Consequently, the total execution time of jobs is decreased on average by 10 minutes, representing a 40% reduction on the total execution time of these jobs (see ATLAS+MDP+SFD-Capacity in Figure 13). We also observe that the execution times of long running jobs are reduced from 30-40 minutes to less than 20 minutes, which represents approximately a 50% reduction. Furthermore, the reduction in the number of failures compensates largely the time spent on the training phase of the predictive algorithm and on adjusting the communication between the JT and TTs. In general, our proposed scheduler ATLAS+ can reduce the overall execution times of tasks and jobs in Hadoop.

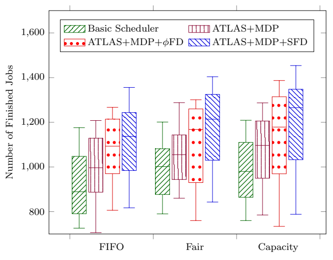


Fig. 7: Finished Jobs

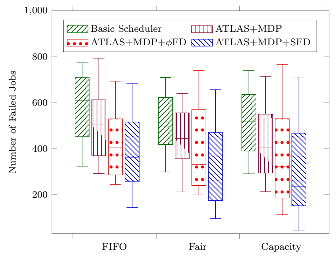


Fig. 8: Failed Jobs

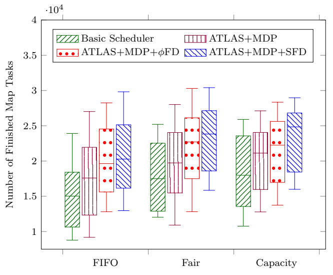


Fig. 9: Finished Map

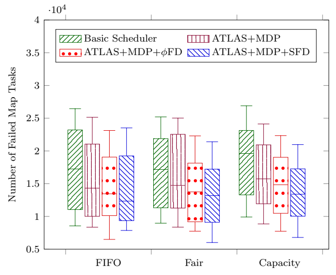


Fig. 10: Failed Map

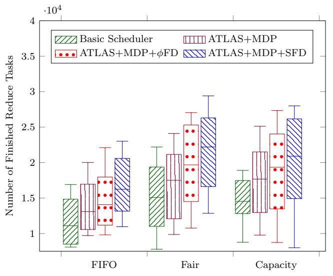


Fig. 11: Finished Reduce

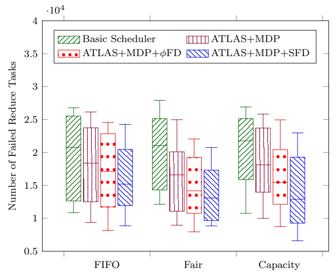


Fig. 12: Failed Reduce

By early identifying the failure of tasks and rescheduling them, ATLAS+ is able to improve the resource utilisation of the cluster. This is expected since the amount of resources that would have been assigned to failed tasks is reduced along with the number of failed tasks. The results presented in Table 4 confirm this anticipated outcome. Table 4 presents the results of the Fair scheduler, which are similair to those for the FIFO and Capacity schedulers. Overall, the jobs and tasks executed using ATLAS+ policies consume less resources than those executed using the FIFO, Fair, or Capacity schedulers (in terms of CPU (22%), memory (20%), and disk (29%)).

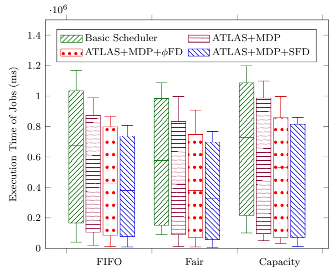


Fig. 13: Exec. Time Jobs

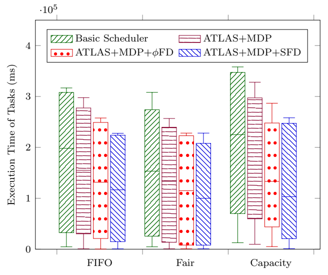


Fig. 14: Exec. Time Tasks

TABLE 4: Resources Utilisation of Hadoop Schedulers

Job/ Task	Scheduler	Fair			
		Basic	ATLAS MDP	ATLAS MDP ϕ-FD	ATLAS MDP SFD
	Resource	Avg.	Avg.	Avg.	Avg.
Job	CPU (ms)	14251	12783	11370	11150
	Memory (10^5 bytes)	9458	8766	8042	7647
	HDFS Read (10^3 bytes)	10568	8615	8339	8257
	HDFS Write (10^3 bytes)	9943	7453	7124	7066
Task	CPU (ms)	4730	4672	4513	4313
	Memory (10^5 bytes)	3007	2955	2912	2496
	HDFS Read (10^3 bytes)	1954	1834	1809	1783
	HDFS Write (10^3 bytes)	1963	1893	1811	1776

