PERFORMANCE GUARANTEES FOR DEADLINE-DRIVEN MAPREDUCE JOBS UNDER FAILURE

BY

FARAZ FAGHRI

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Industrial Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Associate Professor Carolyn Beck
ABSTRACT

Increasingly, large systems and data centers are being built in a ‘scale out’ manner, i.e. using large numbers of commodity hardware components instead of traditional ‘scale up’ using expensive, specialized equipment. However, large numbers of commodity components imply higher rates of failure across such systems. Such failures can cause applications to miss their deadlines for task completion. For this reason, cloud service providers and cloud applications must anticipate failures and engineer their services accordingly.

In this thesis, we first analyze the availability of a commodity data center designed for MapReduce applications. MapReduce is increasingly used in industry for efficient large scale data processing tasks including personal advertising, spam detection, as well as data mining. We show how MapReduce software level fault tolerance can be used to achieve the same availability as scale up data centers. Second, we extend existing job schedulers for deadline-driven jobs to handle machine and software failures and satisfy the service level objectives.
To my family, for their love and support.
ACKNOWLEDGMENTS

This thesis would not have been possible without the support of many people. Many thanks to my adviser, Professor Carolyn Beck, who gave me direction and helped make some sense of the confusion. Also thanks to Professor Roy H. Campbell, for his technical advice and his continuous support for this project. And finally, thanks to my parents and numerous friends for always offering support and love.
CONTENTS

Chapter 1  INTRODUCTION ............................................. 1

Chapter 2  AVAILABILITY OF MAPREDUCE DATA CENTERS .... 3
  2.1 Motivation and Background ...................................... 3
  2.2 Model Description ................................................ 6
  2.3 Results .................................................................. 8
  2.4 Conclusion ............................................................ 9

Chapter 3  FAILURE SCENARIO AS A SERVICE FOR HADOOP
  CLUSTERS ................................................................. 11
  3.1 Motivation ............................................................. 11
  3.2 Related Work ........................................................ 13
  3.3 FSaaS Design ......................................................... 14
  3.4 Evaluation ............................................................ 19
  3.5 Conclusion ............................................................ 24

Chapter 4  PERFORMANCE OF DEADLINE-DRIVEN MAPRE-
  DUCE JOBS ............................................................... 26
  4.1 Motivation ............................................................. 26
  4.2 Review ARIA ........................................................ 28
  4.3 Reactive Latency Masking for a Single MapReduce job .... 29
  4.4 Reactive Latency Masking for a DAG job .................... 33
  4.5 Conclusions and Future Work ..................................... 37

BIBLIOGRAPHY ............................................................. 38
Cloud computing has recently become a mainstream commodity in the IT industry [1]. Computing that was once done locally is now being done in the cloud. But as more and more services are pushed into the cloud, the size of the cloud infrastructures and the data is growing rapidly. Hence the demand for scalable systems capable of accommodating and computing petabytes of data.

One of the major challenges for top cloud providers and data centers is availability. High availability is a major requirement in modern business applications, cloud customers would not move to cloud unless there is a clear analysis of availability in service level objectives (SLO). Accordingly, Highly-available and scalable storage and computation systems are a key component of such environments. Availability analysis of data centers helps engineers to better understand the architecture and configuration of their data centers and allows them to improve their systems. From the cloud customer point of view, availability along the price are two main factors to make a decision of moving to cloud and choosing the right provider.

A complex storage system presents many design challenges in addressing scalability and availability; it becomes important especially in cloud infrastructure where highly-dependable servers are not generally accessible or financially unfeasible.

In recent years new storage and computation solutions has been developed, many of them like MapReduce are designed to be deployed in data centers with commodity hardware. As a result, there has been a growth in number of public and private Hadoop data centers, an open source implementation of MapReduce which allows users to store petabytes of data on commodity servers and provides an easy to use computational platform for big data.

However, as a commodity hardware data centers grow, so does the overall system dependability of equipment failure or software bugs in the data
center. Applications not only need to prepare for occasional failures of the infrastructure, they need to expect different types of failures (ranging from hard drive errors to crash of whole racks) and combinations of them as part of an application’s normal operating procedure [2]. Combinations of failures may have a major impact on performance of an application, sometimes even leading to applications being temporarily out of service [3].

In this thesis, first, in Chapter 2, we analyze the availability of data centers designed for simultaneous storage and computation. These data centers are deployed on commodity hardware and are running MapReduce jobs. We discuss how the availability of these data centers has been increased by software solutions. Next, in Chapter 3, we introduce an assessment and profiling tool for study of failures in Hadoop clusters. And finally, in Chapter 4, we propose a reactive solution to mask the effect of failures for MapReduce jobs to satisfy different service level objectives (SLOs).
Chapter 2

AVAILABILITY OF MAPREDUCE DATA CENTERS

2.1 Motivation and Background

With the increasing volume of data and industry tendency to store and compute the data, new data center designs are more focused on providing financially feasible solutions. Now, companies and individuals are able to use commodity hardware data centers to store and process their data.

Consequences of data centers running on commodity hardware are higher rate of failure and lower rate of availability. However, new software technologies try to increase the availability of data centers through software redundancy rather than premium hardware. As an example of such new technologies, Hadoop provides a higher available data center comparing to commodity data centers which are not running this software.

Apache Hadoop is a framework for running applications on large cluster built of commodity hardware. The Hadoop framework transparently provides applications both reliability and data motion. Hadoop implements a computational paradigm named MapReduce, where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. In addition, it provides a distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both MapReduce and the Hadoop Distributed File System are designed so that node failures are automatically handled by the framework [4].

In Hadoop, hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional. Therefore, detection of
faults and quick, automatic recovery from them is a core architectural goal of HDFS [4].

To reduce the complexity, Hadoop design is being implemented on a generic data center models. In this study we use available designs like Google data center [2]. Hadoop data centers are usually based on a hierarchical data center design, what is different comparing to general designs is the role of machines. In this section, first we briefly review the software level design of a Hadoop clusters, then we go over the data center deployment and role management.

The MapReduce model [5], consists of two computation functions: map and reduce. The map function takes an input key/value pair and produces a list of intermediate key/value pairs. The intermediate values associated with the same key are grouped together and then passed to the reduce function. The reduce function takes intermediate key with a list of values and processes them to form a new list of values.

\[
map(k_1, v_1) \rightarrow list(k_2, v_2) \\
reduce(k_2, list(v_2)) \rightarrow list(v_3)
\]

MapReduce jobs are distributed and executed across multiple machines; the map stage is partitioned into map tasks and the reduce stage is partitioned into reduce tasks. Each map task processes a part of input data: it reads data, applies the user-defined map function on each record, and writes the intermediate data to the local disk for different reduce tasks. The reduce stage consists of two phases: shuffle and reduce phase. In the shuffle phase, intermediate data produced by map tasks will be fetched. All intermediate will be sorted. In case the intermediate data does not fit in memory, in an external merge sort stage, the intermediate data will be shuffled, merged in memory, and written to disk. After all the intermediate data is shuffled, all the sorted files will be merged. Finally, in the reduce phase, the sorted intermediate data is passed to the user-defined reduce function. The output from the reduce function is generally written back to the distributed file system.

