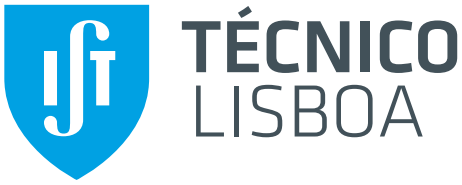

Reliable and Locality Driven Scheduling in Hadoop

Tran Anh Phuong

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Doctor Luís Manuel Antunes Veiga

Examination Committee

Chairperson:	Doctor Luís Eduardo Teixeira Rodrigues
Supervisor:	Doctor Luís Manuel Antunes Veiga
Member of the Committee:	Doctor David Manuel Martins de Matos

July 2014

Acknowledgements

The work here presented is delivered as final thesis report at Instituto Superior Tecnico (IST) in Lisbon, Portugal and it is in partial fulfillment of the European Master in Distributed Computing belonging to cohort of 2012-2014. The Master programme has been composed of a first year at IST, a second year's first semester at Royal Institute of Technology (KTH) and for this work and last academic term, an internship at KerData team INRIA (Rennes, France).

Special thanks to my industrial advisor Dr. Shadi Ibrahim for his support throughout the project. Assisted with his invaluable comments, I was able to successfully conclude this study. I would also like to thank to my supervisor Prof. Luís Antunes Veiga for his continuous support and the encouragement he has provided all throughout the project.

I would like to present my special appreciation to KerData team leader Dr. Gabriel Antoniu for his support and trust in my work.

I am thankful to my family for their valuable support all throughout my life.

Last but not least, to all the professors from IST and KTH that during these last two years challenged me to think out of the box and face always the difficulties in despite of any other matters, they taught me to always work towards bigger and better goals.

This work was done in the KerData team INRIA (Rennes,France). The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/ for details)

Lisboa, September 2, 2014
Tran Anh Phuong

To my family.

Abstract

The increasing use of computing resources in our daily lives leads to data being generated at an unprecedent rate. The computing industry is being repeatedly questioned for its ability to accommodate the unpredictable growth rate of data, and its ability to process them. This has encouraged the development of cluster based data-intensive applications. Hadoop is a popular open source framework known for its massive cluster based data processing power. Hadoop is widely used in the computer industry because of its scalability, reliability, ease of use, and low cost of implementation.

Cloud computing in the recent years has gained increasingly popularity by its cost-efficient and flexible way to leverage the power of commodity hardware. Hadoop based services on the Cloud have also emerged as one of the prominent choices for smaller businesses. However, evidence in the literature shows that faults on the Cloud do occur and normally result with performance problems. Hadoop hides the complexity of discovery and handling failures from the schedulers, but the expenses of failure recovery rest entirely on users, regardless of root causes. We systematically assess these expenses through a set of experiments, and argue that more effort to reduce this cost to users is desirable.

We also analyze the drawback of current Hadoop's mechanism in prioritizing failed tasks. By trying to launch failed tasks as soon as possible regardless of locality, it significantly increases the execution time of jobs with failed tasks, due to two reasons: 1) available slots might not be free up as quickly as expected and 2) the slots might belong to machines with no data on it, introducing extra cost for data transferring through network, which is normally the most scarce resource in nowadays' data centers.

This thesis then introduces a new algorithmic approach called the waste-free preemption. The waste-free preemption saves Hadoop scheduler from choosing solely between kill, which instantly releases the slots but is wasteful, and wait, which does not waste any previous effort but suffers from the two above mentioned reasons. With this new option, a preemptive version of Hadoop default scheduler (FIFO) is implemented. The evaluation demonstrates the effectiveness of the new feature by comparing its performance with the traditional Hadoop mechanism.

Resumo

A crescente utilização de recursos de computação na nossa vida diária leva a que os dados sejam gerados a um ritmo surpreendente. A indústria de computação está sendo desafiada repetidamente na sua capacidade de acomodar a taxa de crescimento imprevisível de dados, e a sua capacidade de processá-los. Isto tem estimulado o desenvolvimento de aplicações intensivas de dados baseadas em clusters. O Hadoop é uma framework open source popular, conhecida pelo seu enorme poder de processamento de dados em clusters. O Hadoop é amplamente utilizado na indústria de computadores, devido à sua escalabilidade, confiabilidade, facilidade de uso e baixo custo de implementação.

A computação em nuvem nos últimos anos tem vindo a ganhar cada vez mais popularidade pela sua forma flexível e eficiente, em termos de custo, para aproveitar o poder de hardware *commodity*. O Hadoop como outros serviços baseados na nuvem surge também como uma das alternativas importantes para as pequenas empresas. No entanto, evidências na literatura mostram que as falhas na nuvem ocorrem e resultam normalmente em problemas de desempenho. O Hadoop esconde a complexidade da descoberta e gestão de falhas dos programadores, mas os custos de recuperação de falhas recaiem quase totalmente sobre os utilizadores, independentemente das suas causas. Avaliamos sistematicamente esses custos através de um conjunto de experiências, argumentando que mais esforço para reduzir esse custo para os utilizadores é necessário.

Analisamos o inconveniente do mecanismo atual da Hadoop em prioritizar tarefas que falharam. Ao tentar iniciar tarefas que falharam o mais rápido possível, independentemente da localidade, isto aumenta significativamente o tempo de execução de trabalhos com tarefas que falharam, devido a duas razões: 1) slots disponíveis podem não ficar livres tão rapidamente quanto o esperado, e 2) os slots podem pertencer às máquinas que não contêm os dados a processar, representando um custo extra para a transferência de dados através da rede, que é o recurso mais escasso nos dias de hoje em data centers.

Apresentamos uma nova abordagem algorítmica chamada de preempção livre de desperdício. A preempção livre de desperdício evita que o escalonador Hadoop escolha apenas apenas entre matar tarefas, o que liberta imediatamente os slots, mas é um desperdício, e esperar, que não desperdiça qualquer trabalho anterior, mas perde para as duas questões acima mencionadas. Com esta nova opção, foi implementada uma versão de escalonador padrão do Hadoop (FIFO) regido por preempção. A avaliação demonstra a eficácia do novo recurso, comparando o seu desempenho com o mecanismo tradicional do Hadoop.

Palavras Chave
Keywords

Palavras Chave

Hadoop

Tolerância a Falhas

Preempção

Processamento Distribuído

Localidade

Desempenho

Keywords

Hadoop

Fault - tolerance

Preemption

Distributed processing

Locality

Performance

Índice

List of Figures

List of Tables

I

Introduction and Background

1 Introduction

"The goal is to turn data into information, and information into insight."
– Carly Fiorina, former chief executive of Hewlett-Packard Company or HP

1.1 Motivation

From the Big Data era

Data insight forms an essential part in today's decision making process. With the massive growth in available data, companies are spending millions of dollars on Business Intelligence and Big Data analytics (Lai 2013). Companies become data-driven, shifting the business policies from the traditional instinct-based decision to analyzing available data. Every two years, the amount of data in the world doubles, and by 2015, it is estimated that the total data on Earth will amount to 7.9 zeta bytes (Roe 2011).

Internet-centric enterprises are among the most active player in Big Data analytic field. Yahoo! uses its large datasets to support research for advertising system. In 2008, Yahoo! reportedly processed 24 billion events per day in the effort of analyzing Web visitors' behavior (TCS). Ebay, an e-commerce organization, allows employees to access 52 petabytes of data (Lampitt 2012) on everything from user behavior to online transactions to customer shipments. Other institutions have also reported to process datasets in the order of terabytes or even petabyte(Thusoo et al. 2010) (Logothetis et al. 2010).

The practice of analyzing huge amounts of data motivated the development of data intensive applications. In this context, Hadoop MapReduce (Apache) is a big data processing framework that has rapidly gained popularity in both industry and academia (Jindal et al. 2011). Facebook, a popular social networking website, is a strong supporter of Hadoop. The company claimed to have the world's largest Hadoop cluster (Borthakur 2010) with more than 2000 machines and running up to 25000 MapReduce jobs per day. Many other enterprises also claim to have Hadoop clusters of various sizes: while most of the enterprises have clusters of less than 100 machines, some consist of up to hundreds of machines (see Table 1.1). The main reasons of such popularity are the ease-of-use, scalability, and failover properties of Hadoop MapReduce.

Cluster Size (No. of machines)	No. of companies, as listed in "Powered by Hadoop" page
<10	42
11-30	31
31-50	11
51-100	14
101-1000	10
>1000	5

Table 1.1: Popularity of Hadoop

To the Cloud

The unprecedented growth in data center technologies and services in the recent years allow smaller companies to take advantages of the "cloud-based" infrastructure. Well-known Cloud providers such as Amazon Web Services(Amazon b), Google App Engine(Google) and Microsoft Azure(Microsoft) respond to the need of data processing by equipping their software stack with MapReduce-like systems. Amazon Elastic MapReduce (Amazon a) is an example for platforms that facilitate large-scale data applications. Many successful case studies have demonstrated the simplicity, convenience and elasticity of MapReduce on Cloud. For example, the New York Times rented 100 virtual machines for a day to convert 11 million scanned articles to PDFs (Gottfrid 2007). Amazon Elastic MapReduce uses Hadoop, and is perhaps a very successful example of scalable MapReduce-Cloud.

Hadoop is also easy to use. Equipped with three different ready-to-use schedulers (FIFO, Fair and Capacity schedulers), Hadoop allows users to have more control in choosing the behavior when it comes to scheduling tasks. Fair and Capacity schedulers also allow administrators to share the cluster with multiple users with certain level of fairness and resource guarantees. The idea is that the available resources in the Hadoop MapReduce cluster are partitioned among multiple organizations who collectively fund the cluster based on computing needs. There is an added benefit that an organization can access any excess capacity not being used by others. This provides elasticity for the organizations in a cost-effective manner.

Regarding reliability, knowing that users are charged for their purchased service in pay-as-you-go model, it is presumably expected that Cloud offers highly reliable environment, but evidence shows that failures do happen in the Cloud (Fox et al. 2009). In fact, researchers interested in fault tolerance have accepted that failure is becoming a norm, rather than an exception. For instance, Dean reported that in the first year of a cluster at Google there were 1000 individual failures and thousands of hard drive failures (Dean 2009). Despite this prevalence of failures, with the absence of clear definition of QoS in the Cloud, the expenses of failures entirely rest on users, regardless of the root causes. With respect to MapReduce-Cloud, Cloud providers rely completely on the fault-tolerance mechanisms which is provided by Hadoop system. This policy entitles users the freedom to tune these fault tolerance mechanisms, but leaving also the consequences and the expenses on the users side.

Hadoop currently handles failures by simply re-executing all the failed tasks. However, all these efforts to handle failures are entirely entrusted to the core of Hadoop and hidden from the task schedulers. To our knowledge, there has been no scheduler that explicitly copes with failure. This potentially leads to degradation in Hadoop's performance. In this study, we address the problem of failure in Hadoop, and present our approach to improve Hadoop performance under failure.

1.2 Contribution

1.2.1 Our contribution

This thesis contributes to the field of data-intensive applications in several ways. First, it systematically assesses the Hadoop architecture, focusing on the fault tolerance mechanisms that Hadoop employs. A set of experiments were conducted to illustrate the limitations of its default mechanism. Second, it explores the possibility of improving the performance by introducing a new Preemption scheduling heuristic in Hadoop. This new option is accompanied with a new scheduler that explicitly deal with failures in Hadoop. Finally, it performs evaluation of the new scheduler and discusses other possibilities to fully utilize the new feature in different scenarios. The specific contributions are as follows:

Assessing the Hadoop architecture and its fault tolerance mechanism

This thesis first analyzes and explains the Hadoop's mechanism to discover and handle failures. A set of experiments demonstrates how Hadoop and its built-in schedulers behave in different scenarios. It briefly discusses potential improvements that can be achieved regarding failures.

Waste-free preemption scheduling

The thesis discusses the potential improvement with the new preemption mechanism. It is the first to present both Map and Reduce preemption mechanism in a work-conserving manner. A new scheduler, namely the Preemptive Locality-driven scheduler that leverages this new option is also presented. The details of design and implementation are briefly discussed.

Evaluation of the new scheduler

We evaluate the new scheduler by comparing it with the already available schedulers that come with the Hadoop distribution package. The evaluation is performed based on the total execution time and the data locality of tasks between different implementations.

1.2.2 Document structure

This thesis is organized as follows. Chapter 2 provides background into the Hadoop framework, the three built-in schedulers as well as a glimpse on the fault-tolerance mechanism of Hadoop. Chapter 3 discusses in detail the failure discovery and handling procedure of Hadoop. Chapter 4 presents some data to illustrate Hadoop's behavior under certain circumstances. Chapter 5 discusses recent works that share the same similar objectives in improving the fault tolerance mechanism in Hadoop. Chapter 6 opens new possibilities to improve Hadoop performance by introducing the work-conserving preemption mechanism with its implementation details. Chapter 7 analyzes the performance of the new feature in comparison with the default scheduler in terms of execution time and data locality. Finally, chapter 8 concludes the thesis with some ideas on how to further improve Hadoop's performance with the introduced feature.

2 Background and Related Work

2.1 Hadoop framework

2.1.1 Map Reduce programming paradigm

MapReduce is a programming paradigm designed for processing and generating large data sets. The MapReduce abstraction is inspired by the Map and Reduce functions, which is commonly found in functional programming languages, such as LISP. Users can easily express their computation as a series of Map and Reduce functions. The Map function processes a series of $< key, value >$ pairs to generate a set of intermediate $< key, value >$ pairs.

$$\text{Map(key1, value1)} \rightarrow \text{list (key2, value2)}$$

Reduce function aggregates all intermediate values that associate to the same intermediate key to produce the final output, also in the form of $< key, value >$ pairs.

$$\text{Reduce (key2, list(value2))} \rightarrow \text{list (key 3, value3)}$$

The WordCount example

WordCount is a simple program that counts the number of occurrences of each word in a large collection of documents. The process can be briefly described as follows:

Map function: inputs are read (typically from a distributed file system) and broken up into a series of single words. Each of the words, as the key, is emitted together with its associate count of occurrence (the value), which is just "1".

Algorithm 1 Map(String key, String value)

```
/*key: document name*/
/*value: document content*/
for each word w in value do
  Emit(w,"1");
end for
```

Reduce function: The pairs are automatically partitioned into groups and sorted according to their key by the framework. At the Reduce function, for each unique key in the sorted list,

the combined result (the sum of all sorted reduce-input pairs of the same key) is calculated and emitted as the final output.