Figure 15 and 16 present the number of failed jobs, and tasks (map and reduce) of the four implementations of ATLAS+ after an execution period of 3 days (120,000,000 jobs and 350,000,000 tasks). Here, we observe that the ATLAS+MDP+SFD based algorithm outperforms the other three implementations of ATLAS+, and it is able to reduce the job failures rate by up to 56.33% for the three schedulers. Furthermore, the obtained results show that ATLAS+MDP+SFD based algorithm can reduce the failures rate by up to 60.21% for the three schedulers. In addition, it can reduce the execution time of the running jobs and tasks (particularly the long running-execution jobs) and improve the resources utilisation. These findings can be explained by the fact that the learning time has an impact on the performance of ATLAS+. Indeed, the more data the scheduler collects, the better the scheduling decisions would be, because it allows the scheduler to learn from its previous decisions. Furthermore, the MDP-based model learns new policies and obtains more knowledge about when and where to apply them.

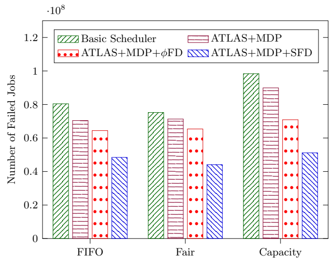


Fig. 15: Failed Job [3 Day]

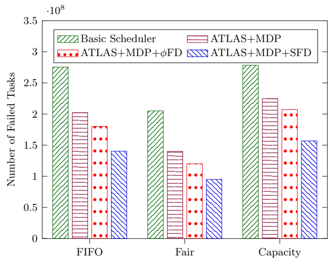


Fig. 16: Failed Task [3 Day]

In addition, ATLAS+ learns new strategies to allocate the resources among the scheduled tasks in order to improve their utilisation and hence, reduce the total execution time of the jobs and tasks. However, we notice that ATLAS+ requires time to access the scheduling rules database and select the appropriate decisions to apply. This is because of the size of the database, the more it generates scheduling rules the longer would be their selection process. We solve this issue by sorting the scheduling decisions by the frequency of usage and the scheduling outcome (finished or failed). We find that this approach can reduce the selection time but, it penalizes some policies since they are not on top of scheduling policies database. On the other hand, we affirm that ATLAS+ is able to identify and catch more failures of tasks and TTs within Hadoop based on the shared failure information, in a comparison with ATLAS [8]. This is because it integrates new strategies to better schedule tasks when there is network congestion, overloaded TTs, straggling tasks, etc. Table 5 presents the benefits of each component in our proposed framework and a comparison between ATLAS and ATLAS+. Overall, we find that ATLAS is able to early identify the failures of tasks within Hadoop by up to 26%. Whereas, ATLAS+ was able catch more failures and reduce the failures rate by up to 33%.

TABLE 5: Comparison ATLAS vs. ATLAS+ (%)

		Failure Rate	Execution Time	CPU Usage	Memory Usage
ATLAS	Task Failure Prediction	26	17	16	14
ATLAS+	Scheduling Policies Modelling	19	14	9	8
	TaskTracker Failure Detection	14	8	7	5

5.2.4.2 Scalability Analysis: To evaluate the scalability of our proposed framework, we assess the performance of ATLAS+ when executing a large workload on a larger cluster. To do so, we performed new experiments on a Hadoop cluster composed of 1,000 nodes using different workloads. More precisely, we performed 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 tasks, respectively. In addition, we used different sizes of tasks (identified as: small/medium/large tasks). We varied the failures rates in our experiments from 5% to 40% while injecting different types of failures. We repeated the experiments 30 times and measured the median values.