In Hadoop implementation, job scheduling is performed by a master node, which manages a number of worker nodes. Master node is also responsible for storing data on Hadoop Distributed File System (HDFS). Based on the
implementation these functionalities might have been decided into Name Node and Job Tracker. Worker Nodes make up the vast majority of machines and do all the work of storing the data and running the computations. Each worker communicates with and receive instructions from their master nodes.

In order to submit Map Reduce jobs describing how that data should be processed, dedicated client machines have Hadoop installed with all the cluster settings. Their role is to load data into the cluster and submit Map Reduce jobs.

Figure 2.1 shows a typical Hadoop cluster. Rack servers are connected to a top of rack switch; the rack switch has uplinks connected to another tier of switches. The majority of the servers are worker nodes with lots of local disk storage and moderate amounts of CPU and Memory.

Hadoop chops a huge chunk of data into small chunks and spread it out over many machines, map and reduce tasks process their portion of the data in parallel.

For fault tolerance, Hadoop, replicates every block of data on multiple machines avoid data loss. Each block will be replicated in the cluster as its loaded. The standard setting for Hadoop is to have 3 copies of each block in the cluster. This can be configured with the dfs.replication parameter in the file hdfs-site.xml. In default implementation, data will be broken into multiple blocks and each block, will be replicated on 3 machines. Hadoop has the concept of “Rack Awareness”; meaning racks and server locations can be manually defined in the system. There are two key reasons for this: Data loss prevention, and network performance. As each block of data will be replicated to multiple machines to prevent the failure of one machine from
losing all copies of data. One replication will be stored in another rack in case that rack experiences a failure such as a switch failure or power failure. Knowing where servers are located in the network topology will help master node to make an intelligent decision about where data replicas should exist in the cluster. There is also an assumption that two machines in the same rack have more bandwidth and lower latency between each other than two machines in two different racks.

### 2.2 Model Description

In this section, we explain the Stochastic Activity Network (SAN) model of a data center at different levels. First we explain the motivation behind reward function then we model a server then a rack and later a data center. It should be noted that failure rates and repair rates have exponential distribution.

#### 2.2.1 Reward Function

We want to measure the availability of the system. So we measure the state of the system to see whether it is in operation or has failed. For instance the reward function for the server considering the working state of processors is as if $\text{processors} - > \text{system.fail} - > \text{Mark()} == 0$ then $\text{reward} + = 1.0$.

To measure the availability, we use this reward function and calculate the mean and variance of $\text{system.fail}$ over certain period of time.

#### 2.2.2 Single Server Model

To decrease the number of generated states, we model a server with components which are more likely to fail and could be replaced easily. This assumption that a server is limited to these components is valid since components that are not modeled are required to be replaced with other components like the main board, therefore we consider the board as the main component.

In table 2.1 the components in the server model with their failure rates and repair rates are mentioned. In this model, at least 15 disks (RAID 5), 2 CPUs, 1 board, 1 controller and 1 power should be operating otherwise the server fails.
Table 2.1: Server components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Units</th>
<th>Failure rate</th>
<th>Repair Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disks</td>
<td>16</td>
<td>1/87600</td>
<td>1/2</td>
</tr>
<tr>
<td>CPU</td>
<td>4</td>
<td>1/131000</td>
<td>1/5</td>
</tr>
<tr>
<td>RAID Controller</td>
<td>2</td>
<td>1/255000</td>
<td>1/8</td>
</tr>
<tr>
<td>Power</td>
<td>2</td>
<td>1/84000</td>
<td>1</td>
</tr>
<tr>
<td>Board</td>
<td>1</td>
<td>1/98000</td>
<td>1/8</td>
</tr>
</tbody>
</table>

Figure 2.2: SAN model for processor.

The SAN model for processors are shown in figure 2.2, and the composed model in 2.3.

2.2.3 Rack Model

For the rack model, in order to decrease the number of states we assume server as one entity with the failure rate close to min failure rate of components. Also, as system scales, repair will take longer time for servers. The other major component in a rack is Top of the Rack (ToR) router which is also included in this model. In table 2.2 the components in the rack model with their failure rates and repair rates are mentioned. Figure 2.4 shows the model for a server in rack and figure 2.5 shows the rack composed model. In this model, at least 8 servers and 1 router should be operating otherwise the rack fails. It should be noted that there is no redundancy for routers in commodity data center designs.

Figure 2.3: Server composed model.
Table 2.2: Rack components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Units</th>
<th>Failure rate</th>
<th>Repair Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>10</td>
<td>1/45753</td>
<td>1/5</td>
</tr>
<tr>
<td>Router</td>
<td>1</td>
<td>1/141000</td>
<td>1/8</td>
</tr>
</tbody>
</table>

Figure 2.4: SAN model for a server in a rack.

2.2.4 Data Center Model

Data center is the most challenging component, because of rack replications we have exponential increase in the number generated states. Therefore, we have only considered a data center with 2 racks which is a smallest possible Hadoop cluster with 3 replication constraint. The composed model is shown in figure 2.6.

2.3 Results

We ran the experiments using TransientSolver for a period of 8760 hours (1 year). Availability of a single server, rack and a data center with 2 racks are computed in table 2.3.

The results show that using commodity hardware over a period of 1 year with periodical repair and maintenance, we expect downtime of 43 minutes for a server, 30 minutes for a rack with 10 server (2 redundant) and 18 minutes for a data center with 2 racks and 4 server redundancy. In this model we assumed that the data center is under periodical repair which is not the case.
in many small data centers when the company contracts another company for longer period of time, like monthly maintenance. In other words, it is true that you can pay a significant premium for hardware with a high mean time to failure (MTTF). However, working with big data requires thousands of disks and servers. Even an MTTF of 4 years would cause about 5 failures per week in a cluster of 1,000 nodes. With less money, using commodity hardware with an MTTF of 2 years, we can expect about 10 failures per week. Which shows they both require same amount of energy to design a fault tolerant system. So it motivates us to provide fault tolerance through software.

Considering the downtime, the challenge for Hadoop or any other data center software platform on top of commodity hardware is how to increase the availability through software. In Hadoop, the fault tolerance mechanism is based on replicating data on different servers. Table 2.4 shows the comparison of commodity data center availability with Hadoop cluster.