Algorithm 2 Reduce(String key, Iterator values)

```
/*key: a word*/
/*value: a list of counts*/
int result = 0;
for each word v in value do
   result += ParseInt(v);
end for
Emit(w,result);
```

2.1.2 Execution overview

As illustrated in Figure 2.1, when the user program calls the MapReduce function, the following sequence of actions occurs:

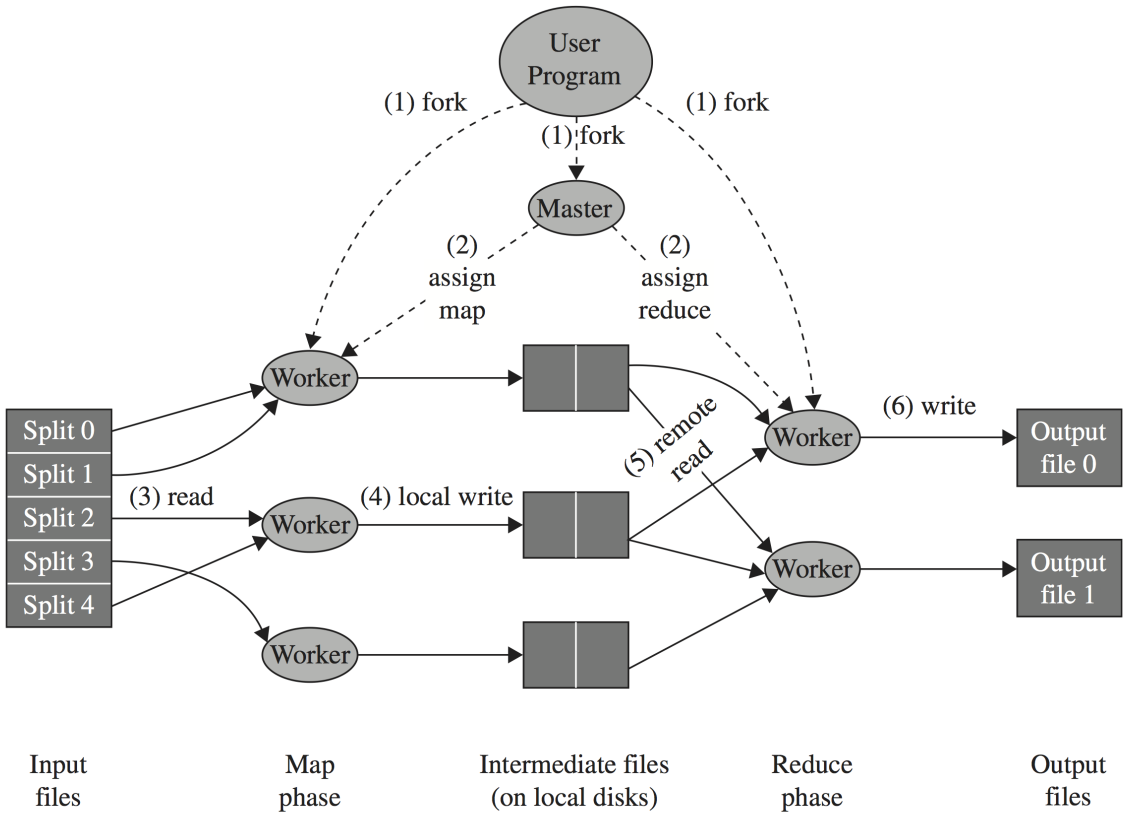

Figure 2.1: Map Reduce's overview execution (Dean & Ghemawat 2008)

The MapReduce framework splits the input files in to M pieces of typically 16 to 128 megabytes (MB) per piece. It then starts many copies of the program on a cluster, including 1

Master and other Workers. The Master node will be responsible for scheduling the job to each Worker, and monitoring the job progress, as well as the workers' health.

Workers with idle capacity will contact the Master asking for tasks of certain types (Map or Reduce, or both). The Master will look into the pool of non-running (including failed) tasks and assign one or more tasks to the Worker. Map tasks are assigned with consideration to data locality. When a Worker is assigned with a Map task, it first reads the content of the corresponding input split (either locally or remotely) and emits $< key, value >$ pairs to the user-defined Map function.

The output of the Map function is first buffered in the memory and periodically written to local disks. A partitioning function partitions the output into R sets, each associated with a Reducer task The Master passes the location of these outputs to Workers with Reduce tasks running, which read these buffered data using remote procedure calls (RPC). The Reduce task then sorts all the intermediate keys so that occurrences of the same key are grouped together. For each key, the entire list of values is passed to the Reduce function. Each Reducer, upon finishing its share of keys, outputs a result file (R output files in total).

2.1.3 Apache Hadoop

Hadoop is an open-source software for reliable, scalable, distributed computing. The project claims are :

(Apache Hadoop)... is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.(Apache)

Hadoop is developed in Java which allows its distribution and portable installation across many available operation systems such as Linux, freeBSD, Windows, Solaris and Mac OSX. It consists on a few sub-projects that we are only interested in a few:

- **Hadoop Common**: consists of the common utilities that support the other Hadoop sub projects.
- **MapReduce**: the implementation of MapReduce, which first appeared in the Apache mailing list in April 2014.
- **HDFS**: a distributed file system that provides high-throughput access to application data.

A common installation of Apache Hadoop includes of the Hadoop distributed file system (HDFS), and the MapReduce implementation. The Hadoop's MapReduce implementation (from now on, we call it Hadoop for short) relies heavily on the HDFS for data storage and

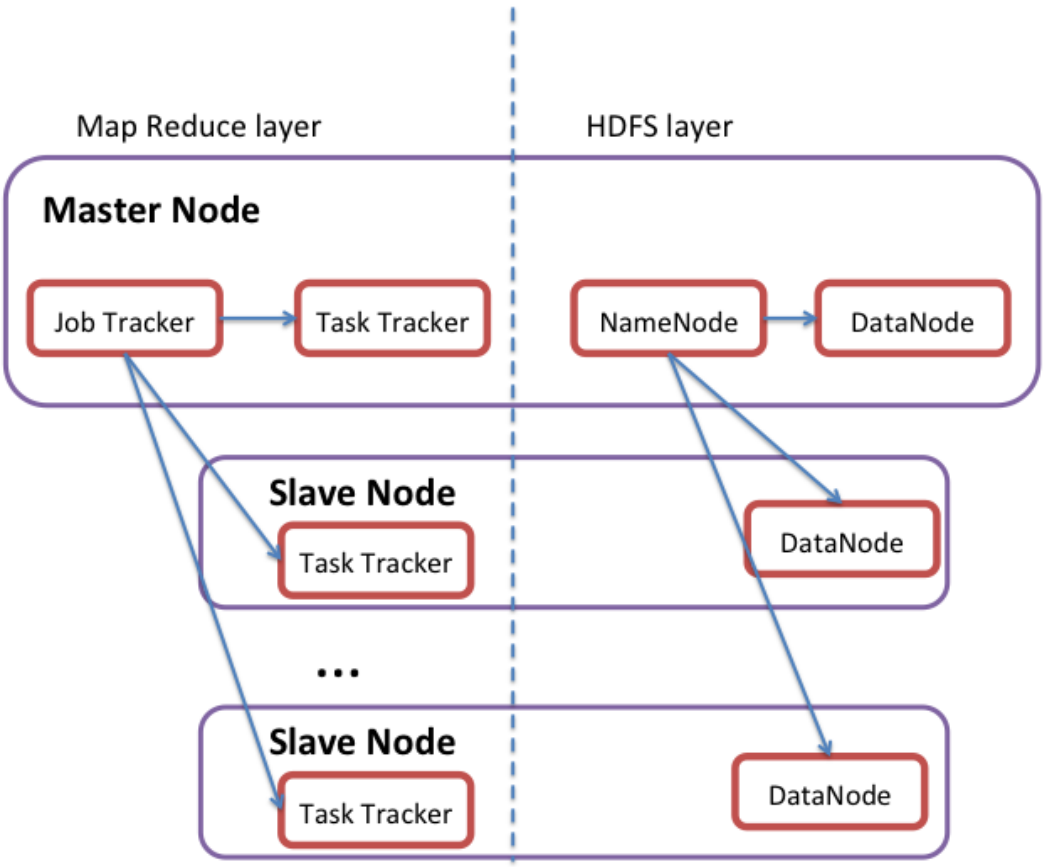

Figure 2.2: Hadoop's simple cluster architecture

access. Both have the Master/Slave architecture: in the Hadoop layer, the Master is called Job Tracker, while the Slaves are called Task Trackers. In the HDFS layer, the NameNode is the master that controls many DataNodes (Figure 2.2).

Each Hadoop cluster contains one Job Tracker. The Job Tracker is responsible for (a) querying the underlying HDFS for the block locations, (b) scheduling the tasks on the slave which is hosting the task's blocks, and (c) monitoring the successes and failures of the tasks. Equivalent to the slave workers are Task Trackers, who execute the tasks as directed by the Job Tracker. Each Task Tracker is set up with certain numbers of Map and Reduce slots respectively, i.e., a Task Tracker cannot have more than these numbers of Map or Reduce tasks running simultaneously. Each Map (or Reduce) task is a separate program that contains the user-defined map (or reduce) function and can be missing (Map-only jobs).

2.2 Hadoop scheduling

Hadoop runs several maps and reduces concurrently on each Task Tracker to overlap communication and I/O. At each heartbeat, the Task Tracker notifies the Job Tracker the number of available slots it currently has. The Job Tracker assigns tasks depending on jobs' priority, number of non-running tasks and potentially other criteria. The first version of Hadoop comes with a fixed FIFO scheduler, which was good for traditional usages such as Web Indexing or log mining, but rather inflexible and could not be tailored to different needs or different workload types.

Since bug report Hadoop-3412, Hadoop has been modified to accept pluggable schedulers, which allows the use of new scheduling algorithms to help optimize jobs with different specific characteristics. At the current stable version, Apache Hadoop is augmented with 3 pluggable schedulers, namely the default FIFO scheduler, the Fair scheduler, and the Capacity scheduler.

2.2.1 FIFO scheduler

The FIFO scheduler is the original scheduler that was integrated inside the Job Tracker. In FIFO scheduling, the Job Tracker simply pulls jobs from a single job queue. Although the scheduler's name suggests the prioritization of old jobs, the FIFO scheduler also takes into account jobs' priority. Algorithm 3 illustrates the comparator the FIFO scheduler uses to decide the order of jobs in the job queue.

Algorithm 3 int compare(Job j1, Job j2)

```
/*Compare priority first: favoring jobs with higher priority*/
if (j1.getPriority() < j2. getPriority()) then
  return 1;
else if (j1.getPriority() > j2.getPriority()) then
  return -1;
else
  /*Compare job start time: favoring older job*/
  if (j1.getStartTime() < j2.getStartTime()) then
    return -1;
  else if (j1.getStartTime() > j2.getStartTime()) then
    return 1;
  else
    /*Tiebraking by job id*/
    return (j1.getID() - j2.getID());
  end if
end if
```

2.2.2 Fair scheduler

The Fair scheduler assigns resources to jobs in a way such that on average over time, each job gets an equal share of the cluster's resources. "Short jobs" (jobs that require less time to finish) are thus able to access the CPU, and will finish intermixed with the execution of "long jobs" (jobs that require longer time to finish). The Fair scheduler was developed by Facebook, and aimed at providing better responsiveness for short jobs, which are the majority at Facebook's Hadoop usage.

The Fair scheduler uses a two-level scheduling hierarchy. At the top level, the Fair scheduler allocates task slots across $pools$ using weighted fair sharing i.e. the higher the weight a pool is configured with, the more resources it can acquire. Each user, by default, is assigned one pool, but organizations can also create special pools for practical reasons. At the second level, each pool allocates its assigned slots among jobs in the pool, using either FIFO with priorities (the same with FIFO scheduler) or a second level of fair sharing. In the second level of fair scheduling, each job is assigned a weight equal to the product of its (user-defined) priority and the number of tasks. Jobs with higher number of tasks generally need more time to finish, and will be assigned more task slots. Note that the Fair scheduler associates the number of tasks with the length of job, which means it assumes tasks have the same length. This assumption is not necessarily true since the length of a task differs between applications: even with the same amount of data, a complicated map (reduce) function will probably take more time to finish than a simple map (reduce) function.

While the FIFO scheduler does not employ preemption, the Fair scheduler uses Kill action to guarantee pools meet their *minimum share*. The minimum share represents a minimum number of slots that a pool is guaranteed to be given as long as it contains jobs, even if the pool's fair share is less than this amount (e.g., because many users are running jobs in the same cluster). The minimum share is automatically adjusted if the total minimum share of all pools exceeds the number of slots in the cluster.

The Fair scheduler gracefully handles short jobs, to which locality is of high importance. A short job will be prolonged a larger portion if is is executed non-locally, compared to a longer job, in which the data transfer time only accounts for a smaller portion. The Fair scheduler is equipped with the Delay technique, which allows the execution of a task on a Task Tracker to be postponed if the scheduler can find a local task for that Task Tracker. Postponed tasks are recorded so that if a task has waited for too long, it will be launched at the next free slot regardless of locality. This is to avoid starvation for tasks in a big cluster, where the chance for a task to have local data on a certain node is rather low. Algorithm 4 illustrates the Fair scheduler with Simple Delay scheduling.

2.2.3 Capacity scheduler

The Capacity Scheduler is designed to run Hadoop MapReduce as a shared, multi-tenant cluster. The central idea is that the resources are partitioned among tenants based on computing

Algorithm 4 Fair Sharing with Simple Delay Scheduling

```
Initialize j.skipcount to 0 for all jobs j.
when a heartbeat is received from node n:
if n has a free slot then
  sort jobs in increasing order of number of running tasks
  for j in jobs do
    if j has unlaunched task t with data on n then
      launch t on it n
      set j.skipcount = 0
    else if j has unlaunched task t then
      if j.skipcount ≤ D then
        launch t on n
      else
        set j.skipcount = j.skipcount +1
      end if
    end if
  end for
end if
```

needs. Unused quotas are divided equally among using tenants. This provides elasticity for tenants (usually organizations) in a cost effective manner.

Although the idea is rather similar to that of the Fair scheduler, the Capacity scheduler has specific characteristics. The Job Tracker is considered as a rather scarce resource, therefore the number of initialized jobs are limited, i.e., not all jobs are always initialized upon submission. Jobs are divided into queues and accessed sequentially in a manner similar to FIFO. Once a job starts, it will not be preempted by other jobs. Preemption is deemed an interesting functionality, but it has not yet been implemented.

2.3 Fault tolerance in Hadoop

Hadoop is designed to help process very large amounts of data using hundreds or thousands of machines. Since data centers usually consist of commodity hardware (Ananthanarayanan et al. 2011) and built incrementally (Rezaei & Mueller 2013), failure has become a norm rather an exception. In order to facilitate the correct execution of jobs in such environment, Hadoop needs to tolerate machine failures gracefully.

2.3.1 Task Tracker failure

As often found in many Master/Slave architecture, the Job Tracker in Hadoop also pings every Task Tracker periodically. If no response is received from a worker in a certain amount of time, Job Tracker marks this Task Tracker as failed. Any Map tasks completed on this Task Tracker that belong to unfinished jobs are reset back to *idle* state and therefore eligible for scheduling

again. Similarly, any map task or reduce task in progress on the failed Task Tracker is also reset to *idle* and becomes eligible for rescheduling.

When a map task is re-executed on another Task Tracker (because the first Task Tracker failed), all Reduce tasks are notified of the re-execution. Reduce tasks that have not read map output from the failed Task Tracker will read the data from the newly completed location instead.

2.3.2 Job Tracker failure

The Job Tracker in Hadoop remains as a single point of failure. All the activities inside a Hadoop cluster are monitored and controlled by the Job Tracker. If the Job Tracker fails, all activities are stopped. Hadoop does not implement any fault-tolerance mechanism related to Job Tracker failure. Users need to monitor the health of the Job Tracker on their own and restart Hadoop jobs if they desire in case of Job Tracker failure.

2.3.3 Speculative tasks

One of the common causes that prolong the execution of a Map Reduce job is a ”straggler”: a machine that fails to progress at the normal rate. Stragglers can arise for a number of reasons, such as contention in disk activities, or CPU overload. Hadoop employs a simple mechanism to alleviate the problem of stragglers: speculative execution. When a Map Reduce operation is close to completion, the Job Tracker schedules backup execution of tasks that are currently still running. The first copy to finish between original and backup task will be marked as Complete, while the other is killed to avoid duplication. Organizations have mixed opinions about the effectiveness of Speculative execution. While Google claimed that speculative execution helped boost performance by 44% (Dean & Ghemawat 2008), Yahoo! decided to disable speculation by default (Seo et al. 2009).

In this chapter, we have introduced the Hadoop framework, its architecture and the conceptual fault tolerance mechanism. In the following chapters, we will discuss this mechanism in depth, and provide a systematical analysis on the effect of failures to the performance of Hadoop.

II

Fault Tolerance Assessment in Hadoop

3 Hadoop fault tolerance mechanism

3.1 Detection of failed tasks

This section analyzes the mechanisms that Hadoop uses to guard against failures. A source-code analysis was performed on Hadoop version 1.2.1 (the version 1.x current stable release at the timet of this thesis). We notice a few changes compared to version 0.21.0 (Dinu & Ng 2012). Although the system is highly configurable, i.e., most of the parameters can be configured through configuration files without re-compiling the source code, we use the default values for simplicity and clarity.