We first measured the overhead generated by ATLAS+ by calculating the Worst Case Execution Time (WCET) of each of our proposed algorithms given the running workload. The obtained results showed that when the size of the Hadoop cluster and a number of scheduled jobs/tasks increase, the overhead associated with ATLAS+ increases as presented in Table 6. For Algorithms 1 and 3, the WCET can reach up to 117 and 183 seconds, respectively. While, it can reach up to 258 seconds for

TABLE 6: Worst-Case Execution Time (Seconds) of Algorithms 1, 2 and 3

Number of Jobs	30,000 Jobs (750,000 Tasks)			60,000 Jobs (900,000 Tasks)			90,000 Jobs (2,250,000 Tasks)		
Type of Task	Small	Medium	Large	Small	Medium	Large	Small	Medium	Large
Algorithm 1	27	63	98	29	74	103	35	96	117
Algorithm 2	211			234			258		
Algorithm 3	45	105	143	53	127	159	71	148	183

TABLE 7: Reduction Rates (%) of Algorithms 1, 2 and 3

Number of Jobs	30,000 Jobs (750,000 Tasks)				60,000 Jobs (900,000 Tasks)				90,000 Jobs (2,250,000 Tasks)			
Reduction Rate (%)	Tasks' Failures	Execution Time	CPU Usage	Memory Usage	Tasks' Failures	Execution Time	CPU Usage	Memory Usage	Tasks' Failures	Execution Time	CPU Usage	Memory Usage
Algorithm 1	32	19	18	19	36	21	19	16	35	23	20	20
Algorithm 2	11	15	10	13	17	17	14	10	11	17	15	14
Algorithm 3	24	21	19	12	21	19	18	13	25	19	18	14

Algorithm 2. This result was expected due to the higher number of running jobs/tasks and nodes. We can explain this result by the fact that ATLAS+ requires more time to collect more data about the running tasks, the scheduler environment, received heartbeats from 1,000 nodes, etc., to generate its scheduling strategies.

Second, we evaluate the performance of ATLAS+, in the newly created cluster, in terms of failures rates, execution time, and resources usage. Although ATLAS+ is characterized by an added overhead, we found out that it could improve the performance of the existing Hadoop schedulers. Concretely, we found out that the generated overhead due to the training time of the prediction model, the scheduling policy calculation time and the time spent to adjust the sending of heartbeats messages, is largely compensated by the time saved on the failed tasks that would have been executed otherwise. To better explain these findings, Table 7 presents the obtained results of the three proposed algorithms in ATLAS+ in terms of reduction rates of the number of tasks' failures, execution time, CPU and memory usage for the different workloads (30,000, 60,000, and 90,000 jobs). Here, we discuss only the results of the Fair scheduler, because we observed that the three schedulers performance follow the same trend. At this level, we can report that ATLAS+ could identify up to 74% tasks' failures and reschedule these tasks accordingly. It was also able to reduce the execution times of tasks by up to 59%. Furthermore, it could improve the resources utilization by reducing the amount of used CPU and memory by up to 53% and 48%, respectively. We should also mention that Algorithm 2 was able to early catch up to 58% of the failures of TaskTrackers in the new created Hadoop cluster. Upon these failures' detections, Algorithm 2 could help ATLAS+ better assign tasks to alive nodes and avoid poor scheduling decisions leading to tasks' failures.

In light of these results, we can confirm that the sizes of the cluster and workloads have a direct impact on ATLAS+ performance. Indeed, the more data the scheduler collects from its environment (nodes, tasks, failures, etc), the better scheduling decisions would be. In other words, this would allow the scheduler to learn more about the failures and obtain more knowledge about how to avoid them.

6 Threats to Validity

6.1 Construct Validity

Construct validity threats is about analysing the relation between theory and observation. Our proposed algorithm for ATLAS+ considers that tasks characteristics are the main factors that impact the scheduling outcome of a task, this may not be the case. Particularly, the resource allocation strategy can affect the scheduling decision. But, while building ATLAS+, we find a low correlation between the amount of allocated resource and the scheduling outcome of a task. So, the resource allocation is more likely to affect resource usages than scheduling outcomes. Nevertheless, ATLAS+ may identify task failures that are due to shortage of resources using data from its environment by collecting information about the available resources on the TTs to reschedule tasks on under-loaded nodes. This will allow the scheduler to make scheduling decisions based on usage characteristics. Hence, it can offer better resource utilization and provide improvement in job running time for ATLAS+.