### 2.4 Conclusion

In conclusion, industry is moving toward store and process all the data they can get their hands on, which motivates much cheaper and more scalable
Table 2.4: Availability of commodity data center components and Hadoop cluster.

<table>
<thead>
<tr>
<th>Component</th>
<th>Generated States</th>
<th>Mean</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data center (2 rack)</td>
<td>506</td>
<td>0.993266</td>
<td>6.806655E-005</td>
</tr>
<tr>
<td>Hadoop with 3 replication</td>
<td>1108</td>
<td>0.999778</td>
<td>8.174664E-005</td>
</tr>
<tr>
<td>Hadoop with 4 replication</td>
<td>1310</td>
<td>0.999901</td>
<td>5.673209E-005</td>
</tr>
</tbody>
</table>

solutions comparing to premium and reliable data center. Commodity hardware data centers are feasible for anyone in terms of price and scalability, however, **reliability and availability in these data centers must come from software.**

Hadoop fault tolerance is based on making 3 replications of each chunk of data and store them on different servers and racks, a redundancy which is provisioned at the software level. With default replication of 3 we can achieve much higher availability, however, we would dedicate significant disk space for these replications. In Hadoop clusters, there is a trade-off between availability and disk space. User can choose the number of replications according to their requirements; if processing particular data is critical, user should increase the number of replications, hence, allocate extra space to that data.
3.1 Motivation

Cloud computing has recently become a mainstream commodity in the IT industry [1]. Consequently, the impact on overall system dependability of equipment failure or software bugs in the cloud infrastructure has increased. Applications not only need to prepare for occasional failures of their cloud infrastructure, they need to expect different types of failures (ranging from hard drive errors to crash of whole racks) and combinations of them as part of an application’s normal operating procedure [2]. Combinations of failures may have a major impact on performance of an application, sometimes even leading to applications being temporarily out of service [3]. Unfortunately, the use of small sandbox-testing models is not sufficient to predict the effects of failures on realistic applications running on large cloud-based infrastructures.

A few teams have proposed failure injection as a possible way to address the reliability challenges of cloud infrastructures. The first publicized system developed specifically for failure injection in cloud computing infrastructure is the “Chaos Monkey” (ChM) from Netflix [6], which was recently open-sourced (July 2012). In ChM, a simple failure model (namely a whole node failure) is simulated across a virtual cluster running in the Amazon Elastic Compute Cloud (Amazon EC2). The model injects a continuous sequence of failures into the Amazon auto-scaling group service, causing frequent failures of different nodes within the infrastructure in order to identify weaknesses in the applications that run on it.

Later, Gunawi et al. introduced “Failure as a Service” (FaaS) as an approach to the challenge, and proposed a basic design for such a service [7]. Their motivation stemmed from the ChM system, and they intended to build
upon that idea.

Both of the previous efforts simulated certain failure modes, for example, crashing of a node. However, real-world faults and errors result in many diverse failure scenarios, potentially including combinations of different failure modes [8]. In addition, many failures in a cloud environment tend to cascade from being small problems to being large problems very quickly [3]. Therefore, we believe that a system designed to assess the real-world dependability of a cloud-based system is needed, and that a failure system should be able to handle complex fault and error injection scenarios and also simulate combinations of different failure modes.

In this paper, we introduce a Failure Scenario as a Service (FSaaS) model, designed and implemented for Hadoop clusters. Our FSaaS model can currently be used by cloud service providers and clients who rely on Hadoop MapReduce clusters. By targeting Hadoop, we aim to provide FSaaS services to a wide spectrum of users. Consider that an average of one thousand MapReduce jobs were executed on Google’s clusters every day in 2008, and that more than ten thousand distinct MapReduce programs were implemented internally at Google in a four-year period [5]. There exist many types of Hadoop workloads [9], and this paper shows that for the workloads we study, they behave very differently under various failure scenarios. Because it is difficult to design general failure scenarios that fit all types of Hadoop workloads, we profile the behavior of several different Hadoop workload types against failures and generate a series of template failure scenarios that have high impact on these particular job types. As an example, in Section 3.4 we compare data-intensive application versus CPU-intensive application workload behavior under different failures and find that the behaviors are different in important ways. Using those scenarios, we are creating an FSaaS model that would allow users of Hadoop clusters to pick the proper template failure scenario to run against their applications. Failures in the model are randomized and are synthetically introduced into components, allowing a Hadoop application to see a series of failures that simulate real-world failures. Our tools can also inject real faults into running Hadoop applications, as a mechanism to best mimic real-world dependability challenges.

The rest of this chapter is arranged as follows. In Section 3.2, we will talk about related work in this area and compare it to our own work. In Section 3.3, we will discuss our methodology and the design of the experiment.
we conducted. Section 3.4 describes experimental results. We conclude in
Section 3.5.

3.2 Related Work

When Netflix moved from their own data centers to Amazon Web Services,
they developed Chaos Monkey [6] in order to evaluate how potential failures
in AWS would affect their ability to provide continuous services. Chaos
Monkey would kill EC2 instances to test how they affected overall services
to clients. Netflix performed the tests in real scale, enabling them to find the
bottlenecks in their system and areas where improvements were necessary.
Because failures in data centers are common [2], this type of testing will be
important for most organizations running their services on the cloud.

Dinu et al. performed an evaluation of Hadoop’s performance under com-
pute node and process failures [3]. They observed that even single failures
had detrimental effects on running times of jobs. It was observed that several
design decisions in Hadoop, such as delayed speculative execution (SE), the
lack of sharing of failure information, and overloading of connection failure
semantics, make Hadoop’s performance sluggish in the presence of failures.
Jin et al. derived a stochastic model to predict the performance of MapRe-
duce applications under failures [10]. They generated synthetic data to run
their MapReduce simulator to confirm the accuracy of their model.

[11, 12] present tools for efficient injection of failures into cloud software
systems, like HDFS, and evaluation of cloud recovery. Cloud software sys-
tems like Hadoop include fault tolerance and failure recovery, but some fail-
ures may have unpredictable effects on performance of Hadoop [3]. Also,
some failures may not be accounted for when failure recovery is implemented
in cloud software systems; or failure recovery may even be buggy [3]. Hence,
[11] is based on the need for state-of-the-art failure-testing techniques. The
authors address the challenges of dealing with combinatorial explosion of
multiple failures through their work in PreFail [11]. PreFail is a pro-
grammable failure injection tool that provides failure abstractions to let
testers write policies to prune down large spaces of multiple-failure com-
binations. The main goal of PreFail is to find reliability bugs in large-scale
distributed systems. [12] presents a similar tool called Fate, a framework for
cloud recovery testing. FATE is designed to *systematically* push cloud systems into many possible failure scenarios. Similar to [11], FATE aims to solve the challenge of massive combinatorial explosion of failures by implementing smart and efficient exploration strategies of multiple-failure scenarios. FATE achieves fast exploration of failure scenarios by using strategies that prioritize failure scenarios that result in distinct recovery actions. In summary, PreFail [11] and FATE [12] are tools that allow users to inject failures into cloud systems to analyze their failure recovery. They both aim to solve the challenge of combinatorial explosion of multiple failures, PreFail through pruning policies and FATE through prioritization strategies. However FATE is deployed in only three cloud systems (HDFS, ZooKeeper, and Cassandra) and their systematic behavior during failure scenarios. Instead, we focus on failure impact on jobs running on Hadoop as a more high-level framework. Our approach is to systematically analyze the MapReduce jobs rather than the frameworks.