Hadoop employs a static timeout mechanism for the detection of fail-stop failures. It keeps track of each Task Tracker's last heartbeat, so that should a Task Tracker does not send any heartbeat in a certain amount of time, that Task Tracker will be declared Failed. Each Task Tracker sends a heartbeat every 0.3s (many literatures have claimed that the heartbeat interval is 3 seconds, however here we use the value we found in the source code). The Job Tracker checks every 200s for any Task Tracker that has been silent for 600s. Once found, the Task Tracker is labeled as a failed machine, and the Job Tracker will trigger the failure handling and recovery process

3.1.1 Declaring Map output lost

Hadoop allows quick detection of Map task failures, in the form of Map Output Lost. The loss of a Task Tracker makes all map outputs stored on that machine inaccessible to Reduce Tasks. A Map Output is declared lost if the Job Tracker receives enough notifications from Reduce Tasks that they cannot obtain the map output. In fact, the map output is recomputed if the number of notifications that a Map Output is unavailable is higher than 3, or 50% of the total number of Reduce tasks in the job. The double condition allows Hadoop to avoid prematurely false conclusion in a system with high number of Reduce Tasks, as well as eager false conclusiosn when the amount of running Reduce Tasks is small.

A notification is, by default, sent immediately if a read error occurs while the Reduce task R is copying map output from Map task M. User can also change the default true Boolean parameter *"mapreduce.reduce.shuffle.notify.readerror"* to false, so that a notification is only sent after every 10 failed retries. A back-off mechanism is used to dictate how soon a node is contacted again after a connection error. The penalty time is calculated as follows:

$$\textit{Penalty (in seconds)} = 10 * ((1.3) \hat{} (F_M^R))$$

where (F_M^R) is the number of times the Reduce task failed to fetch Map M's output.

Note that with the above-mentioned back-off mechanism, a Map output will be declared Lost after roughly $\sum_{i=1}^{10} 10 * ((1.3)^i) = 554$ seconds from the first attempt. Since all the values in the above formula are hard-coded, this early Map Output Lost detection mechanism will not be useful if the expiry time is set to a smaller number.

3.1.2 Declaring a Reduce Task faulty

A Task Tracker considers a reduce task to be faulty if the task fails too many times trying to copy map outputs. There are 2 events that can trigger the declaration of a failed reduce task: 1) the Reduce task has failed more than both 30 times and 10% of the amount of Map tasks in the jobs, or 2) the Reduce task is "too faulty" to continue i.e., it has failed so many times. The 2^{nd} condition is only checked whenever there are 10 failed attempts in fetching a certain map output, and three conditions need to be simultaneously true for the 2^{nd} event to be triggered. First, there must be too many unique unavailable Map outputs: either more than 5 or equals to the number of not-yet-copied Map outputs. Second, more than half of the attempts made by Reduce Task R have failed. Finally, the Reduce task has stalled for more than half the amount of time during which it has made progress.

If a Reducer happens to meet the conditions for the 2 events above, it will be considered as Failed, and will be returned to the Job Tracker for another scheduling opportunity. Since all failed Reducers are incomplete Reducers, failed Reducers will be sent to another queue before getting back to the normal failed queue. Details will be discussed in the next section.

3.2 Failure handling and recovery

Tasks that were running on the failed Task Tracker will be restarted on other machines. Map tasks that completed on this Task Tracker will also be restarted if they belong to jobs that are still in progress and contain some reduce tasks. Completed Reduce Tasks are not restarted, as the output is supposedly stored persistently on HDFS.

Hadoop failed tasks are either added to a queue for failed tasks (Map task), or back to non-running queue (Reduce task). Both the queues are sorted in the order of failed attempts: tasks with higher number of failures are positioned at the beginning of the queues. In case tasks have the same number of failed retries, the task ID is used to tie-break.

Hadoop distinguishes between completed failed tasks and running failed tasks. While the completed failed tasks are added directly to the failed queue as mentioned above, running failed tasks are added to different lists called the Task Clean Up lists. This is because failed running tasks might leave some remnants of corrupted data on the local storage of the Task

Tracker, and they can cause trouble to later executed attempts of the same task. In order to avoid this, the Job Tracker will launch a clean up task (of the same type) for every item in the Task Clean Up list. Only after a Task Clean Up task finishes is the failed task moved to the failed queue.

Hadoop prioritizes failed tasks to run first over pending (non-running) tasks when it comes to assigning new tasks. The aim of this decision is to quickly discover any jobs' "internal failures". Hadoop jobs' "internal failures" are failures that are specific to a certain job and cannot be tolerated by re-execution mechanism. One example of internal failures is corrupted input files. While each task consumes a certain amount of resources (disk space, computational cycles, memory...) faster discovery of internal failures allows Hadoop to quickly purge those failed jobs and release the resources for other waiting jobs, hence achieving better utilization of resources.

Although Hadoop pays much effort to achieve locality for Map tasks in normal situations (it tries to assign as many local tasks as possible, while only assigingn at most 1 non-local task regardless of the number of available slot at each time), it completely ignores locality when it comes to failed tasks. As long as there are failed (Map) tasks, any Task Tracker that has free slots will be assigned the maximum number of tasks that it can handle. This leads to degradation in the performance of Hadoop when there are many failed tasks, as the number of non-local tasks might become very high.

3.3 Speculative execution

Besides the physical failures when a node stops functioning, Hadoop has speculative mechanism to guard against another type of failures, called time failure. Time failure occurs when a task fails to progress at an average normal rate. In an heterogeneous environment, this is likely to happen at nodes with lower capacity. In cloud environments, due to the fact that some physical nodes can host an unknown number of virtual machines, some of those virtual machines might suffer from resource starvation, which in turn leads to slow progress of tasks on those machines.

In either situation, those "straggler" tasks prevent jobs from finishing and can lead to serious underutilization of resources. Hadoop avoids these situations by running a speculative algorithm, which attempts to improve jobs' running time by duplicating under-performing tasks. When the Map/Reduce task pool is empty and there are free slots in the system, the Job Tracker tries to identify the slowest running tasks, and accordingly launch a copy of these tasks on other machines.

In the currently examined version of Hadoop (1.2.1), a simple algorithm is employed to find tasks to execute speculatively. If the progress score of any task is less than the average of its category minus 0.2, and the task has started for more than one minute, it is considered for speculative execution. There are a few points worth noticing about the current implementation

of this algorithm:

1) Speculative tasks will not be launched if there are other tasks in the job (failed and non-running tasks).

2) The average progress score of a category (Map or Reduce) is calculated by the total progress score of all tasks (including completed and non-running tasks) in the system divided by the total number of tasks in that category. A Map task progress score is calculated by the fraction of input that it has processed. For a reduce task, the execution is divided into three phases, each of which accounts for $\frac{1}{3}$ of the score:

- The Shuffle phase, when the task fetches map outputs
- The Sort phase, when map outputs are sorted by key
- The Reduce phase, when the user-defined function is applied to the list of map outputs within each key

In each phase, the score is the fraction of the data processed. For example, a task that has fetched 50% of the map outputs (still in Shuffle phase) has a progress score of $0.5 * \frac{1}{3} = \frac{1}{6}$. A task that has "reduced" half the number of map output keys has the progress score of $\frac{1}{3} + \frac{1}{3} + 0.5 * \frac{1}{3} = \frac{5}{6}$. There is no "fraction of data sorted" in the Sort phase, therefore the progress score of Sort phase has only 2 values: 0 and $\frac{1}{3}$.

Since the average progress score includes already completed tasks, the speculation algorithm does not accurately address the stragglers among currently running tasks. In fact, when the job comes near the end (most of the tasks have finished), a newly launched task always has very low progress score compared to the average, and is often prone to be executed speculatively.

3) The age of a task is calculated from the moment it is initialized, rather than the moment when it is launched. This is very close to the initialization time of the job, when all the metadata for the tasks are constructed. It means a task can have its speculative copy launched only a few seconds after the launch of the original, providing that the job has been running for more than 60s, and the average progress score is high enough to trigger speculation. This mechanism might sound strange at first, but it probably evolves from the fact that the dominant use of Hadoop is for short jobs with tasks that can finish in less than 60s (Chen et al. 2011).

This speculative algorithm has a potential drawback of excessive launching of tasks in the last "wave". A "wave" in Hadoop is a set of tasks that are launched and finish at similar point in time. A job J with M map tasks and R reduce task running in a cluster with S_M map slot and S_R reduce slot has potentially at least $\lceil \frac{M}{S_M} \rceil$ map wave and $\lceil \frac{R}{S_R} \rceil$ reduce wave. If the number of waves is large enough that the average progress score is higher than 0.2, and the job has run for more than 60s, then any tasks from the last wave will be executed speculatively.

To guard against this excessive launching phenomenon, the default Hadoop scheduler (FIFO) is equipped with a mechanism to limit the capacity of each Task Tracker depending

on the number of unfinished tasks. Each Task Tracker cannot assign the number of tasks more than the number of its capacity multiplied by the ratio between the number of unfinished tasks and the cluster capacity (called the load factor). In a cluster with 15 nodes, each has 2 Map slots (so the total cluster capacity is 30 slots), a job J with 100 Map tasks will run in 4 waves. In the last waves, there are $(100 - 3 * 30) = 10$ tasks left, and the load factor of the cluster would be $\frac{10}{30} = \frac{1}{3}$. Each Task Tracker will then have its new capacity of Min ($\lceil \frac{1}{3} * 2 \rceil$, 2) = 1 slot. After assigning 10 map tasks, there are only $(1 * 15 - 10 =)5$ slots for speculative execution, instead of Min (*running tasks, free slots in the cluster*) = $Min(10, 30 - 10) = 10$ speculative tasks.

This guarding mechanism seems to be omitted from the other 2 schedulers (Fair scheduler and Capacity scheduler). However, organizations have reported to turn off the speculative execution, as they mostly degrade the performance of Hadoop rather than improving it (Seo et al. 2009).

3.4 Life cycle of a Task

To conclude the failure handling process of Hadoop, we summarize the life cycle of a task as in figure 3.1:

Initially, all tasks are in pending (non-running) state. The Job Tracker assigns tasks to the Task Trackers as free slots become available. Tasks that are assigned move to Running state, and eventually to Completed state. Should a task fails; it is marked as Failed and sent back to the Job Tracker waiting for re-launch. In the course of a task, the Job Tracker might declare that task is a straggler, and launch a copy of that task on a different Task Tracker. The first finishes is marked as Completed, while the other is marked as Killed.

In the next chapter, we will present a systematic analysis of Hadoop performance under failures. We consider two type of failures: stress - a mild variant, and actual Task Tracker failure. The experiment results will give insight about actual Hadoop's behavior, which will help us in the real problem behind the degradation of performance under failures.

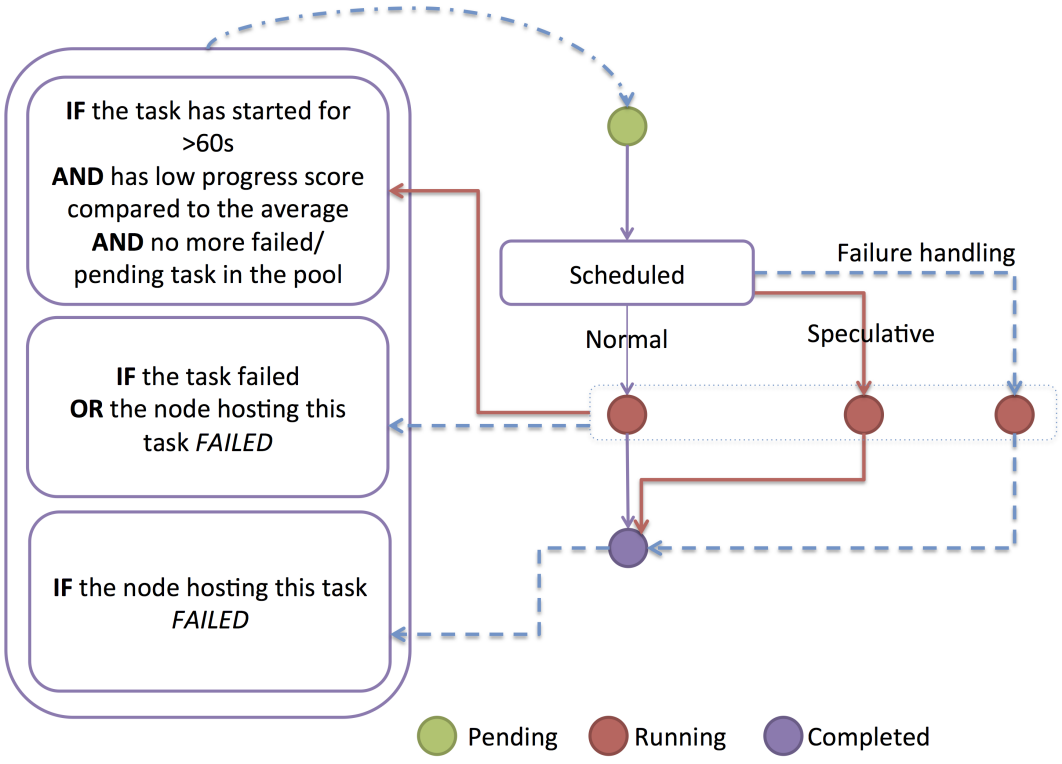

Figure 3.1: The life cycle of a task in Hadoop

4 Systematic Assessment of Hadoop Performance Under Failures

This chapter presents a systematic performance evaluation of the 3 schedulers that come in the Hadoop release package, namely the default FIFO scheduler, the Fair scheduler and the Capacity scheduler. We conduct a series of experiments to assess the impact of stress (a mild variance of failure) and failures in the execution of multiple applications in Hadoop. The metrics that we are interested in include average execution time, individual job execution time and data locality. In some experiment, we also concern about the execution of speculative tasks, as these tasks can either speed up the execution time, or wastefully compete for resources.

4.1 Experiment settings

Cluster setup

We conduct our experiment on (Grid5000). Grid5000 is a large-scale experiment testbed, which provides the research community with highly configurable infrastructure. We perform our experiment on the Rennes site with 21 nodes: 1 Master node running the Job Tracker and Name Node processes, and 20 Slave nodes running Task Tracker and Data Node processes (all the other experiments, unless states otherwise, will always have 1 dedicated node as the Master, and many other slave nodes). Each node is equipped with 2.5Ghz Intel Xeon L5420 CPU with 8 cores, 16GB Memory and one 320GB SATA II hard drive. Nodes are connected using Gigabit Ethernet cables.

Hadoop setup

On the Grid5000 testbed described above, we deployed and configured a Hadoop cluster using the 1.2.1 stable version. Each node is configured with 6 Map slots and 2 Reduce slots (1 slot per core on average). The number of Reduce tasks is set at 40 tasks. The HDFS replication is set at 2, and the block size is set at 128MB. To better cope with small workload, we set the expiry time (the amount of time a Job Tracker will wait before declaring a Task Tracker failed if there was no heartbeat from that Task Tracker) to 60 seconds instead of the default 600 seconds. Besides that, all the other configurations are kept at the default value.

Benchmarks

We use WordCount and TeraSort applications from the official Hadoop release's example package. Data is extracted from PUMA data set (Ahmad et al. 2012) to create different input sizes.

Jobs

We run a series of 6 jobs (from the combinations between applications and input sizes. Each job is submitted 10 seconds after each other, and job types are intermixed so that we have mixed set of long, medium and short jobs of different characteristics.

#	Application	Input size
1	Terasort	31GB
2	WordCount	11GB
3	TeraSort	2.8GB
4	WordCount	30GB
5	TeraSort	12GB
6	WordCount	1.1GB

Table 4.1: List of jobs and their input size used in the experiment, in the order of submission

4.2 Hadoop under stress

In this experiment, we want to understand Hadoop behavior under the circumstances when one of the nodes in the cluster gets stressed. In a shared cluster, nodes may run many different processes at the same time for utilization reasons. It is not uncommon to have some of the nodes having more running processes than other. The situation becomes even more significant when virtualization is employed. Since each of the virtualized nodes has to compete for resources from the physical machines, spontaneous congestions can happen frequently.