6.2 Internal Validity

Internal validity threats concern the techniques and tools used to build and evaluate our proposed solution. For instance, we adapt and apply four existing algorithms that are used to adjust the sending of heartbeats between the master and the workers of network applications in the cloud. The drawback of doing this is that if the communication interval is small (*e.g.*, 2 minutes), the TTs can send many heartbeats to the JT resulting in too frequent messages exchanges and an overhead on the JT. Furthermore, ATLAS+ can consider alive nodes as dead because of receiving their messages after the expiry interval (due to the small time interval). In addition, we use data from Google clusters [7] to specify the amount of injected failures (up to 40%). Also, we use Amazon EMR to create the Hadoop nodes, which is a real world environment, where other failures can occur. However, it is possible that such Hadoop clusters do not face this failure rate. Therefore, we perform more experiments to demonstrate the benefits of our framework under low failure rates of 1-2%. The obtained results show that ATLAS+ can reduce the number of failed jobs by up to 9% and the number of failed tasks by up to 12%. These results confirm our claim that the more data the scheduler collects about failures from its environment, the better the scheduling decisions would be, because it allows the scheduler to learn from its previous decisions. On the other hand, the injected failure cases may not represent real failure scenarios, which can affect the performance of the proposed scheduler. Therefore, it is very important to validate our case study with a more diverse set of Hadoop clusters and different failure rates.

6.3 Conclusion Validity

Conclusion validity threats is about analysing the relation between the treatment and the outcome. The main goal of ATLAS+ is to provide better scheduling decisions with a minimal impact on the execution time of the received tasks. Although we integrate new procedures to current Hadoop schedulers, we have verified that they do not introduce a large overhead. In addition to the performed experiments to evaluate the scalability of ATLAS+ (Section 5.2.4.2), we measured the Worst Case Execution Time (WCET) of our proposed algorithms for different other scenarios (small/medium/large tasks and cluster). Specifically, we performed experiments to execute 10,000 tasks using ATLAS+ to measure the added overhead for each single task separately. We repeated the experiments 100 times and measured the median values. In the following, we present the performance results of the Fair scheduler that achieves similar results to the ones of the FIFO and Capacity scheduler.

For Algorithm 2, we measure the WCET for three types of clusters: 10-nodes, 50-nodes, and 100-nodes Hadoop cluster. The obtained WCET values of Algorithm 2 are 54, 132 and 216 seconds for 10-nodes, 50-nodes, and 100-nodes cluster, respectively. At this level, we should mention that all steps of Algorithm 2 are off the critical path of the scheduler. This is because they are used to collect data about the received heartbeats and to adjust the expiry interval timeout accordingly. Hence, the integration of Algorithm 2 within Hadoop does not impact the execution time of the scheduled tasks; it only impacts the communication time between the JT and TTs.

For Algorithms 1 and 3, the obtained results when executing three different types of tasks including "small, medium, and large" tasks can be summarized in Table 8. For Algorithm 1, the steps from line 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 line 13 to 17 in Algorithm 1. We measured the complexity of these steps in terms of time, and found that it can reach 73 seconds. For Algorithm 3, all the steps are on the critical path of the scheduler. We measured the complexity of this algorithm in terms of time, and found that it can reach 158 seconds. Overall, we can claim that by reducing the number of failed tasks and the overall resources utilisation ATLAS+ was able to compensate the added overhead of its different components.

TABLE 8: Worst-Case Execution Time of Algorithms 1 and 3

	Worst-Case Execution Time (Seconds)		
	Small Tasks	**Medium Tasks**	**Large Tasks**
Algorithm 1	20	58	73
Algorithm 3	57	104	158

6.4 Reliability Validity

Reliability validity threats are related to the replication of our study on other platforms. The proposed framework can be integrated with other cloud platforms like Microsoft Azure, or Google platform. To do so, it requires to collect logs from these platforms, build the statistical predictive models and validate them, and finally adjust the proposed MDP-model and its corresponding reinforcement learning algorithms. Then, the proposed framework can be integrated and built on top of any cloud scheduler to reduce task failure rates and provide better resources utilisation and execution time.