### 3.3 FSaaS Design

Our goal is to implement FSaaS for Hadoop clusters by designing a set of failure scenarios that contain collections of effective failures for specific types of applications. FSaaS and the scenarios can be used by organizations as a Quality Control tool to improve their applications. Also, it can be used as a method of determining upper bounds on required resources for applications with possibilities of failure during operation, to satisfy service-level agreements (SLA).

We first identified a set of various common failures that may affect Hadoop jobs, described in Section 3.3.1. Then we evaluated effects of individual failures on different types of job, and evaluated the performance of the jobs against the failures and a combination of them. As a result, we have developed a set of sample scenarios (described in Section 3.3.2) for a few different types of workloads (Section 3.3.3).

To make efficient use of the FSaaS service, we allow users to find a failure template to evaluate their job by selecting a job type (I/O intensive, CPU intensive, or Network intensive) and appropriate matching failure scenario for their particular Hadoop job. The template injects the set of failures into
a Hadoop cluster, and helps the users to identify weaknesses, bottlenecks, and hot spots in their application in the presence of failures.

Next we describe the failure injection framework, the different MapReduce applications representing MapReduce job types in our case study, the utilized evaluation metrics, and our experimental testbed.

3.3.1 Failure Injection Framework

We are using AnarchyApe [13] as our failure injection base code. AnarchyApe is an open-source project, created by Yahoo!, developed to inject failures in Hadoop clusters. About ten common failures have been implemented in AnarchyApe, and more can be added. Each failure in AnarchyApe is implemented as a failure command. In our templates, we execute these commands to inject the failures into nodes in the Hadoop cluster (see Section 3.3.2).

Here are some common failures in Hadoop environments, proposed in [13]:

- Data node is killed
- Application Master (AM) is killed
- Application Master is suspended
- Node Manager (NM) is killed
- Node Manager is suspended
- Data node is suspended
- Tasktracker is suspended
- Node panics and restarts
- Node hangs and does not restart
- Random thread within data node is killed
- Random thread within data node is suspended
- Random thread within tasktracker is killed
- Random thread within tasktracker is suspended
- Network becomes slow
- Network is dropping significant numbers of packets
- Network disconnect (simulate cable pull)
- One disk gets VERY slow
- CPU hog consumes x% of CPU cycles
- Mem hog consumes x% of memory
- Corrupt ext3 data block on disk
• Corrupt ext3 metadata block on disk

In our case studies, we have used a handful of these possible failures (Section 3.4).

3.3.2 Failure Scenarios

Currently, to create a scenario, the user constructs a shell script specifying the types of errors to be injected or failures to be simulated, one after another. A sample line in a scenario file could be as follows:

```
java -jar ape.jar --remote cluster-ip-list.xml -fb lambda -k lambda
```

where the -fb is a “Fork Bomb” injection, the -k is a “Kill One Node” command, and the lambda specifies the failure rates.

Users can define lambda parameters by computing Mean Time Between Failures (MTBF) of a system. MTBF is defined to be the average (or expected) lifetime of a system and is one of the key decision-making criteria for data center infrastructure systems [14]. Equipment in data centers is going to fail, and MTBF helps with predicting which systems are the likeliest to fail at any given moment. Based on previous failure statistics, users can develop an estimate of MTBF for various equipment failures; however, determining MTBFs for many software failures is challenging.

To evaluate our case studies in this paper, we have developed a set of ready-to-use failure scenarios. However, the easy programmability of FSaaS ensures its use in other use cases. Users can create user-defined scenarios; as an example, a user can categorize data center failures into two groups: equipment failures and HDFS failures, and use their known failure rates or some publicly available reports in order to set lambda parameters. Some publicly available data for data center failures in Google can be find at [15] and for HDFS failures from Hortonworks at [16].

3.3.3 MapReduce Applications

At Google alone [5], more than ten thousand distinct MapReduce programs have been implemented; hence, for the FSaaS to be applicable to all types of
programs, good program classifications are needed. We categorize MapReduce programs as network-(data transfer), CPU-(computation), or I/O-(local data access) intensive.

We have used the following representative applications for the above resource-intensive categories:

- **Sort** sorts a set of records that have been distributed into S partitions and is inherently network-intensive [9].

- **WordCount** computes the frequency of occurrence of each word in a large corpus and reports the ten most common words. WordCount performs extensive data reduction on its inputs, so it is CPU-intensive and transfers very little data over the network [9, 17].

- **RandomWriter** randomly chooses words from a small vocabulary (100 words) and forms them into lines in Map task. The map outputs are directly committed to the distributed file system, so there is no Reduce task in RandomWriter. RandomWriter is I/O-intensive [17].

3.3.4 Profiling

Since our testbed is set up on Amazon AWS [18], we have used the Amazon CloudWatch [19] service to monitor AWS cloud resources. Amazon CloudWatch monitors AWS resources such as Amazon EC2, and with it, one can gain system-wide visibility into resource utilization, application performance, and operational health [19].

3.3.5 Evaluation Metrics

We can use various metrics to measure the impacts of failures on different application types. As different applications have different properties, our FSaaS service can use more than one metric. Some commonly used metrics in large-scale networks are as follows:

- **Makespan**: total time taken by an experiment until the last job completes.
Table 3.1: Amazon EC2 Instance Specs

<table>
<thead>
<tr>
<th>Memory</th>
<th>1.7 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2 Compute Unit</td>
<td>1</td>
</tr>
<tr>
<td>Storage</td>
<td>160 GB</td>
</tr>
<tr>
<td>Platform</td>
<td>32-bit or 64-bit</td>
</tr>
<tr>
<td>I/O Performance</td>
<td>Moderate</td>
</tr>
<tr>
<td>API name</td>
<td>m1.small</td>
</tr>
</tbody>
</table>

- **System Normalized Performance (SNP):** the geometric mean of all the ANP values for the jobs in an experiment, where ANP stands for the “Application Normalized Performance” of a job [20].

- **Slowdown norm:** some scaled $l_p$ norms of the slowdown factors of the jobs across each experiment [9].