Stressing

We launch some extra processes (**while** loop for CPU stress & **dd** command for IO stress) on one of the nodes from the Slave set. These processes act as the stress factor on that node. Each process lasts for 30 seconds, and is launched interleaved with each other. Between any 2 stressing processes there is a gap of 30 seconds of stress-free to simulate sporadic stress condition. The first stress process is launched 60 seconds from the beginning of the experiment.

Result

Total execution time and data locality

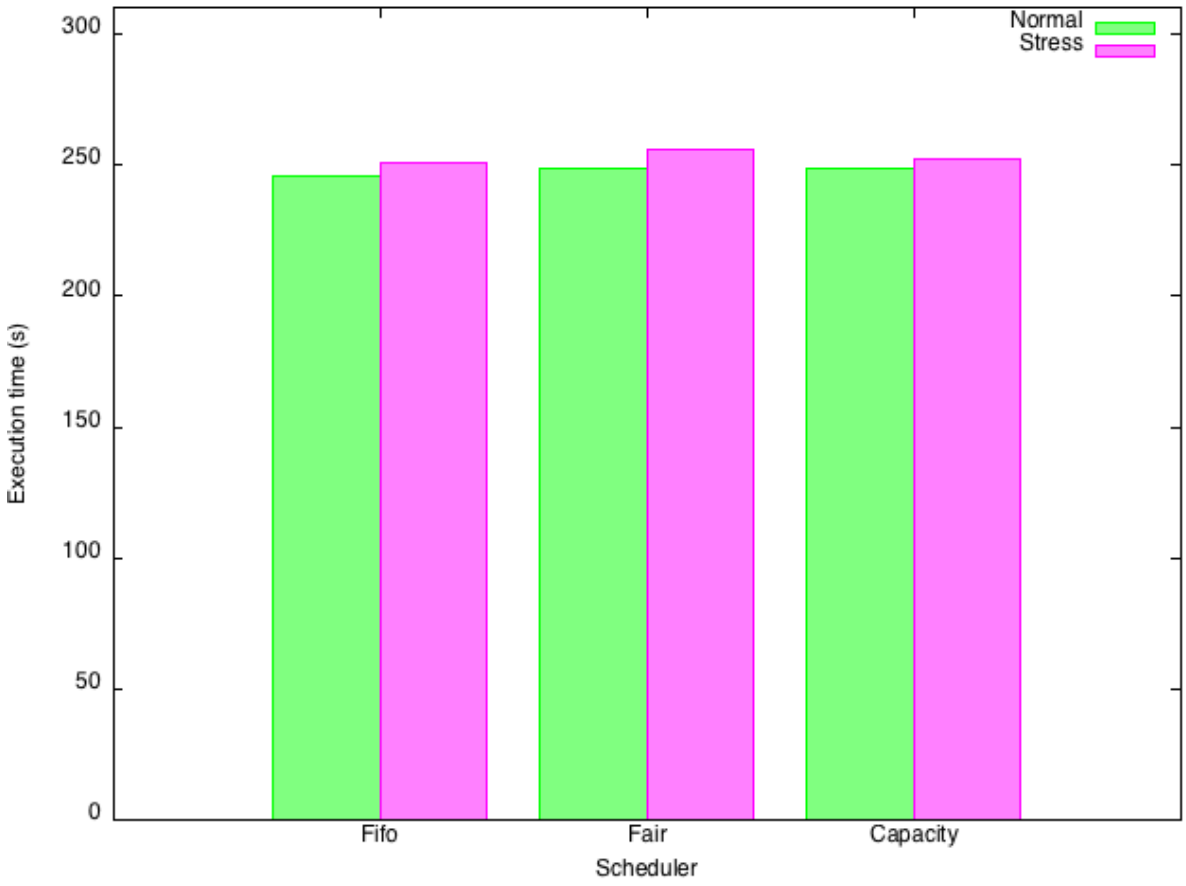

Figure 4.1: Total execution time under 3 different schedulers

Figure 4.1 presents the total execution time of the experiment test under 2 different scenarios. In Normal situation where there are no stresses or failures, all the three schedulers demonstrate similar performance in term of finishing time: in average, Fair and Capacity schedulers both finish after 249 seconds, and FIFO finishes after 246 seconds. In Stressed condition, the three schedulers start to show some differences. FIFO scheduler once again finishes first with 4 seconds of degradation. Capacity scheduler also suffers the same amount, while Fair scheduler gets prolonged for 7 seconds (finishes after 256 seconds) and becomes the slowest among the three.

Figure 4.2 presents the locality (the ratio between the number of locally executed tasks and total number of tasks) of the 3 schedulers under different situations. Fair scheduler demonstrates the highest locality as expected, thanks to the Delay technique. FIFO scheduler and Capacity scheduler demonstrate similar performance in this aspect, and significantly lower

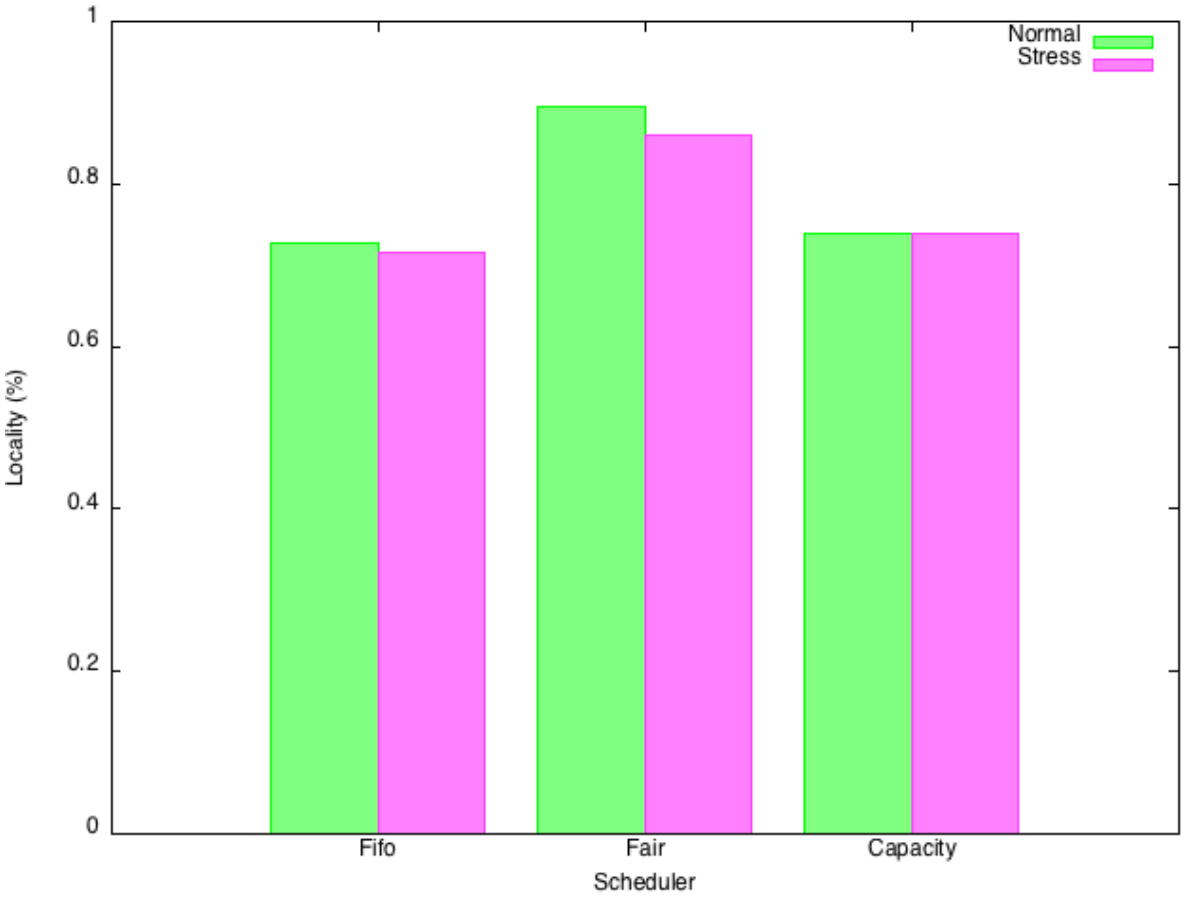

Figure 4.2: Data locality of the 3 schedulers under different situations

compared to Fair scheduler. The difference was not reflected in the total execution time, as network bandwidth is rather abundant in our experiment: all the nodes are in the same rack, and besides Hadoop, there were no other users' running processes that involves network during the course of the experiment. The result of this abundance is that even a chunk of 128MB can be quickly transferred between nodes without much delay. This setting is normally not true in a multi-purpose, multi-tenant cluster: network bandwidth is generally considered as a scarce type of resource (Rao et al. 2012).

Under stressed conditions, all the three schedulers witness degradation in data locality. This is because tasks on the stressed node are likely to become stragglers, and receive speculation from Hadoop. Although Hadoop also tries to provide locality for these speculative tasks, if the original copy was launched on a node with data, then the second launch will have less chance to be local.

Another effect of stressing a node is that the increase in number of speculative tasks is accompanied with more waste in the resources. Speculative tasks are launched, but they eventually are killed when the original tasks finish. We can see more about this phenomenon in

figure 4.3 and 4.4

Speculative execution

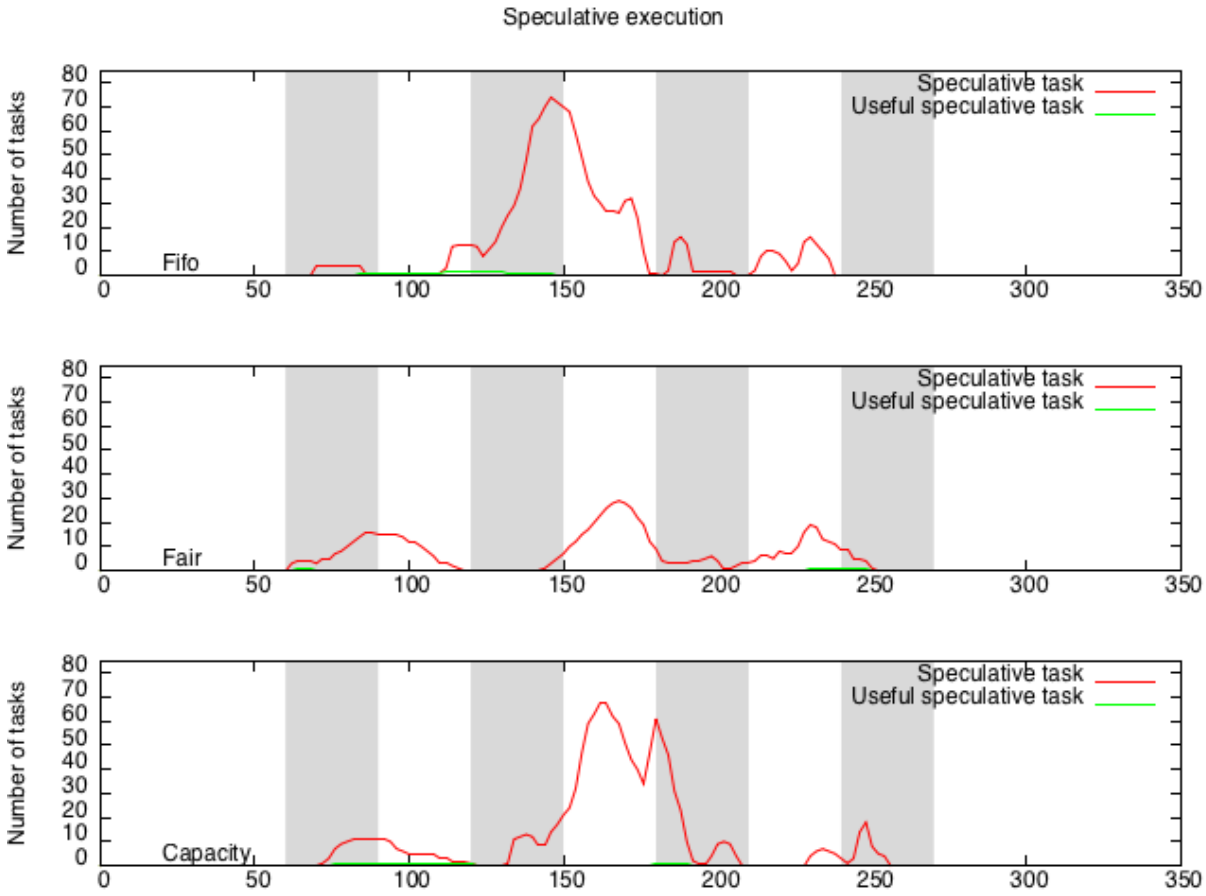

Figure 4.3: Speculation execution in Normal situation of the 3 schedulers

Figure 4.3 shows the number of running speculative tasks at different points in time of the Normal scenario. The red color depicts the total number of running speculative, while the green color, noted as "Useful speculative tasks", illustrates those speculative tasks that actually finish before the original one, and hence, are meaningful. Note that the figure shows the number of running speculative tasks at different points in time during the course of the experiment: tasks are generally accounted for more than once.The shaded region marks the duration during which stress processes are running. Although there was no stress processes during Normal scenario, we keep the shaded color for the ease of comparison with stressed scenarios. Figure 4.3 shows that the speculation mechanism in Hadoop is not very effective in this scenario: most of the speculative tasks are actually wasted.

Figure 4.3 also compares the difference in how each scheduler chooses tasks to execute speculatively. The FIFO scheduler and the Capacity scheduler show a similar pattern in specu-

lative execution: when running Capacity without concerning about share, the **default** queue in Capacity is basically a FIFO queue. However, there are still some differences in the number of concurrently running speculative tasks at each moment: the FIFO scheduler has the padding mechanism to slow down the rate of assigning tasks while the Capacity scheduler does not, and this difference alters the number of occupied slots in the last wave of a job. Fair scheduler has less speculative tasks compared to the other two schedulers. In Normal scenario without stress and failure, where speculative tasks are mostly wasted, This means Fair scheduler achieves better utilization of resources compared to the other 2 schedulers..