6.5 External Validity

External validity threats concern the generalization of our results. Our case study is performed on a 100-nodes Hadoop cluster running on Amazon EMR. Further studies can be done to validate the results of ATLAS+ on a larger scale. In addition, it is necessary to use different failure cases and rates, to validate the failure detection mechanism on ATLAS+. To generalize these findings, we plan to extend and evaluate our proposed framework on Spark [31], a novel in-memory computing framework for Hadoop. Specifically, we will extend Spark using the three components of the proposed framework described in Section 3 and evaluate the performance of the used algorithms on Hadoop to integrate them within Spark.

7 Related Work

Many approaches have been proposed to improve scheduling decisions in Hadoop. We discuss the most relevant to our work in the following:

7.1 Fault-Tolerance Mechanisms in Hadoop

Hadoop tracks the processing of the received tasks using the JT which will re-schedule map and reduce tasks on other nodes in case of a failure. Despite the fact that this solution is simple and guarantees the successful processing of failed tasks, it is not always effective and comes with additional costs (*e.g.*, resources usage, extra delays in execution time). For instance, the JT has to reschedule all tasks belonging to the failed jobs including the finished tasks despite their successful completion. To alleviate this issue, some studies have proposed fault-tolerant mechanisms for Hadoop. Dinu *et al.* [5], who analyse the performance of Hadoop under failure, claim that many failures occur in Hadoop due to the lack of sharing failure information (*e.g.*, straggling tasks, TT failure, etc). Therefore, they design RCMP [32] , as a first order failure resilience strategy, that allows for efficient job recomputation upon failure by recomputing the necessary tasks rather than data replication. But, RCMP is only valid for I/O intensive jobs, which makes it not valid for all types of Mapreduce workload (*e.g.*, CPU intensive jobs). Hao *et al.* [33] implement an adaptive module to track the heartbeat-based communication between the TTs and the JT. This module can adjust the *expiry interval* for the JT to detect whether a TT is considered as dead or not, according to various job sizes in a Hadoop cluster. Moreover, they develop a reputation-based detector to decide whether a worker is failed or not; when its reputation is lower than a specified threshold. This approach can help detect failures of TT early, and reduce the total execution time of jobs. RAFT [34], a Recovery Algorithm for Fast-Tracking in MapReduce, is proposed by Quiane-Ruiz *et al.* to track tasks at different checkpoints. The checkpoints are responsible for storing the execution status of tasks. When a task encounters a failure, the JT will re-schedule the task from the last available checkpoint. RAFT does not re-execute the finished tasks belonging to the failed jobs and only failed tasks will be re-executed; which would reduce some additional costs (*e.g.*, reduce the total execution time by 23%). Yildiz *et al.* [35] propose Chronos, a failure-aware scheduling strategy that enables an early action to recover the failed tasks in Hadoop. Chronos is characterized by a pre-emption technique to carefully allocate resources to the recovered tasks. It could reduce the job completion times by up to 55%. However, it is still relying on wait and kill pre-emptive strategies, which could lead to resource wastage and degrade the performance of Hadoop clusters. Our work is different in early detecting the task failure before its occurrence and an early recovery action, which allow to avoid resource wastage compared to Chronos.

7.2 Adaptive Scheduling in Hadoop

There are also related work on adaptive scheduling in Hadoop. LATE [36] is proposed to prioritize tasks waiting in the queue based on collected information about running tasks and their progress. LATE can improve the scheduling decisions by considering the progress rate of the running tasks and the availability of resources in the cluster; which could reduce the total execution time by a factor of 2 in Hadoop clusters. Hadoop clusters are a heterogeneous environment, where there are machines with different software and hardware configurations. Quan *et al.* [37] show that these configurations can help improve scheduling decisions in Hadoop. To do that, they propose the SAMR (Self-Adaptive MapReduce scheduling) algorithm, which estimates the progress of tasks based on collected hardware system information. While SAMR could integrate different information about the hardware system, it does not considered other important factors about job characteristics (*e.g.*, the task size, data locality, etc). To overcome these limitations, ESAMR (Enhanced Self-Adaptive MapReduce scheduling) [38] is proposed to take into account new information about straggling tasks, job size, and remaining time. SARS [39] (Self-Adaptive Reduce Start time) was proposed as a scheduling algorithm to decide when to start a reduce task. SARS uses information about the completion time of maps and reduce tasks and the job total completion time to evaluate the impact of different times to start the reduce tasks on the total execution time. SARS could reduce response time on average by 11%.