- **Data Transfer (DT):** the total amount of data transferred by all tasks during an experiment. DT is split into 3 components: data read from local disk, data transfer across a rack switch, and data transfer across the central switch.

In this paper we have used Makespan for evaluations. We intend to use additional metrics in future work.

### 3.3.6 Testbed Description

For our testbed, we used Amazon Web Services (AWS) [18] EC2 machines and ran MapReduce jobs through the AWS Elastic MapReduce service. AWS is a widely used Infrastructure as a Service (IaaS) for running cloud services and it simplifies the setting up of Hadoop clusters to run MapReduce jobs through the Elastic MapReduce service. AWS provides a highly scalable infrastructure, which is essential for our experiments.

We set up our Hadoop cluster with 10 AWS EC2 machines. Each machine was a standard small instance type. Table 3.1 gives details on that instance type.
3.4 Evaluation

In this section, we present the results gathered from our experiments and discuss the data that led to the creation of the failure scenarios. As stated in Section 3.3, we gathered a variety of different failures that could happen inside a Hadoop cluster and injected them into different Hadoop jobs. The Hadoop jobs we selected represented three different types of workloads, namely I/O-intensive, CPU-intensive, and network-intensive. Those workloads put heavy pressure on different resources within a Hadoop cluster; hence, we can use them to identify better set of failures that have larger impacts in these different failure scenarios. Our results show that the behavior of each of the workloads under our failure scenarios varied significantly according to the type of workload. In our experiments, we chose the following three Hadoop jobs: Random Writer, Hadoop Sort, and Word Count. They correspond to I/O-intensive, network-intensive, and CPU-intensive workloads, respectively.

Initially, we observed the impacts of different failures on different Hadoop jobs. First, we implemented the following failures in AnarchyApe [13]: kill nodes, disconnect network for duration of period, slow down network for duration of period, fork bomb at a particular node, and drop network packets for duration of period at a certain rate. Second, we ran our three different types of Hadoop jobs many times, each time with one of the above failures injected. Our goal in the experiments was to determine the impact levels of the above failures on the different job types.

In Figure 3.1, bars “None” through “Packet Drop” show job completion times for Word Count, Random Writer, and Sort Hadoop jobs with injection of different failures. We can see that different failures had different levels of impact on the completion times of the Hadoop jobs. The “None” bars show the Hadoop job completion time with no failures. When no failures were injected, the Word Count job took, on average, about 40 minutes. For a “Kill One Node” injection, one of the Hadoop nodes was terminated during the execution. Killing one node during the execution did not affect the running time at all. This shows that for CPU-intensive jobs, Hadoop is well-built to recover from individual node failures without loss of computation time. In a “Network Disconnect” failure, we brought down the network interface for a few minutes in several nodes. In the “Network Slow” failure, we slowed down network packets for a few milliseconds at a few nodes. In a “Fork
Bomb” failure, we started a fork bomb at a few Hadoop nodes. Finally, in the “Packet Drop” failure, we dropped a particular percentage of packets at several Hadoop nodes. These additional failures had a significant impact on completion time, Fork Bomb being the worst culprit. For a network-intensive workload (Sort), Network Disconnect and Packet Drop had surprisingly little impact on job completion times. After further investigation, we discovered that the Sort job has many bursty network communications, so Packet Drop and Network Disconnect failures may not have a big impact unless they occur during times of bursty network communication. For the I/O-intensive workload, Random Writer, there were no failures that affected the job completion time significantly more than others. The last column shows the combination of the top three failures for each job.

Figure 3.2 shows the remaining map tasks measured against time for the Word Count, Sort, and Random Writer Hadoop jobs. The slope of the lines gives us some insight into how Hadoop handles failures as they are being injected. For Word Count, the lines are fairly level for the Network Slow and Network Disconnect failure injections, meaning that Hadoop did not do anything to help when those failures occurred, and that the failures caused a steady increase in completion time for each map task. The Fork Bomb data show exactly what we suspected. The more horizontal part of the graph shows that for a period of 7 minutes, no map tasks were completed,
Figure 3.2: Remaining Map tasks for Word Count, Sort, and Random Writer.
severely degrading the performance of the jobs. For Random Writer jobs, the remaining map tasks were affected differently by the different failure injections, but they all ended up leveling out to similar results towards the end.

After identifying failures with the most impact on each of the different jobs, we proceeded to inject combinations of failures into each of the jobs. For each job (Word Count, Random Writer, and Sort), we selected three of the failures with the most impact on that job and injected combinations of these three failures to observe the resulting behavior. For Word Count jobs, we injected a combination of Network Disconnect, Network Slow, and Fork Bomb failures. For Random Writer, we injected Network Disconnect, Fork Bomb, and Packet Drop. Finally, for Sort jobs, we injected Kill Node, Network Slow, and Fork Bomb failures. Interestingly, the Word Count job completed in less time compared to completion times under the three failures injected individually and separately. The most reasonable explanation is that failures have different impacts depending on the times they are injected in relation to the status of the running jobs, which relates to the Hadoop job phases at the moment of injection. On the other hand, and as expected, the Random Writer and Sort jobs took longer to complete under the combination of failures than under any individual failures.

In review, even though we did not experiment with a huge number of applications, our study does show that application behavior varies considerably, depending on the type of job and the phase of the job that is executing when a failure occurs.

Now, we study potential opportunities of masking failures in single MapReduce jobs and DAG jobs. Our evaluation consists of empirically analyzing the impact of failures on MapReduce job duration and finding the worst case situations where failure has highest impact on makespan.

Our testbed is a 10 node Hadoop cluster on phoenix.illinois.edu cluster. Each node has 2 processing core, 6 GB of memory and 30 GB of SSD. We have stored 3.8 GB of Twitter data, total of 516474 tweets on HDFS streamed by Flume from Twitter Streaming API.

Our MapReduce jobs are generated by Hive queries. The first analysis is on a simple Hive query select count(*) from tweets; this query is equivalent of MapReduce WordCount and will generate Mapers on each datanode then runs one reducer in order to combine the results.
Figure 3.3: \textit{select count(*)} behavior under various time different failure injections.

Figure 3.4a illustrates various Map Reduce failure-free phases of this query over time. Each datanode runs one Mapper and after all the mappers are done one datanode will be responsible for the last count of tweets. The failure-free execution time is around 90 second, however, with node failure after 20 second in figure 3.4b and after 60 seconds in figure 3.4c, job completion time will dramatically increase.

In our setup, we have one TaskTracker in each datanode. TaskTracker sends a heartbeat messages to the JobTracker, by default the timeout for TaskTracker to be considered Killed is 10 minutes; since the scale of our experiments is much smaller in our setup we have changed this value to 15 seconds by overriding the parameter \texttt{mapred.tasktracker.expiry.interval}. This could be also seen in the figures as the JobTracker has waited 15 seconds before recovering the failed node.