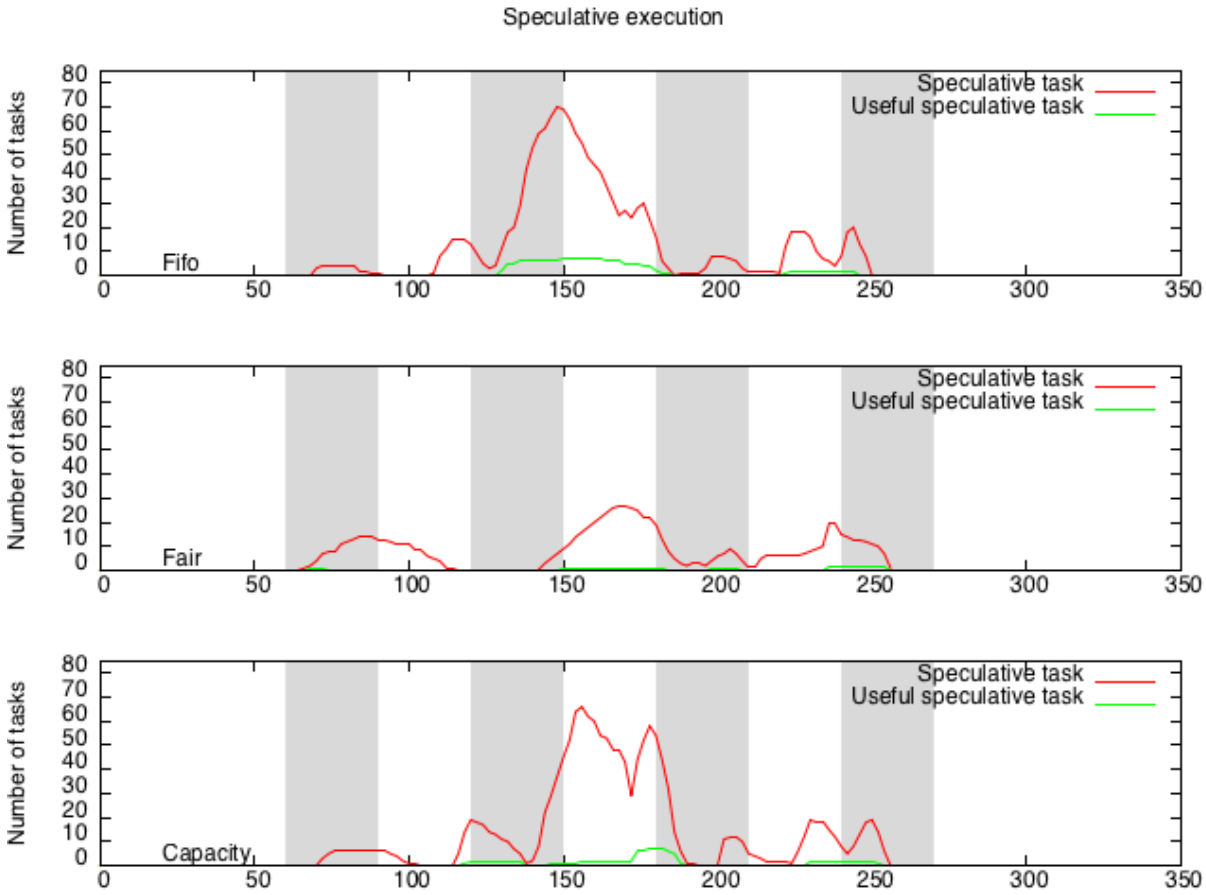

Figure 4.4: Speculation execution in Stressed situation of the 3 schedulers

Figure 4.4 illustrates the speculative execution of the 3 schedulers, but in the Stress scenario. Both the number of speculative tasks, as well as the number of useful tasks increase (though slightly), showing the effect of stressing processes on Hadoop. Once again, the FIFO and Capacity schedulers show similar behavior, while the Fair scheduler still introduces fewer number of speculative tasks compared to the other two. The occurrence of speculative tasks is generally delayed a few seconds: this is because stressed nodes take more time to finish their tasks, and therefore delay the last wave for a short period of time. We can also observe more "useful speculative tasks" (green tasks): the speculative mechanism proves to be useful,

though limited.

4.3 Hadoop under failure

Failure is a part of everyday life, mainly due to large scale and shared environments. Earlier studies have reported that failures are frequent in large-scale distributed system (Dean 2009) (Schroeder & Gibson 2010) (Pinheiro et al. 2007) (Chandra et al. 2008) (Schroeder et al. 2009). Some failures can be quickly recovered, but some are disastrous. For instance, on Thursday April 21st, 2011 (Dailymail 2011), a outage with Amazon Cloud occurred and resulted in crashing major businesses and websites including the New York Times, Reddit, Quora and Hootsuite etc. Some of these sites went offline for up to 12 hours, and they suffered performance degradation for at least one week after the outage. Even worse, Amazon was not able to recover some of the customers' data even after many ultimate efforts.

Failures are a major concern during run-time execution of Hadoop applications. Reported experiences indicate that both transient (i.e., fail-recovery) and permanent (i.e., fail-stop) failures are prevalent, and will only worsen as the amount of computation increases. For example, Google reports 5 average worker deaths per MapReduce job in March 2006 (Dean 2006), and at least one disk failure in every run of a 6-hour MapReduce job with 4,000 machines (Czajkowski 2008).

We evaluate Hadoop's performance when there are failures in the system. To mimic the failure, we simply kill the Task Tracker process on one of the slave nodes. Failure injection time is set at 80 seconds since the beginning of the experiment (although other failure injection times result in different levels of degradation; in this section we only present one representative case). The Task Tracker process is never restarted (fail-stop). The Data Node process is kept running, so that no re-replicate activities occur. Data is still accessible from that node, but there will be no more tasks to be launched from the same machine. Note that since we set the replication factor to 2, the maximum number of failures that Hadoop can tolerate is 1, as any number more than this will result in loss of data and render the job failed.

The default expiry time (the amount of time after which a task tracker will be declared "lost" if there was no heartbeat) is 600 seconds. This value is considerably large compared to the job size (the largest job in this experiment only takes around 200 seconds to finish). We change this value to 60 seconds for a more timely reaction to failure.

Result

Total execution time and locality

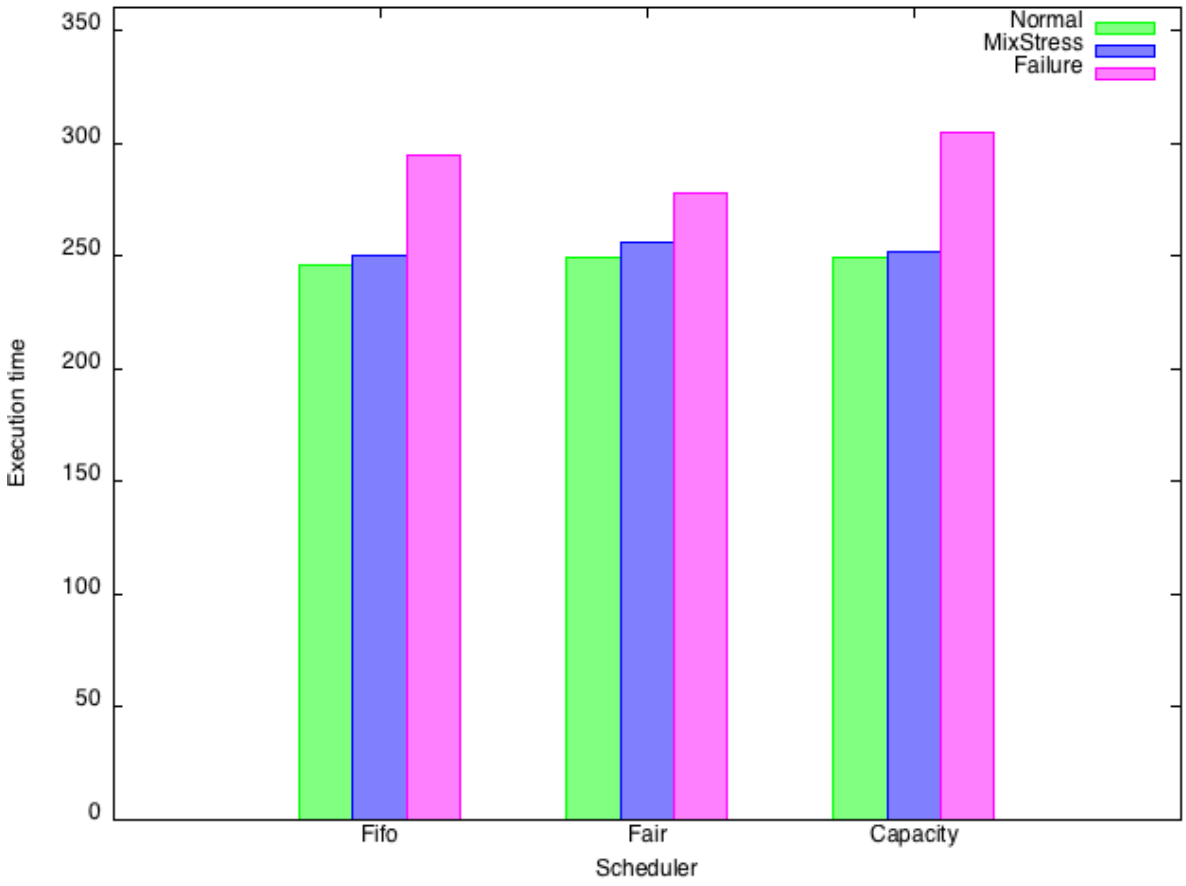

Figure 4.5: Total execution time of Hadoop in 3 scenario: Normal, Mix Stress and Failure

Figure 4.5 presents the total execution time in 3 different scenarios: Normal, Mix Stress and Failure. The Mix stress scenario is included for better comparison. As we can see from Figure 4.5, failures prolong the execution of Hadoop jobs by a significant amount of time as much as 56 seconds (roughly 22.5%) in case of Capacity scheduler. Fair scheduler appears to suffer the least: its execution time is prolonged for only 29 seconds (10.4%), and it also finishes the fastest among the three schedulers under failure (278 seconds compared to 294 seconds for FIFO, and 305 seconds for Capacity scheduler).

The small degradation of Fair scheduler can be explained by the fact that Fair scheduler allows multiple jobs to share the cluster proportional to their job sizes. Each job now has less resources at a point in time compared to that in FIFO. When a failure occurs, since jobs have been progressing slower than that in FIFO, they can overlap the useful effort (to finish other tasks) with the expiry time (failure detection window). Besides since the failed node only ac-

counts for 5% of the total number of slots, there may be a chance that none of the tasks on the failed node belongs to a job (especially Reduce tasks). This job will not be blocked and can be finished even during the failure detection window. This helps limit the impact of a node failure on jobs under the scheduling of Fair scheduler.

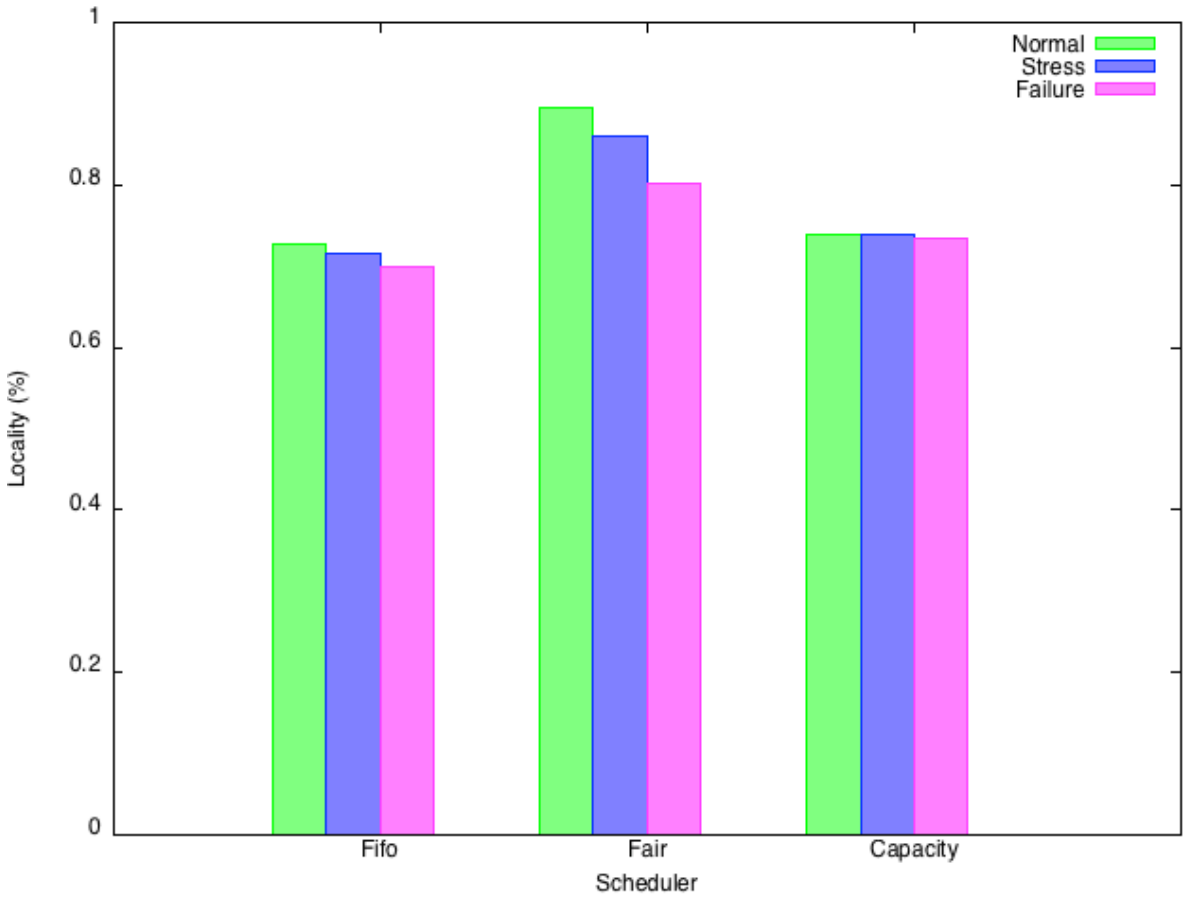

Figure 4.6: Data locality of the 3 scheduler under Normal, Mix Stress and Failure scenario

Figure 4.6 shows the percentage of locally executed tasks over the total number of tasks in the 3 different scenarios: Normal, Stress, and Failure. Fair scheduler still enjoys the highest number of locality, even though this number is decreasing. FIFO and Capacity scheduler shows some degradation, though this degradation is rather small compared to Fair scheduler (3% and 1% compared to 9%). To explain this phenomenon, remember that Fair scheduler was designed based on the assumption that most tasks are short and therefore, nodes will release slots quickly enough for other tasks to get locally executed. However, in case of failure, the long failure detection time (expiry time) creates the illusion of long-lasting tasks on failed nodes. These "fake" long tasks break the assumption of Fair scheduler, leading to higher degradation.

Speculative execution

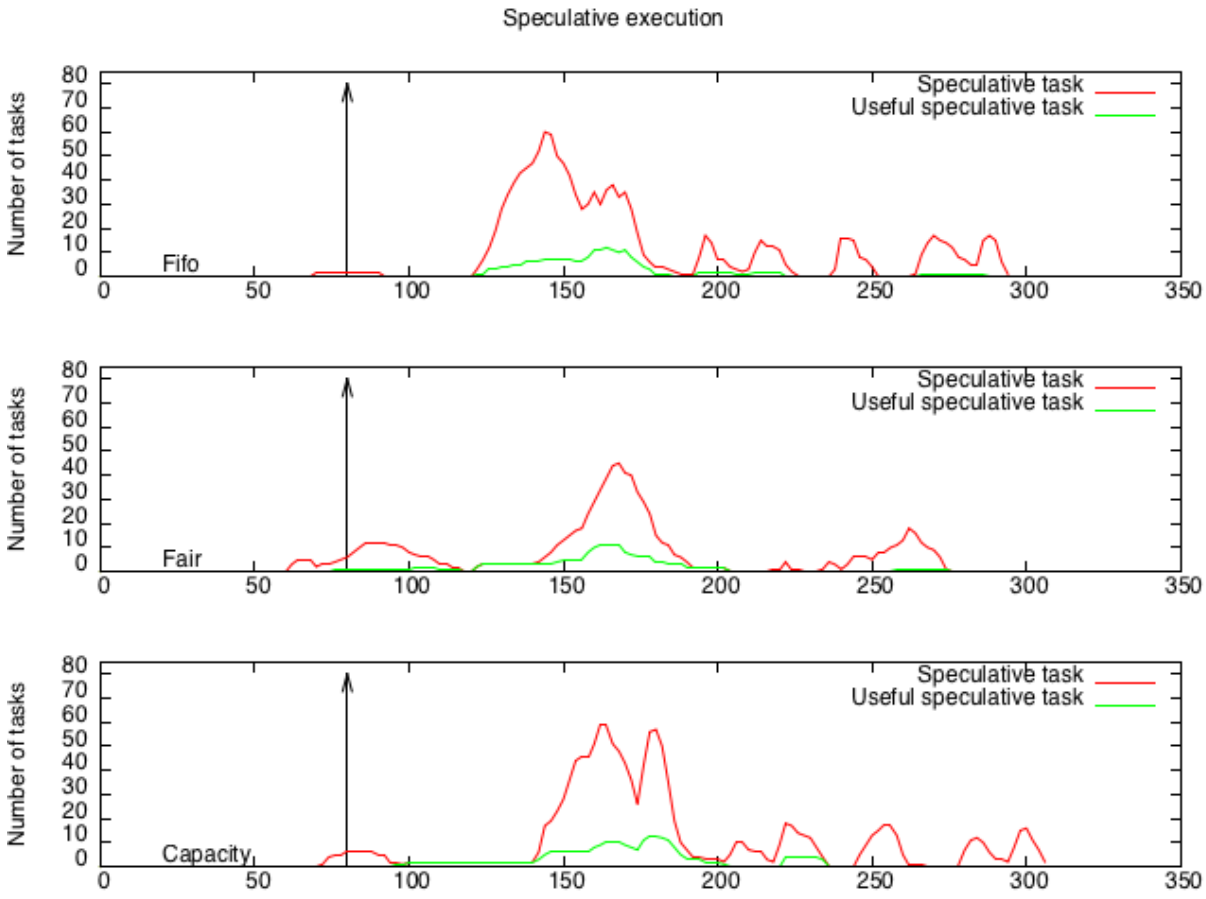

Figure 4.7: Speculation execution in the Failure scenario of the 3 schedulers

Figure 4.7 demonstrates the speculative execution of the 3 schedulers under failure. The number of "useful speculative tasks", i.e., those speculative tasks that finish before the original ones, increased compared to Normal execution (Figure 4.3). This is because during the 60 seconds between the failure of task tracker and its failure discovery, some speculative tasks were launched and finished successfully. Other than this increase in the number of successful speculative tasks, all other observations remain the same: The FIFO and Capacity schedulers show similar patterns in speculative execution; the Fair scheduler has the least number of speculative tasks among the three.