8 Conclusion

In this paper, we propose a dynamic and failure-aware scheduling framework for Hadoop that can adjust its scheduling strategies based on collected information from the Hadoop cluster. We demonstrate the possibility of predicting potential task failures early, using historical information about events occurring in the cloud. Second, we propose an MDP-based model to guide the scheduler, to make better scheduling decisions. Finally, we propose to use adaptive algorithms to adjust the frequency of communication between nodes in a Hadoop cluster. To show the benefits of our solution, we integrate our framework within Hadoop and build ATLAS+ (An AdapTive faiLure-Aware Scheduler), a new scheduler for Hadoop. To the best of our knowledge, ATLAS+ is the first adaptive scheduler that can early identify failure of tasks and TTs using collected information from cloud environment, and adjust its scheduling decisions on the fly. We implement ATLAS+ in Hadoop and deploy it on a 100-node Hadoop cluster in Amazon Elastic MapReduce (EMR). We compare the performance of ATLAS+ with those of three main Hadoop schedulers. The obtained results show that ATLAS+ outperforms the three common schedulers of Hadoop. It can reduce the number of failed jobs by up to 43% and the number of failed tasks by up to 59%. Also, ATLAS+ can reduce the total execution time of jobs and tasks and reduce CPU and memory usage. As a future work, we plan to extend ATLAS+ using scheduling procedures to optimize the resources allocation across tasks. In addition, we can use unsupervised algorithms to train the prediction algorithm in ATLAS+, and evaluate their impacts on Hadoop scheduler.

References

[1] K. Lee, Y. Lee, H. Choi, Y. Chung, and B. Moon, "Parallel Data Processing with MapReduce: A Survey," *SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[2] S. Kurazumi, T. Tsumura, S. Saito, and H. Matsuo, "Dynamic Processing Slots Scheduling for I/O Intensive Jobs of Hadoop MapReduce," in *International Conference on Networking and Computing*, 2012, pp. 288–292.

[3] T. Jian, M. Shicong, M. Xiaoqiao, and Z. Li, "Improving ReduceTask Data Locality for Sequential MapReduce Jobs," in *IEEE INFOCOM*, 2013, pp. 1627–1635.

[4] L. Jinwei and S. Haiying, "A Low-Cost Multi-Failure Resilient Replication Scheme for High Data Availability in Cloud Storage," in *Proceding of IEEE International Conference on High Performance Computing, Data, and Analytics*, 2016, pp. 1–10.

[5] F. Dinu and N. Eugene, "Understanding the Effects and Implications of Compute Node Related Failures in Hadoop," in *Symposium on High-Performance Parallel and Distributed Computing*, 2012, pp. 187–198.

[6] Y.-P. Kim, C.-H. Hong, and C. Yoo, "Performance Impact of Job-Tracker Failure in Hadoop," *International Journal of Communication Systems*, vol. 28, no. 7, pp. 1265–1281, 2015.

[7] 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*, 2015, pp. 58–65.

[8] ——, "ATLAS: An Adaptive Failure-Aware Scheduler for Hadoop," in *International Performance Computing and Communications Conference*, 2015, pp. 1–8.

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

[10] 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*, 2013, pp. 130–135.

[11] V. D. Valerio and F. L. Presti, "Optimal Virtual Machines Allocation in Mobile Femto-Floud Fomputing: An MDP Fpproach," in *IEEE Wireless Communications and Networking Conference Workshops*, 2014, pp. 7–11.

[12] D. L. et al., "A Spearman Correlation Coefficient Ranking for Matching-score Fusion on Speaker Recognition," in *IEEE Region 10 Conference TENCON*, 2010, pp. 736–741.