From Figure 3.3, we can see one important factor in job completion time under failure is the time in which a failure has been injected. The Figure shows results of 37 different fault injections; each point represents the time at which a failure has been injected and its effect on its makespan under that failure. It is clear that, if a failure occurs at later stages of a MapReduce job it will take longer time for that job to finish; one reason is results of already calculated mappers are inaccessible, therefore, JobTracker has to re-schedule...
those tasks, when as if the failure occurs at earlier stages there is more time
to recover failed tasks. This could potentially gives us a good estimate of an
upper bound of job’s makespan under failure.

From this observation, we can see that if a failure occurs toward the end of
a single job, there is not much opportunity to mask the extra latency caused
by the failure.

3.5 Conclusion

Failure in the cloud is a normal occurrence now, and ensuring that an applica-
tion can withstand large and widespread failure is essential for the longevity
of the service. Hadoop has seen many different types of applications across
industry and academia, and, like other cloud services, it needs to be tested
against different types of failures. Running an FSaaS against Hadoop ap-
plications can help to identify failure vulnerabilities in those applications,
allowing developers to fix them and provide better service.

Services running in the cloud are difficult to test using traditional meth-
ods. Having a background service continually causing different parts of the
infrastructure to fail would go a long way towards identifying faults in a
running application on a production environment. One of the reasons it is
difficult to test at such a large scale is that the number of failures that can
occur at any one time is very large. We have presented a system for profiling
Hadoop applications in an attempt to narrow down the testing strategy and
allow for a Hadoop application to be failure-tested as efficiently as possible.
Such testing can serve two purposes. One is to identify weak spots in an
application and attempt to fix them. The other is to identify the quantity of
running cloud resources you need to stave off complete failure in the event
of isolated system crashes.
Figure 3.4: MapReduce execution scenarios with failure for `select count(*)` Hive query.
Chapter 4

PERFORMANCE OF DEADLINE-DRIVEN MAPREDUCE JOBS

4.1 Motivation

The motivation of this study comes from the increasingly strict latency requirements for data-intensive jobs. In our world today, a couple of microseconds faster could result in a huge financial gain. For example, High-Frequency Trading (HFT) trades, which accounted for approximately 50% of all US equity trading volume in 2012 [21], have decreased the execution time from several seconds in 2000 to several milli- and even microseconds in 2010 [22]. In order to make that kind of lighting fast and risky decision, a huge volume of data coming from many sources needs to be accurately processed. A small delay would take away millions of dollars.

Current technologies, particularly Cloud Computing and parallel job processing frameworks (e.g., MapReduce [5] and Dryad [23]), are considered new key ingredients for winners in this speeding race. Many parallel clusters based on these frameworks have been used in production systems, which serve real-time, recurring, and business-critical jobs [24]. However, there are still many major roadblocks before these technologies can be faster adopted to replace existing in-house server farms and software. One of them is the current lack of effective latency-assurance techniques for parallel jobs.

The latency of current parallel job processing frameworks is sensitive to failures [25, 26], which are the norm rather than the exception in nowadays large-scale data centers [27, 28]. There are many sources of failure in data centers, namely hardware failure, software failure, and malicious attacks. In addition, commodity hardware is seeing increasing use in data center, which largely increase the component failure rates. For example, Google reports the Mean Time To Failure (MTTF) of computer nodes in a studied cluster was only 4.3 months [28].
Previous studies, in the area of *automating resource allocation for parallel jobs*, such as [29] and [30] address the problem of inferring and allocating resource for MapReduce and Hive/Pig programs to meet the SLO. However, these studies do not take into account failures which might interrupt their models. Ferguson et al. present a system call Jockey [31], which also aims providing latency guaranteeing for parallel jobs. In order to compensate for potential future failures, Jockey conservatively allocates additional resources at the start, however, this approach is unnecessary when no failure occurs.

In this chapter, we describe some adjustments to previous models in order to account for failures. In Chapter 3, we analyzed various outcomes of failure over latency sensitive jobs. we studied potential opportunities of masking failures in single MapReduce jobs and DAG jobs; based on our founding, scheduler possible reactions against partially failed jobs are: kill the job and start over; or, allocate more recovery resources to the partially failed job or the next jobs in the DAG. In this chapter We will study the second option in more details, we propose our reactive strategies to deal with failures. The idea is we increase allocated resources to a job (i.e., increase the numbers of map and/or reduce slots) on-the-fly only when a failure is detected. Hence, if no failure is detected, with the optimally allocated resources, the job will finish by the deadline. In case a failure is detected, the scheduler will re-estimate the necessary resource to make up the lost time caused by failure in order to meet the the deadline.

We propose a reactive solution to mask the effect of failures for MapReduce jobs. Specifically, our solution consists of two strategies, which are applied in order:

- Reactively increasing resource allocation for a single parallel job when a failure happens to compensate for the extra latency caused by the failure. If the extra latency cannot be masked, i.e., the job misses its own deadline, the strategy is considered failed, then the next strategy is applied.

- Reactively increasing resource allocation for the next phase in the DAG of parallel jobs. In case there is no good solution that can completely mask the extra latency, the next strategy is applied.

Our proposal is based on [29] to estimate the completion time of MapReduce jobs in the presence of failures. The completion time models take the
number of allocated resources (i.e., map and reduce slots) as parameters to output the estimated completion time. This model can be inversely solved to find the appropriate resource (number of map and reduce slots) to satisfy a given deadline - a predefined completion time. Our newly adjusted models require parameters such as delay time caused by failures in different phases of MapReduce job. We use FSaaS, discussed in Chapter 3 to obtain these parameters.

We first review ARIA completion time estimation model. Next, for each strategy we discuss a completion time estimation model. Later, we motivate the inverse problem, with which we reactively obtain estimates of the resource allocation for MapReduce jobs given their deadlines, after failure occurrence.

4.2 Review ARIA

ARIA [29] provides a model to estimate the lower and upper bounds on the completion time of MapReduce jobs. A basic block of ARIA’s model is the makespan theorem to determine the performance bounds for a given set of $n$ tasks that is processed by $k$ servers. Each server can process only one task at a time. In a MapReduce environment, a task could be a map task or a reduce task, a server could be a map slot or a reduce slot, respectively. Let $T_1, T_2, \ldots, T_n$ be the duration of $n$ tasks. The assignment of tasks to servers is done using a simple, online, greedy algorithm: assign each task to the server with earliest finishing time. Let $avg$ and $max$ be the average and maximum duration of the $n$ task respectively.