Although rather obvious, we also include a situation when the default expiry time (600 seconds) is used. The total execution time is by far longest in this default setting. This is because there are some already finished tasks on the failed node when it failed. These Map outputs will either have to wait until the node is declared Failed, or there are enough "Failed to fetch Map output" notifications in order to be re-executed. The longer the expiry time, the longer the job is blocked, and therefore total execution time becomes longer. The Speculation

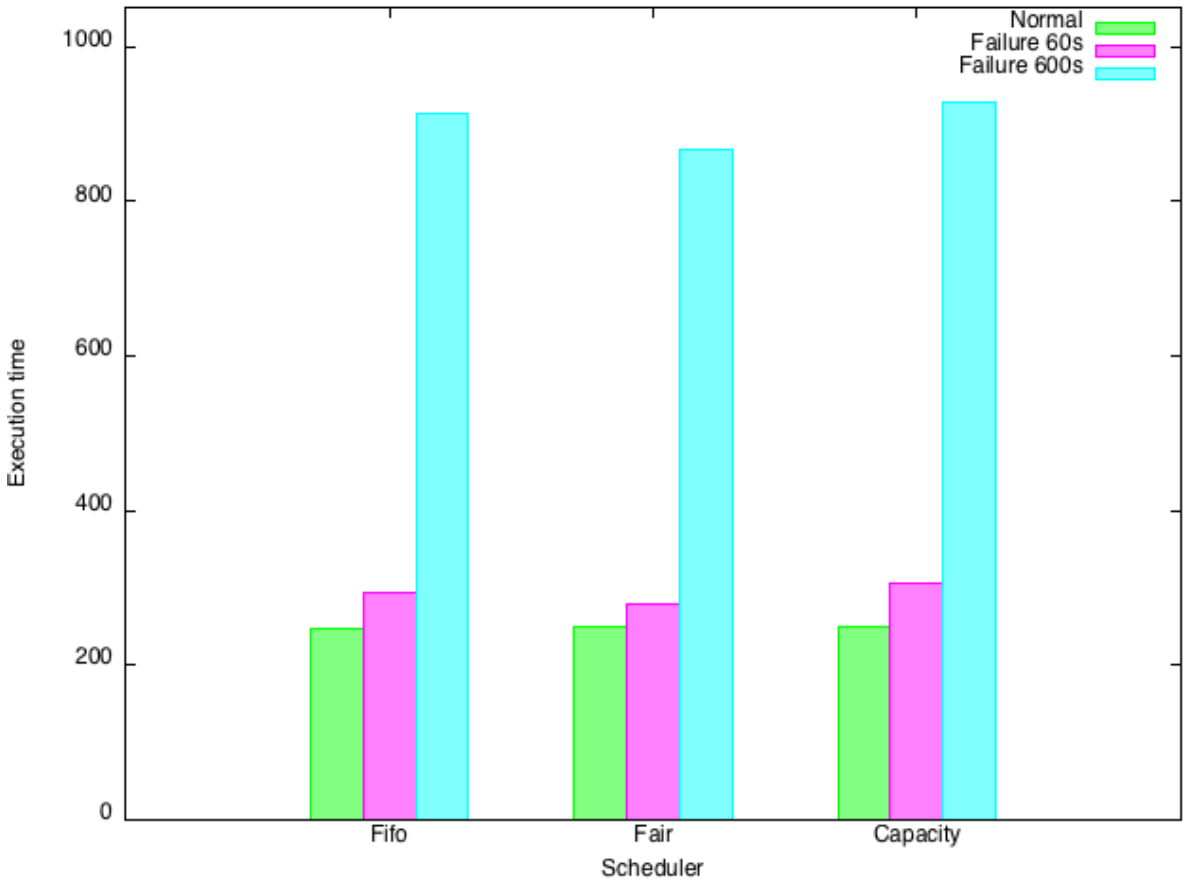

Figure 4.8: Total execution time in 3 different situations: Normal, Failure with 60s of expiry time, Failure with 600s of expiry time

mechanism does not improve the situation, as it can only speculate on currently running tasks.

This chapter discussed the effect of failure to Hadoop performance. In the next chapter, we will briefly introduce some related works in the scope of fault tolerance in Hadoop.

5 Related Works

Failures significantly increase the execution time of Hadoop applications. There are two factors that contribute to this degradation: the timeout in failure detection, and the failure handling mechanism that Hadoop employs.

Hadoop uses a fixed value for the expiry time regardless of the workload. The default value is 10 minutes, which is of disadvantages to small jobs. Although this timeout would be the same in value, the toll for larger jobs (jobs that take longer time to finish) is relatively smaller. Besides, during the period of failure detection, larger jobs might have other un-finished tasks to run, so the delay time overlaps with other tasks' execution time and therefore, the penalty can be reduced even more. However, it is not the case for small jobs. There has been effort in trying to adaptively adjust the expiry time. (Zhu & Chen 2011) introduces a job size estimator in order to adjust this value according to job size. Smaller jobs will benefit greatly from this adaptive expiry time value in case of failure.

Other efforts to improve Hadoop performance under failure includes attempts to protect intermediate data. Upon failure, intermediate map output that is stored on the failed machine becomes inaccessible for non-finished Reduce tasks, and those tasks need to be re-executed. (Ko et al. 2010) develop the Intermediate Storage System (ISS) to keep the intermediate data safe. ISS uses an asynchronous rack- level selective replication mechanism, which minimizes the effect of run-time server failures on the availability of intermediate data. Another attempt is from (Bicer et al. 2010). Their system design can be seen as a check point based approach, where the reduction object is periodically copied to another node. Therefore, if one worker fails, its reduction value exists on another node.

Preserving intermediate data can be promising in case of failures, but it induces a very high cost in term of resources (storage space) as well as time (replicating intermediate data to a number of machines is costly). It also affects the resources utilization, when intermediate data is generally only useful during the course of the job, and will be discarded after the job finishes.

There exists another problem regarding the failure handling mechanism of Hadoop that often gets unnoticed. When a task is declared failed, it gets "special treatment" in the manner that, failed tasks will be launched as soon as any slot becomes available, regardless of data locality. In a cluster where Data Node and Task Tracker processes co-reside, a machine failure will reduce the replication factor for those data splits originally on that node. Providing that Hadoop tries its best to provide locality for tasks in normal situation, it is likely that the failed tasks will have one less machine to run locally, which in turn leads to lower locality in general.

Providing locality for tasks is crucial for performance of Hadoop in large clusters because network bisection bandwidth becomes a bottleneck (Dean & Ghemawat 2008). Besides, since most of the Hadoop usage is for small jobs (jobs with small number of map tasks), it is difficult for a small job to obtain slots on nodes with local data. Data then has to be transferred through the network, which might significantly increase the execution time if network bandwidth is scarce. Providing locality for these jobs will greatly increase the performance of Hadoop in term of time and resource preserving.

Unfortunately, achieving high locality is not easy. Zaharia et al. (Zaharia et al. 2010) introduces the Delay technique inside the Fair scheduler to improve locality of tasks. Instead of strictly following the order of jobs, the Fair scheduler allows behind jobs launch their tasks first if the head-of-line job fails to launch a local task. However, the Fair scheduler relies on the assumption that tasks are mostly short and slots are freed up quickly. In case of long tasks that occupy the slots, a node may not free up quickly enough for other jobs to achieve locality.

Map locality is important, but Reduce locality can also improve Hadoop performance. During the Shuffle phase of a Reduce tasks, a large amount of data is transferred through the limited network. Traditional Hadoop tries to cope up with this problem by providing the Combiner function, which is basically a Map-side Reduce function. Combiner indeed helps reduce the amount of Map output, but they cannot help with the skew in data: some Map tasks have more data for a certain Reduce tasks, and less for others. Providing Reduce locality means placing the Reduce tasks on nodes that have the most data for these Reducers. However, providing Reduce locality poses the same problem above: slots may not be available.

In the effort to overcome the above-mentioned problems, we propose preemption. Preemption allows a task to quickly release the slot for more urgent tasks. Locality can be assured with the employment of preemption. Also, preemption allows the scheduler to have better control on the resources (i.e., the task slots). Shorter tasks can preempt a longer one to achieve fast response time.

Preemption in Hadoop

To our knowledge, there has been not much work aiming at providing the preemption feature for Hadoop.

- (Liu et al. 2012) introduces a Preemptive Deadline Constraint Scheduler (PDCS), which aims at minimizing the total completion time of jobs under deadlines. Traditional non-preemptive schedulers have to wait for previously assigned jobs' completion or halt. This delays the execution of production jobs, sometimes render them violating their deadlines. To avoid this, PDCS employs the Hadoop built-in preemption mechanism (kill) to provide slots for near deadline jobs. Upon submission, jobs are checked whether they can finish under its deadline or not using estimation. Jobs are then scheduled if the number of available slots meets the requirement. Otherwise, the scheduler would determine whether these jobs are legal to preempt the slots that have been already allocated.

- (Wang et al. 2013) introduce the Fair Completion Scheduler (FCS) that supports Reduce Task pre-emption. A long Reduce task would occupy the reduce slot, and significantly increase the completion time of shorter jobs. By check pointing the Reduce task, the reduce slot can be passed on to a different shorter job. After the short job finishes, the long Reduce task picks up the work from where it was left off, and continue until the end.

- (Pastorelli et al. 2014) propose to leverage the already available POSIX signals such as `SIGTSTP` and `SIGCONT` to suspend running tasks. In Hadoop, Map and Reduce tasks are regular Unix processes running in child JVMs spawned by the TaskTracker. This means that they can safely be handled with the POSIX signaling infrastructure. The state of tasks is implicitly saved by the operating system, and kept in memory. If not enough physical memory is available for running tasks at any moment, the OS paging mechanisms saves the memory allocated to the suspended tasks in the swap area.

  Pastorelli's approach save the states of the JVM and can be applied seamlessly to arbitrary tasks regardless of types. However a suspended process can only be resumed on the same machine it was suspended on. If the same task gets scheduled on a different machine, it has to be restarted from scratch, losing work done so far: in that case, the `suspend` is effectively analogous to a delayed `kill`.

Discussion

Although very interesting, the above mentioned efforts all suffer from some drawbacks. The Preemptive Deadline Constraint Scheduler approach employs the naive Kill primitive from Hadoop, which incurs a large amount of wasted work. The Fair Completion Scheduler only concerns about preempting Reduce tasks but ignores the case of Map tasks. Pastorelli's OS-assisted preemptive primitives allow a seamless preemption mechanism for all types of tasks, but does not support migration, i.e. restarting the preempted tasks on a different node. These drawbacks limit the usefullness of these approaches. In the next chapter, we will discuss more about the choices between preemption styles, and our approach to overcome these drawbacks.

III Algorithmic Solution

6 Pause and Resume A waster free preemption mechanism

6.1 Wait or Kill: Hadoop's dilemma

Preemption is a highly desirable feature in many cases. In schedulers that deal with fair sharing of resources, the scheduler must reallocate resources between jobs when the number of jobs changes. Since resources are normally fully utilized (in fact, this is also a novel requirement for any schedulers), newly arrived jobs often have to suffer the under-shared condition. Preemption can help release resources from over-shared jobs to give to under-shared ones, thus balancing the fairness in the system.

In other cases, it is sometimes desirable to preempt some long running jobs to give the slots to shorter jobs. Shorter jobs contain a smaller number of tasks, and have a lower chance to get a local task, compared to a longer one. Even if the splits are of the same size, the time to transfer data from one node to another will be more significant for a short job, compared to a long one. Preempting long jobs to provide local execution for short jobs not only helps reduce the relative response time, but also reduces the average waiting time for jobs.

However, original Hadoop only comes with a simple approach for this need: the Kill mechanism. Killing is fast and dirty: it reallocates resources instantly and gives control over locality for the needed ones, but it has a serious disadvantage of wasting the work of killed tasks, especially long running tasks. Moreover, the amount of wasted work becomes more consequential if the killed task is a Reduce task: later re-launched copy of the task will have to fetch all the map output again, causing more stress on the network resource which has always been scarce.

Waiting for running tasks to finish helps avoid the above-mentioned disadvantage. However, waiting will negatively impact fairness, as a new job needs to wait for tasks to finish to achieve its share. Besides, locality is not guaranteed, as the new job may not have any input data on the nodes that free up.

Figure 6.1 demonstrates an example of the three methods when a short map task needs a slot from a longer one. Figure 6.1-a presents the arrival and local execution length of each task as a rectangle. Shorter task arrives in the middle of the execution of longer task. Figure 6.1-b presents the outcome of this execution in case a wait decision is made. The length of short task is prolonged, as the task might have to execute remotely. Figure 6.1-c illustrates the kill decision. The first part of long task is discarded, and it is re-launched after the completion of short task. This scenario tends to have the longest execution time. Figure 6.1-d illustrates the situation when a waste-free preemption mechanism is implemented. All the work that long

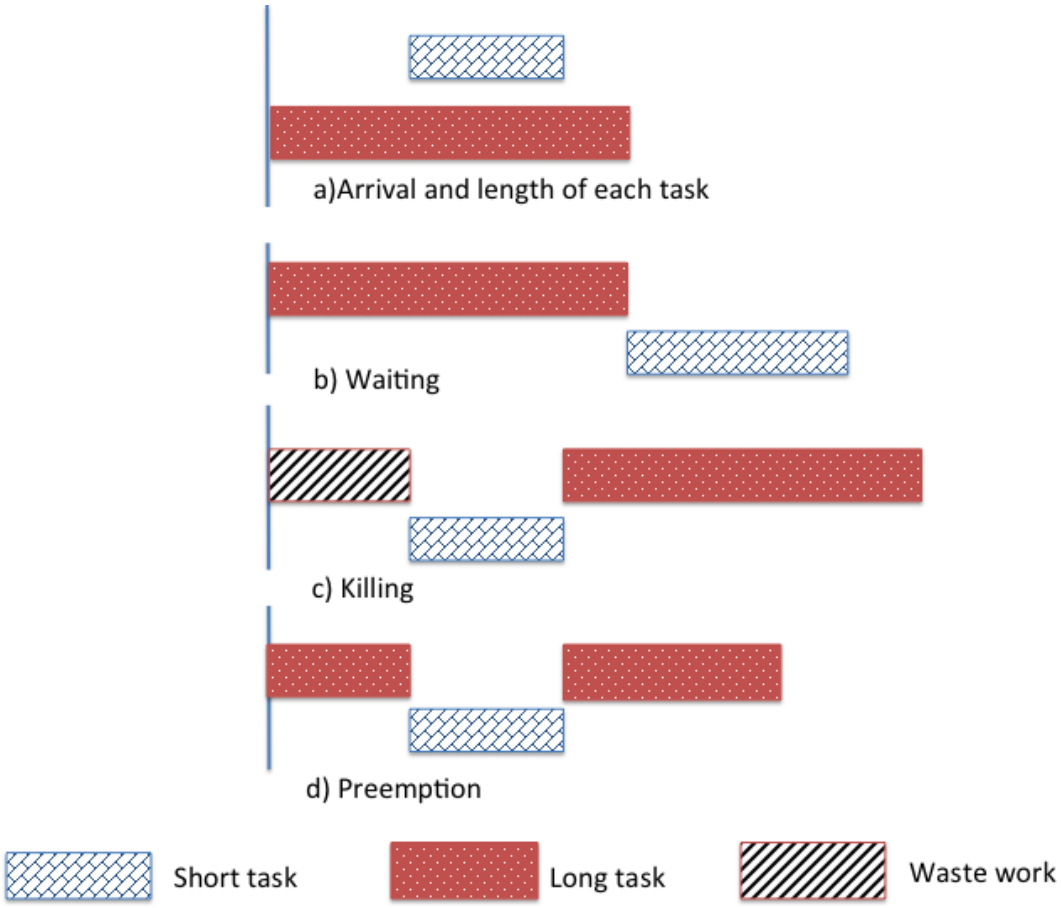

Figure 6.1: Illustration of different scenarios: waiting, killing and preemption

task has done is saved at the moment short task arrives. After completion of short task, long task picks up where it left off, and continues until it finishes. This scenario promises the best completion time, as well as fast response time, low average waiting time for tasks.