[13] The R Project for Statistical Computing. [Online]. Available: http://www.r-project.org/,2017

[14] W. Chen, S. Toueg, and M. K. Aguilera, "On The Quality of Service of Filure Detectors," *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 561–580, 2002.

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

[16] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The phi; Accrual Failure Detector," in *IEEE International Symposium on Reliable Distributed Systems*, 2004, pp. 66–78.

[17] X. Naixue, V. Athanasios, W. Jie, Y. Richard, R. Andy, Z. Yuezhi, S. Wen-Zhan, and P. Yi, "A Self-tuning Failure Detection Scheme for Cloud Computing Service," in *IEEE International Parallel and Distributed Processing Symposium*, 2012, pp. 668–679.

[18] 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*, 2012, pp. 668–679.

[19] N. Mastronarde and M. van der Schaar, "Online Reinforcement Learning for Dynamic Multimedia Systems," *IEEE Transactions on Image Processing*, vol. 19, no. 2, pp. 290–305, 2010.

[20] 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*, 2017, pp. 372–382.

[21] 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*, 2016, pp. 92–97.

[22] 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*, 2016, pp. 142–147.

[23] 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*, 2014, pp. 500–507.

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

[25] R. Jia, B. Xiangping, X. Cheng-Zhong, W. Leyi, and Y. George, "VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration," in *International Conference on Autonomic Computing*, 2009, pp. 137–146.

[26] 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*, 2013, pp. 108–115.

[27] Amazon EC2 Instances. [Online]. Available: http://aws.amazon.com/ec2/instance-types/,2017

[28] Apache Hadoop Documentation. [Online]. Available: http://hadoop.apache.org/,2017

[29] M. K. et al., "Hadoop Performance Modeling for Job Estimation and Resource Provisioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 441–454, 2016.

[30] 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*, 2012, pp. 5:1–5:6.

[31] 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*, pp. 1–17, 2016.

[32] 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*, 2014, pp. 962–971.

[33] H. Zhu and H. Chen, "Adaptive Failure Detection via Heartbeat under Hadoop," in *IEEE Asia-Pacific Services Computing Conference*, 2011, pp. 231–238.

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

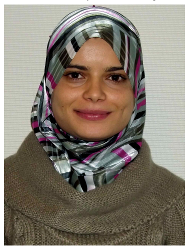
[35] O. Yildiz, S. Ibrahim, and G. Antoniu, "Enabling Fast Failure Recovery in Sshared Hadoop Clusters: Towards Failure-aware Scheduling," *Future Generation Computer Systems*, vol. 74, pp. 208 – 219, 2017.

[36] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *International Conference on Operating Systems Design and Implementation*, 2008, pp. 29–42.

[37] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, "SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment," in *International Conference on Computer and Information Technology*, 2010, pp. 2736–2743.

[38] X. Sun, C. He, and Y. Lu, "ESAMR: An Enhanced Self-Adaptive MapReduce Scheduling Algorithm," in *International Conference on Parallel and Distributed Systems*, 2012, pp. 148–155.

[39] Z. Tang, L. Jiang, J. Zhou, K. Li, and K. Li, "A Self-Adaptive Scheduling Algorithm for Reduce Start Time," *Future Generation Computer Systems*, vol. 4344, no. 10, pp. 51–60, 2015.



Mbarka Soualhia holds an M.Sc degree in Engineering concentration Information Technology from École de Technologie Suprieure (ÉTS),Canada. She is currently a Ph.D candidate at Concordia University, and she is working as research assistant under the supervision of Prof. Sofiène Tahar and Prof. Foutse Khomh. Her research focuses on designing adaptive software components and software architecture in distributed systems and their verification.

Foutse Khomh is an associate professor at the École Polytechnique de Montral, where he heads the SWAT Lab. on software analytics and cloud engineering research. He received a Ph.D in Software Engineering from the University of Montreal in 2010. His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytic.

Sofiène Tahar received the Ph.D. degree with distinction in computer science from the University of Karlsruhe, Germany, in 1994. Currently, he is a professor and the research chair in formal verification of system-on-chip at the Department of Electrical and Computer Engineering, Concordia University. His research interests are in the areas of formal hardware verification, system-on-chip verification, analog and mixed signal circuits verification.