The makespan theorem says: the makespan (or completion time) of the greedy task assignment is at least:

$$T^{low} = \frac{n \cdot avg}{k},$$

and at most:

$$T^{up} = \frac{(n - 1)avg}{k} + max$$

This theorem can be applied to compute the completion time range of each phase of a MapReduce job, given the average and maximum bounds can be obtained from profiling or analyzing past execution logs. A MapReduce job
is divided into three phases: map \((T^{\text{low}}_M, T^{\text{up}}_M)\), shuffle/sort \((T^{\text{low}}_{Sh}, T^{\text{up}}_{Sh})\), and reduce \((T^{\text{low}}_R, T^{\text{up}}_R)\). Due to the fact that \(k\), the number of servers, is usually smaller than \(n\) (number of tasks), tasks execution occur in waves: a group of \(k\) tasks execute concurrently.

The final formula for the lower and upper bounds of the overall completion time of a MapReduce job \(J\) are:

\[
T^{\text{low}}_J = T^{\text{low}}_M + Sh^{\text{avg}}_1 + T^{\text{low}}_{Sh} + T^{\text{low}}_R
\]

\[
T^{\text{up}}_J = T^{\text{up}}_M + Sh^{\text{max}}_1 + T^{\text{up}}_{Sh} + T^{\text{up}}_R
\]

\(Sh^{\text{avg}}_1\) and \(Sh^{\text{max}}_1\) are the average and maximum non-overlapping portions of the first shuffle wave. These parameters can be obtained from profiling or past executions.

Based on this equation, the inverse problem can be solved to obtain estimates of the resource allocation for MapReduce jobs given their deadlines.

4.3 Reactive Latency Masking for a Single MapReduce job

This section describes our solution to reactively increase the resource to cope with failures in a single parallel job. In order to do that, we need to understand how failures affect parallel jobs, particularly their completion times. The discussion uses MapReduce model, but we believe it could be generalize to other parallel job models, which also have dependencies between execution phases.

We use ARIA model [29] to analyze the effects of failures when they happen in different phases of MapReduce jobs. Based on the analysis, we introduce a solution based on speculative execution to mask failures, which occur early in MapReduce executions.

4.3.1 Effects of Failures on the Completion Time of parallel jobs

This sections analyzes the effects of failures that terminate or delay only one single task in a MapReduce job. This analysis can be generalized for failures
that affects multiple tasks.

 Failures Affecting Map Phase

 This section analyzes the effects of failures that occur in map phase of MapReduce jobs. Figure 4.1b and 4.1c illustrate two examples of failures occurring in map tasks. We consider the execution time of a wave (could be either map, shuffle, or reduce wave) is the non-overlapping time after the previous phase finishes until all the tasks in the wave finish. In failure-free execution, tasks in a same wave tend to finish close together, therefore the overlapping portion between two consecutive waves is often small. However, in an execution affected by a failure, the overlapping time is the additional time incurred by the failure (i.e., failure detection and task re-execution time).

 When a map task experiences a failure, the time incurred by the failure is counted toward the wave that the map task belongs to. The completion time of other waves can still be estimated by the method described in Section 4.2. Let $T_{M}^{\text{fail}}$ denote the additional time added by the failure to the overall completion time of the map phase. $T_{M}^{\text{fail}}$ can be obtained by performing fault injections to Map tasks.

 The completion time of the first shuffle wave is also affected by the failure. Recall that the completion time of the first shuffle wave is the non-overlapping time of the first shuffle wave with the map phase. The overall completion time of the map phase changes, therefore overlapping portion of the map phase
and the first shuffle wave changes as well. Consequently, the non-overlapping time portion of the first shuffle wave have to change. Let \( Sh_1^{max} \) denote the non-overlapping time of the first shuffle wave in an execution affected by failures.

The completion time of other shuffle waves and map waves remain the same because of the strict barriers between these waves.

Equation 4.2 is adjusted as follows when a failure is present in Map tasks:

\[
T_{J}^{up} = T_{M}^{up} + T_{M}^{fail} + Sh_1^{max} + T_{Sh}^{up} + T_{R}^{up} 
\]  

(4.3)

The added time caused by a failure in map phase to the overall completion time of the MapReduce job \( J \) is

\[
\Delta T_{J}^{Mfail} = T_{M}^{fail} + Sh_1^{max} - Sh_1^{max} 
\]  

(4.4)

Failures Affecting Shuffle Phase

This section analyzes the effects of failures that occur in shuffle phase of MapReduce jobs. Figure 4.1d and 4.1d illustrate two examples of failures occurring in shuffle phases. Since the shuffle phases only use output of the map phases, failures in the shuffle phases do not affect the map phases. A shuffle phase has a strict barrier with the next reduce phase. Therefore, failures in the shuffle phases do not affect the completion time (in term of the length) of the reduce phase.

Let \( T_{Sh}^{fail} \) denote the additional time added by the failure to the overall completion time of the shuffle phase. Equation 4.2 is adjusted as follows when a failure is present in shuffle phases:

\[
T_{J}^{up} = T_{M}^{up} + Sh_1^{max} + T_{Sh}^{up} + T_{Sh}^{fail} + T_{R}^{up} 
\]  

(4.5)

The added time caused by a failure in shuffle phase to the overall completion time of the MapReduce job \( J \) is

\[
\Delta T_{J}^{Shfail} = T_{Sh}^{fail} 
\]  

(4.6)
Failures Affecting Map Phase

This section analyzes the effects of failures that occur in reduce phase of MapReduce jobs. Figure 4.1f illustrates an examples of failures occurring in a map phase. Similar to shuffle phase, failures in reduce phase do not affect neither map phase nor shuffle phase.

Let $T_{R}^{\text{fail}}$ denote the additional time added by the failure to the overall completion time of the reduce phase. Equation 4.2 is adjusted as follows when a failure is present in reduce phases:

$$T_{j}^{\text{up}} = T_{M}^{\text{up}} + Sh_{1}^{\text{max}} + T_{Sh}^{\text{up}} + T_{R}^{\text{up}} + T_{R}^{\text{fail}}$$ (4.7)

The added time caused by a failure in reduce phase to the overall completion time of the MapReduce job $j$ is

$$\Delta T_{j}^{Rfail} = T_{R}^{\text{fail}}$$ (4.8)

We need to obtain $T_{M}^{\text{fail}}, Sh_{1}^{\text{max}}, T_{Sh}^{\text{fail}},$ and $T_{R}^{\text{fail}}$ (i.e., via fault injection) in order to predict the job completion time under failures. These parameters will be obtained by profiling applications using FSaaS. Once these parameters are obtained, we could also develop strategies to allocate resources to allow MapReduce jobs still meet deadlines even when failures happen.