While it would always be very useful to have Map task preemption, the impact for having Reduce task preemption is even more significant. Experiments from (Tan et al. 2013) showed that even with schedulers that aims at providing fairness to jobs like Fair Scheduler, a long occupying Reduce task can defeat the effort.

6.2 Pause and Resume: a waste-free preemption mechanism

A preemption mechanism needs to be efficient and lightweight so that it can react fast enough to the dynamic system workloads. In this section, we introduce our preemption mechanism that can preempt tasks at almost any time during its execution, with low overhead and negligible delay. Due to the differences between the natures of Map tasks and Reduce tasks, our

waste-free preemption distinguishes Map and Reduce tasks. We highlight that our mechanism is fully portable, transparent as it leaves the current API of Hadoop and HDFS intact: all the existing Hadoop applications can run without any modifications.

6.2.1 Map task preemption

When a job is submitted to the Job Tracker, all of its Map tasks are initialized. Each Map task is given a "split" (or chunk) of data to process. The Job Tracker keeps track of which split belongs to which task, and has a cache list of local tasks for each node for quick retrieval. When a Task Tracker is assigned a task, it either loads the data chunk from its local storage, or fetches it from the nearest node. It then launches the actual map task by looping through every input $< key, value >$ pair and applies the map function on those pair. Once all the input $< key, value >$ pairs have been processed, the Map task reports back to the Job Tracker as "Completed", and map output is entrusted to the Task Tracker to serve to Reduce tasks.

Although the name is "pause and resume", our Map preemption mechanism is implemented in a structurally different manner. There is no "pause" in Map task preemption, but we actually split the map tasks into 2 sub-tasks at the boundary between input $< key, value >$ pairs. The first sub-task covers all the map input $< key, value >$ pair that the task has processed so far. This sub-task is considered Completed and will report back to the Job Tracker as a normally completed task. The second sub-task includes all the map input $< key, value >$ pair that has not yet been processed. This sub-task will be treated as a new independent task with almost no difference compared to any other non-running tasks. The independence between $< key, value >$ pairs of the Map Reduce programming paradigm guarantees that our mechanism works correctly and does not produce any extra (or lose any) $< key, value >$ pair, thus ensuring correctness.

The mechanism requires the machines to know which $< key, value >$ pairs belong to the task's covering range. In our implementation, we augment each task with the information about its range by providing the starting key and the ending key of the range. A normal task would have the values of both starting key and ending key equal to 0, signaling that its covering range spans the whole split. A sub-task will have either starting key, or ending key, or both the two keys to be different from 0. Our mechanism does not dictate the maximum number of sub-tasks an original task can split into: a task can split to any number of sub-tasks, and a sub-task, upon creation, is treated as a normal original task, and can be further split into more sub-sub-tasks.

A Map task now does not simply loop through every single $< key, value >$ pair like in the original source code. At every $< key, value >$ pair, it needs to make sure the pair falls inside the task's covering range. A simple comparison between starting key, ending key and the current key at each round serves the purpose. Besides, the Map task now needs to actively listen to the signal from the Task Tracker in order to stop at any time. Finally, before committing the map output, the Map task needs to mark the last processing input $< key, value >$ pair, so the Job

Algorithm 5 Pseudo for new Map task logic

```
/*get the information of the range*/
startKey = task.getStartKey();
endKey = task.getEndKey();
boolean canGoOn = true;
while (task.hasNextKey() AND canGoOn) do
  currentKey = task.getCurrentKey();
  if ( currentKey < startKey) then
    /*Skip*/
  else if (endKey ≠ 0 AND currentKey > endKey ) then
    /*has finished this task's range*/
    canGoOn = false;
  else
    apply map function(currentKey);
  end if
  /*get stop signal from Task Tracker*/
  canGoOn = !getStopSignal();
end while
if ( (endKey == 0 AND task.hasNextKey()) OR currentKey < endKey ) then
  markCurrentKey(currentKey);
end if
```

Tracker can know whether this task has completely finished or needs to be split into sub-tasks. Algorithm 5 illustrates the new Map task's workflow.

The creation of sub-tasks does not modify the underlying data split's structure. Sub-tasks still require the same data split to process compared to its parental original task. This poses a drawback in our mechanism regarding the efficiency of data usage: the range of a sub-task might only cover, for example, 10% of the data, but the same amount of data (1 chunk = 64 MB by default) will be required to transfer. It is then advisable not to split the task into too many small sub-tasks, as the network and computational waste would be high.

A few more details need to be taken care in order to facilitate the correct execution of our Map preemption mechanism. First, Reduce tasks need to know about the existence of the new sub-tasks. In Hadoop, the Job Tracker notifies Reduce tasks about the location of map outputs through a Map Completion Event message. Reduce tasks assume that if they receives enough Map Completion Event messages (equals to the number of Map tasks that was initialized at the beginning), they will know about all the existing Map outputs' locations. Since our number of Map tasks changes, we need to include this information inside the Map Completion Event message to notify the Reduce tasks about the newly born sub Map tasks.

Second, the preemption command cannot be delivered in a timely manner to the Map tasks. Communication between child Map task process (the JVM instance that actually carries out the work of a Map task) and the Task Tracker process is rather limited: a child task only updates its progress every (by default, the parameter PROGRESS_INTERVAL) 3 seconds. There is no direct way for the Task Tracker to command the child task, except forcefully killing it. We modify

the progress update channel to transmit the command: the communication will return a value signaling whether it is safe for the task to continue processing. Since this value is only transmitted at each progress update, there is at least a PROGRESS_INTERVAL (by default, 3 seconds) delay between the moment the Task Tracker receives the preemption request, and when the Map task is actually preempted. In our implementation, we adjust the PROGRESS_INTERVAL parameter to be equal to the heartbeat interval for better reaction, even though the value itself does not affect the correctness of the mechanism.

6.2.2 Reduce task preemption

A Reduce task is divided into 3 different phases. The Shuffle phase fetches all the segments that belong to it from intermediate Map Output data. The Sort phase performs a sort operation on all the fetched data, which was kept in $< key, value >$ pair format. Finally, the Reduce phase applies the user-defined reduce function on each of the Reduce-input $< key, value >$ pairs. Since the 3 phases are heavily dependent on each other, a similar "splitting" approach like in Map task preemption is unviable. We instead choose the traditional "pause and resume" approach where, at the moment of pausing, all the data of the Reduce task is stored in the local storage of the Task Tracker and, at the moment of resuming, data is loaded back to the memory.

Our mechanism allows a Reduce task to preempt itself at any time during the course of the Shuffle phase, and at the boundary of other phases. The Sort phase is usually very short due to the fact that Hadoop launches a separate thread to merge the data as soon as they become available. Preempting Reduce task during Reduce phase is also feasible, however we will consider this in future work.

6.2.2.1 Pause

During the Shuffle phase, a Reduce task fetches all the segments that belong to it from all intermediate map outputs. According to the sizes of the segments, the Reduce task stores them either to local disks or in memory. Meanwhile, multiple merging threads merge fetched data into larger segments and store them in an ordered structure that later can be popped out to feed to Reduce functions. Preserving the state of the Shuffle phase means to keep track of the shuffling status of all segments. Upon receiving preemption request, the Reduce task stops all the fetching and merging thread gracefully: it allows the threads to finish the last unit of work they are currently on. Fetching threads can finish fetching the last segment of Map output, while Merging threads can finish merging the current segment.

After stopping all communication and sorting threads, the Reduce task flushes all the in-memory data to the disks while leaving all the on-disk segments untouched. In-memory data includes in-memory segments and other data to keep track of the progress (number of copied segments, number of sorted segments). These data are kept in files stored in each task attempt's specific folder so that later re-launched attempt can access to these files and resume operation.

Preemption at the phases' boundary follows exactly the same procedure. Data is flushed to disks, and all the information needed to re-create the running task is also stored in files. After that, the Reduce task preempts itself and releases the slot. The task reports back to the Job Tracker with a new status of "SUSPENDED". Suspended tasks will go through almost the same procedure as Failed tasks, except increasing the count for failed tasks. They are also added back to the pool for later resumption.

6.2.2.2 Resume

The information about the previously launched Task Tracker is stored inside the task. Upon re-launch, the Reduce task checks to see if it has been launched somewhere before. The location can be either local (task is re-launched on the same Task Tracker) or remote (task is re-launched on a different Task Tracker). The task then tries to fetch the already processed data from the previously launched Task Tracker before resuming to the point where it left off.

Our mechanism is incremental in the sense that it allows the task to be preempted multiple times. However, we do not allow the re-launched task to be preempted during the period when it fetches data from a previously launched Task Tracker. The Task needs to finish resuming to the previous state before being able to be preempted again. This is to prevent having flushed data on too many Task Trackers, which will make the resuming process complex and inefficient.

6.3 Preemptive locality-driven scheduler

In order to demonstrate the effectiveness of out new mechanism, we designed a new scheduler, called the Preemptive locality-driven scheduler (PLS), that employs this new option to provide better locality for Map Reduce jobs. In normal situations without any failure, PLS behaves similarly to the default FIFO scheduler of Hadoop. However, when failures occur, PLS actively leverages the preemption mechanism to provide local execution for failed tasks.

6.3.1 Design of the Preemptive Locality-driven scheduler

FIFO is the default scheduler that comes with the current stable release of Hadoop (version 1.2.1). As mentioned earlier, FIFO simply executes jobs in the order they were submitted. When a failed task is discovered, it is sent to the failed queue and will be executed as soon as any slot becomes available, regardless of locality. This might lead to performance degradation if many failed tasks are assigned non-locally. The Preemptive Locality-driven scheduler addresses this problem by trying to provide local execution for failed tasks using its preemptive primitive.

In Hadoop, the Job Tracker maintains the latest status of each task tracker (as seen from the latest heartbeat). Leveraging this information allows the scheduler to make smarter decision about the allocation of resources. Upon a task's failure, PLS assembles a list of Task Trackers that has data local to the failed task (if it was of Map type), or simply list of all running Task

Trackers (if it was of Reduce type). If any of these task trackers has a free slot of the right task type, PLS simply allows the task to be launched from that task tracker at the next heartbeat. If no free slot is found, PLS tries to find appropriate task from all the currently running tasks on those task trackers to preempt. Figure 6 illustrates the logic of finding a task to preempt:

Algorithm 6 Finding a task to preempt

```
/*get the list of all suitable Task Trackers*/
List taskTrackers = getListOfTaskTracker();
/*get the list of all tasks running on those task trackers*/
List tasks = getListStatusesOfTasks(taskTrackers);
/*sort the tasks in the reverse job order*/
tasks.sort();
for {Task t : tasks} do
  if ( t belongs to jobs of lesser priority)
  AND t.progressScore < THRESHOLD
  AND !t.isCleanUp() AND !t.isSetUp()
  AND !preemptList.contains(t) then
    preempt_task(t);
    preemptList.add(t);
    return;
  end if
end for
```

PLS only allows failed tasks to preempt tasks from later submitted jobs. To respect the FIFO order, PLS prioritizes preempting tasks that belong to the latest submitted jobs. However, not all tasks would be considered for preemption. Preempting a task at, for example, 0.99 progress score would not be advisable, as it only introduces unnecessary waste which is more costly than waiting for that task to end (and perhaps the task will be already completed by the time of the next heartbeat). We impose a THRESHOLD of progress score so that almost complete tasks will not be considered for preemption. This THRESHOLD value is configurable via mapred-site.xml file, and the default value is set to 0.8.

PLS also needs to maintain a list of "already chosen for preemption" tasks for duplication check. The preemption request is only sent when the next heartbeat from the Task Tracker hosting the task arrives. In the mean time, other failed tasks might trigger a preemption decision routine, and might end up selecting the same promising task. The list of "already chosen for preemption" makes sure no two failed tasks decide to preempt the same running task, thus ensuring the correctness of preemption decision.

In case PLS cannot find any task to preempt (for example, tasks all belong to the same or higher priority jobs, or all tasks are close to completion), PLS lets the task to be launched at any arbitrary Task Tracker that has free slot the earliest. This is similar to the original FIFO scheduler's procedure. This is to avoid having to wait for too long for a local slot.

6.4 Discussion about usability

The preemptive locality-driven scheduler is just one example of how the Pause and Resume mechanism can be utilized to improve Hadoop performance. Consider the Fair scheduler, where Hadoop sometimes needs to kill tasks to rescue slots for under-shared pools. Providing that jobs arrive at pools at different rate, a serious problem of "mass killing" can arise when Fair scheduler tries to meet the ever-changing share ratio.

Pause and Resume does not help to avoid this problem, but it can limit the effect of Killing by preserving the previous effort of tasks. Capacity scheduler does not support preemption once a task is launched, but the idea was proposed and put into consideration for future extension. Providing a preemptive version of Fair and Capacity scheduler is on the roadmap for our future work.

Pause and Resume can also be used in other common task scheduling algorithms that have not yet been seen in Hadoop. The Shortest Remaining Time First scheduler (Silberschatz et al. 1998) is a well-known OS-task scheduler that aims at optimizing the average response time of tasks. Periodically, the OS scheduler checks the estimated remaining time of tasks, and preempt the currently running task for shorter one if needs. The idea can also be applied to the Hadoop scheduler, where shorter jobs will receive resources before longer ones. However, preempting Hadoop tasks (and jobs) can be extremely expensive in term of wasted CPU cycles and time, if smaller jobs keeps arriving. The waste-free preemption mechanism introduced in this thesis can be leveraged to provide better utilization of resources.

Waste-free preemption is a powerful tool for resource control. We expect to investigate more in the usability of this feature to not only improve performance of Hadoop in normal situations, but also in other cases, such as task failure or node failure.

IV Evaluation and Conclusion

7 Evaluation

In order to prove the efficiency of Pause and Resume preemption mechanism, we present the result from selected experiments. The goal is to demonstrate how the Preemptive Locality-driven scheduler manages to help jobs under failures. For this purpose, we concern about the execution time of jobs that suffer the failure, as well as their localities.

7.1 Experimental setup

Cluster setup

We continue to use Grid5000 for our experiment. The size of the cluster and the specification of each node remains the same (1 Master + 20 Slave nodes).

Hadoop setup

The comparing Hadoop version is still the 1.2.1 current stable release. Each node is now equipped with only 2 Map and 2 Reduce slots, as in the default value. The number of Reduce tasks for each job is set at 20 in order to trigger preemption. HDFS parameters remain at replication factor of 2 and chunk size of 128MB. For timely reason, the expiry time is set at 30 seconds. Speculation is turned off for correct evaluation of Pause and Resume's efficiency.

Jobs

For simplicity, we only run 2 jobs of WordCount application with the same input size of 30GB. Failure is injected at 150 seconds to maximize the number of failed tasks from job 1.