4.3.2 Masking Failure with Speculative Execution

Figure 4.2: Utilizing speculative execution to mitigate the effects of failures in MapReduce jobs

The first strategy is to speed up the MapReduce job that experiences failure. Figure 4.2 illustrates how this strategy works. When a failure is detected in a map phase, we utilize speculative execution to launch all tasks in the next waves together when their scheduled times come, regardless of the unavailability of the slot that is executing the failure-affected task.
4.4 Reactive Latency Masking for a DAG job

This section describes our solution to reactively increase the resource to cope with failures in a DAG of parallel jobs.

We use the model proposed by [30] to analyze the effects of failures on DAG jobs. From the analysis, we find that some failures can be masked by the concurrency nature of DAG jobs. In case a failure does affect the overall latency of the DAG job, we propose to reactively re-estimate the resource needed to speed up the rest of jobs in DAG. Figure 4.3c illustrates this strategy.

The next sections present our failure analysis of DAG jobs.

4.4.1 Model DAG jobs

A DAG job consists of multiple parallel jobs executing in stages. The structure of the execution plan, including both concurrent and sequential branches, can be expressed by a DAG of parallel jobs. Figure 4.4 shows an example of DAG execution plan. The plan consists of seven MapReduce jobs \{j_1, j_2, j_3, j_4, j_5, j_6, j_7\}.

Due to the dependencies between jobs (expressed by edges in DAG), MapReduce jobs in a plan execute in stages. Jobs in a same stage can be executed concurrently. A job can only be started when all jobs of previous stages have finished. In the example shown in figure 4.4, there are five execution stages:

- first stage \{j_1\};
- second stage \{j_2, j_3\};
4.4.2 Estimate Completion Time of Failure-free DAG jobs

It is trivial that the completion time of each DAG job is the summation of the completion times of all its stages. Let $T_{S_i}$ denote the completion time of the stage $S_i$. The completion time of a $S$ stages DAG job can be estimated as follows:

$$T_P = \sum_{1 \leq i \leq S} T_{S_i}$$ (4.9)

It is, however, non-trivial to estimate the completion time of each individual stage due to the concurrency jobs. Even more challenging, the completion time depends on the jobs’ execution orders, which by default is non-deterministic in many execution engines (e.g., Pig and Hive). [30] proposes that given the ability to estimate the completion time of each individual parallel job in a stage, it is possible to determine an optimal execution order for a stage, such as using Johnson’s algorithm [32]. With the optimal execution order, [30] proposes the following model for estimating the completion time of a stage. Let:

- $\{J_1, J_2, ... J_{|S_i|}\}$ be the optimal execution order of jobs in stage $S_i$;
- $S_{J_i}^M$ be the start time of job $J_i$’s map phase;
- $E_{J_i}^M$ be the end time of job $J_i$’s map phase;
- $S_{J_i}^R$ be the start time of job $J_i$’s reduce phase;
- $E_{J_i}^R$ be the end time of job $J_i$’s reduce phase.

Then the stage completion time is:

$$T_{S_i} = E_{J_{|S_i|}}^R - S_{J_i}^M$$ (4.10)
Let $T_{M_i}^M$ and $T_{R_i}^R$ denote the completion times of the map and reduce phases of job $J_i$. Then the end time of map and reduce phases can be estimated as follows:

$$E_{M_i}^M = S_{M_i}^M + T_{M_i}^M$$  \hspace{1cm} (4.11)$$

$$E_{R_i}^R = S_{R_i}^R + T_{R_i}^R$$  \hspace{1cm} (4.12)$$

In an optimal order, a map phase can start right after the map phase of the previous job finishes. But the reduce phase can only start after both the reduce phase of the previous job and its map phase finish. Figure 4.6a and Figure 4.5a illustrate two examples of this relation. Thus, the start times of map and reduce phases can be estimated as follows:

$$S_{M_i}^M = E_{M_{i-1}}^M$$  \hspace{1cm} (4.13)$$

$$S_{R_i}^R = \max\{E_{M_i}^M, E_{R_{i-1}}^R\}$$  \hspace{1cm} (4.14)$$
4.4.3 Estimate the Completion Time of DAG Jobs under Failures

Failure in a stage of a DAG job

When a failure occurs in a MapReduce job, it will increase the completion time of that job as analyzed in section 4.3.1. This section analyzes how this increasing affects the completion time a stage in DAG.

Let consider a failure occurs in the map phase of job $J_i$. The new estimated end time of the map phase is:

$$E_{M_{J_i}}^{M} = E_{M_{J_i}}^{M} + \Delta T_{Mfail}^{M}$$  \hspace{1cm} (4.15)

The start time of the reduce phase of job $J_i$ is affected as follows:

$$S_{R_{J_i}}^{R} = \max\{E_{M_{J_i}}^{M} \}, E_{R_{J_{i-1}}}^{R}\}$$  \hspace{1cm} (4.16)

$$\iff S_{R_{J_i}}^{R} = \max\{E_{M_{J_i}}^{M} + \Delta T_{Mfail}^{M}, E_{R_{J_{i-1}}}^{R}\}$$  \hspace{1cm} (4.17)

Therefore:

$$S_{R_{J_i}}^{R} = S_{R_{J_i}}^{R} \iff E_{M_{J_i}}^{M} + \Delta T_{Mfail}^{M} \leq E_{R_{J_{i-1}}}^{R}$$  \hspace{1cm} (4.18)

$$\iff \Delta T_{Mfail}^{M} \leq E_{R_{J_{i-1}}}^{R} - E_{M_{J_i}}^{M}$$  \hspace{1cm} (4.19)

This means the effect of the failure in the map phase could be masked (i.e., show no effect on the overall completion time of the stage) if the condition (4.19) is satisfied: in the failure-free execution, the end time of the reduce phase of the previous job is sufficiently later than the end time of the map phase of the current job. Figure 4.5 illustrates a scenario when condition (4.19) is satisfied between to jobs. Figure 4.6 illustrates a scenario when condition (4.19) is not satisfied causing an increase in the overall completion time of two jobs.

However, when a failure occurs in a shuffle phase or in a map phase, it always results in an increase in the overall completion time of the stage.
4.5 Conclusions and Future Work

This chapter presented our initial study toward the design and implementation of a tool that can provide latency-guarantee for MapReduce jobs in the presence of failures. The tool is a combination of two strategies which aim at masking the latency effect of failures at different levels. First, the tool attempts to mask the latency-effect in each single parallel job that suffers the failure. When the first strategy fails, the second strategy is applied to mask the latency by increasing resources for other parallel jobs in the next phases of the DAG jobs.

For each strategy we discussed a completion time estimation model. In future work, we use this model to build an inverse problem, with which we can reactively obtain estimates of the resource allocation for MapReduce jobs given their deadlines, after failure occurrence.

We believe that our solutions will result in a better overall resource utilization for MapReduce jobs frameworks.