7.2 Overview results

Execution time

Figure 7.1 shows the finishing time of each job in the 4 different schedulers: the three default schedulers of Hadoop (FIFO, Fair and Capacity scheduler) and Preemptive Locality-driven scheduler (PLS). There is little difference in the total execution time of the two jobs (represented

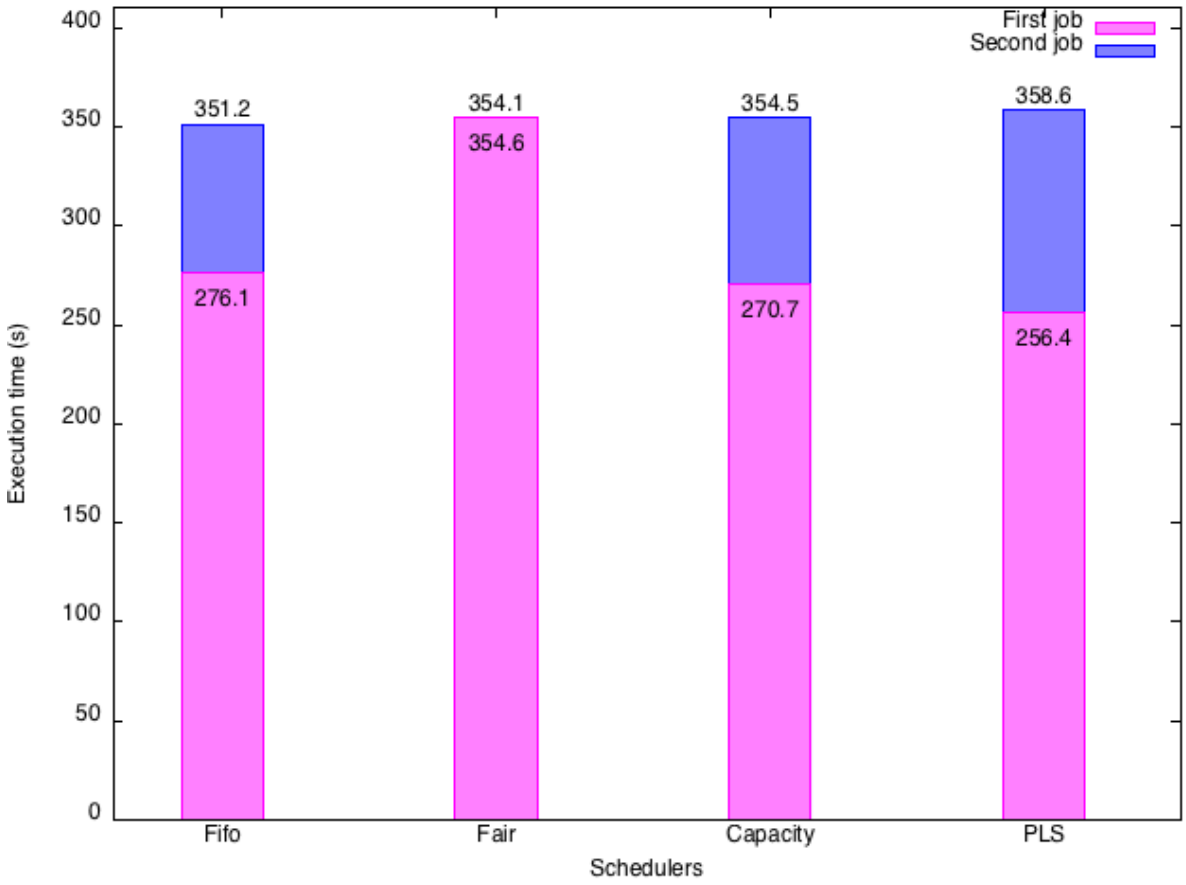

Figure 7.1: Finish time of jobs with different schedulers

by the finishing time of the seconds job), but the trend is that PLS has to pay the most time to finish the set of jobs. However, PLS shows good improvement in the finishing time of job 1 - the job that suffers from the failure. PLS only requires 256.4 seconds to finish job 1, roughly 14 seconds (5.2%) faster than in Capacity and about 20 seconds (7.2%) when compare to FIFO scheduler. Fair scheduler performs poorly on this metric, as jobs share the resources of the cluster. In fact, job 1 takes more time to finish than job 2 does in Fair scheduling. This can be explained by the fact that job 1 starts first and has more failed tasks than job 2.

To further understand about the efficiency of PLS, as well as the actual behavior of Pause and Resume preemption feature, we perform the same experiment with 3 different flavors of PLS. We omit completely the preemption function of PLS and mimic the FIFO scheduler in a flavor called FIFO*. FIFO* still has to pay the overhead of allowing Pause and Resume at any time (this overhead will be discussed later). The Kill-PLS uses the Kill primitive provided by Hadoop instead of Pause and Resume. Finally, the PLS that leverages the Pause and Resume function is also included.

Figure 7.2 compares the execution of Hadoop with different flavors of PLS. The FIFO*

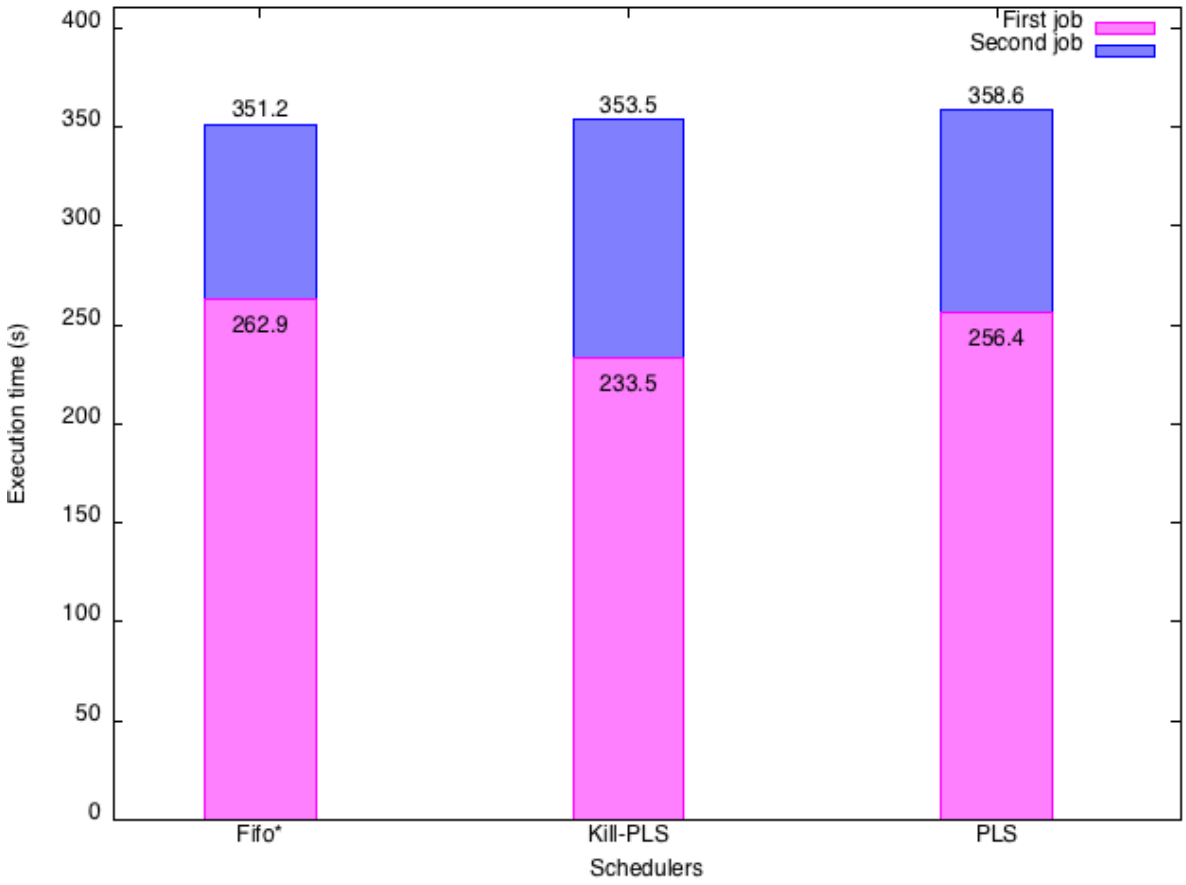

Figure 7.2: Execution with different flavors of PLS

enjoys the best performance due to the fact that there is no killing or preemption overhead. Tasks are launched and finish naturally without any waste work (from killing) or delay (from splitting and re-launching a task).

Though witnessing some degradation in the total performance, Kill-PLS and PLS gain some improvement in term of Job 1's execution time. Both of Kill and Preemption finish the first job faster than that of FIFO*, in the order of roughly 19 (11.2%) and 7 (2.5%) seconds, respectively. Kill-PLS outperforms the other 2 competitors thanks to the fact that kill command can instantly rescue the slots from currently running tasks, while preemption command needs to pay some delay so that running tasks can save the states and release the slots. However, PLS manages to save some work from preempted tasks. PLS only requires an extra 102 seconds to finish the second job, compared to 120 seconds in Kill-PLS (an improvement of 15%).

Data locality of the first job

Figure 7.3 shows the locality of the first jobs in the three different flavors. FIFO* has the lowest value of data locality (90.4%) due to the fact that failed tasks are assigned regardless of locality.

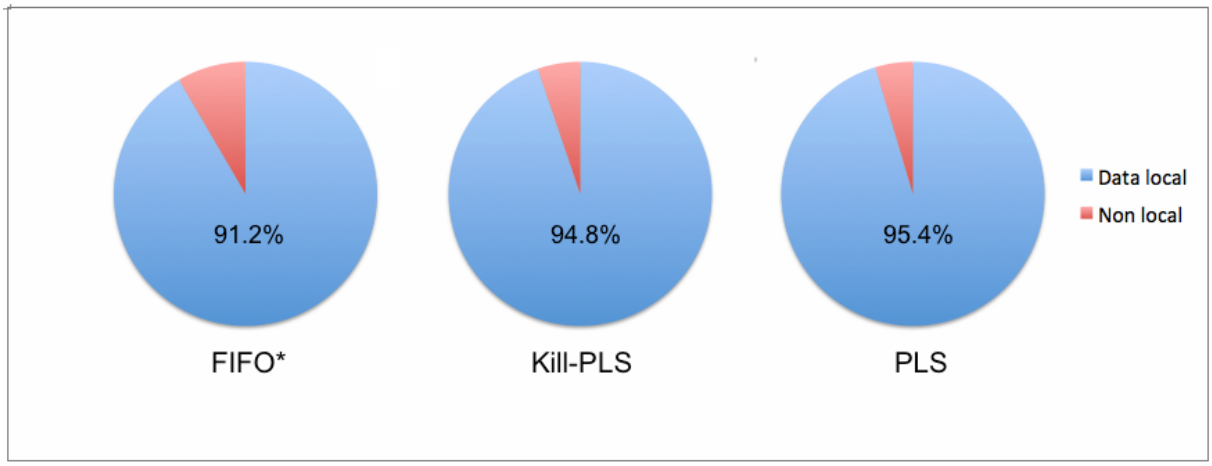

Figure 7.3: Locality of the first job

Kill-PLS and PLS observe higher locality in Map tasks and equal each other with the values of 94.8% and 95.4%, respectively. This means PLS manages to improve the locality of Map tasks by almost 5%. Out of 12 failed tasks (tasks that were initially launched on the failed node), FIFO* assigns all 12 tasks remotely (rack-locally), while Kill-PLS and PLS assigns only 3 remotely (as in table 7.1).

FIFO*	Kill-PLS	PLS
0/12 (0%)	9/12 (75%)	9/12 (75%)

Table 7.1: Number of local re-executed tasks

Overhead of preemption

To facilitate the preemption feature, Hadoop needs to pay some extra overhead. Map tasks need to compare the current key with the allowed range of that task, at each input key. Reduce tasks, though little, also need to check for signal from the Task Tracker if it needs to stop once in a while. However, experiments show that this overhead is rather small: Figure 7.4 shows that the 3 different implementations (the original FIFO, the FIFO*, and the PLS) have similar execution time.

In fact, the Map tasks in the preemption-supported version indeed requires longer time to finish compared to that of original version. To measure this overhead, we take the average execution time of only Map tasks from the first wave of execution (40 tasks). During the first wave, there were no Reduce tasks, and all the Map tasks were local. The average value will reflect the overhead accurately. Table 7.2 proves that the execution overhead is at the level of 1 second for each tasks.

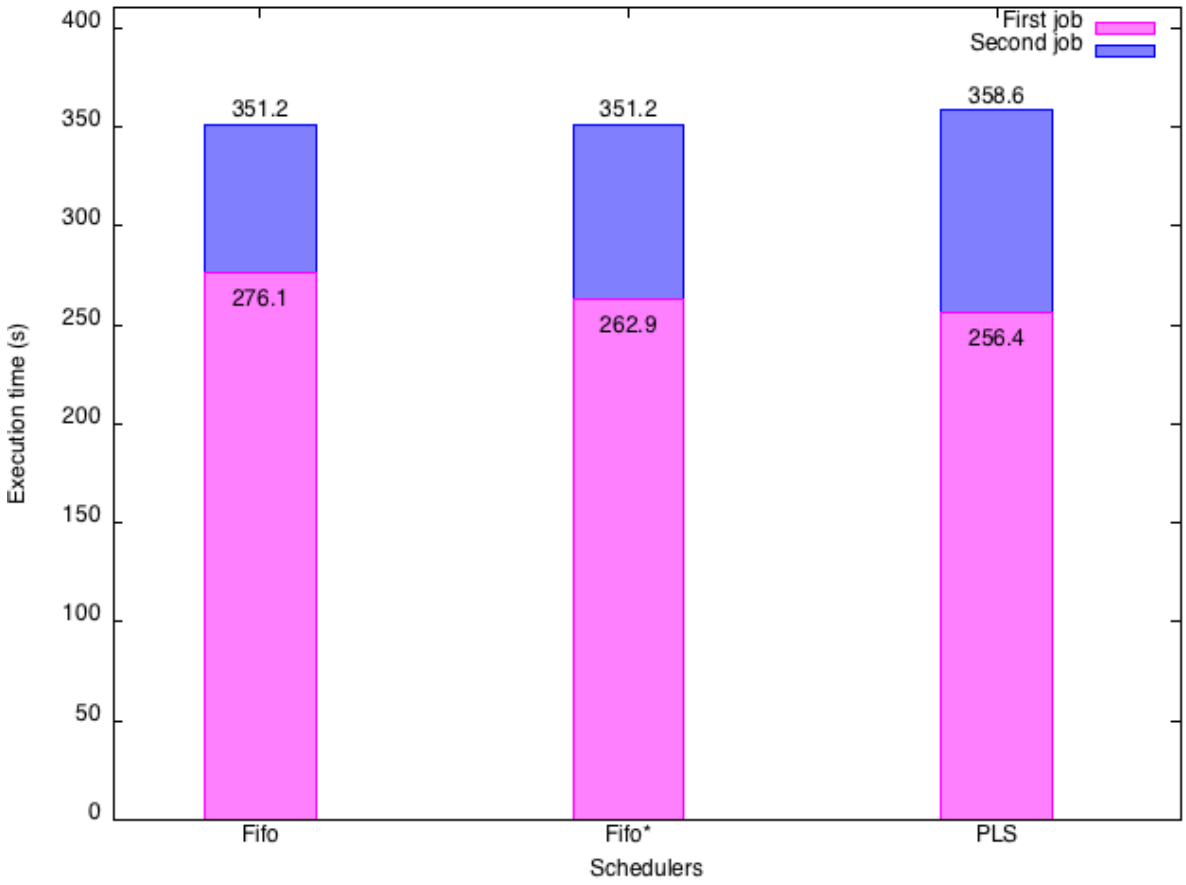

Figure 7.4: Overhead of preemption in normal cases

	Original	Mod Fifo
Time	23.65	24.61 (+4%)

Table 7.2: Overhead on Map tasks

7.3 Zoom in the tasks execution

We present the execution graph of all tasks throughout the course of jobs to illustrate the differences in scheduling between the three flavors of PLS, as well as to verify the effectiveness of preemption.

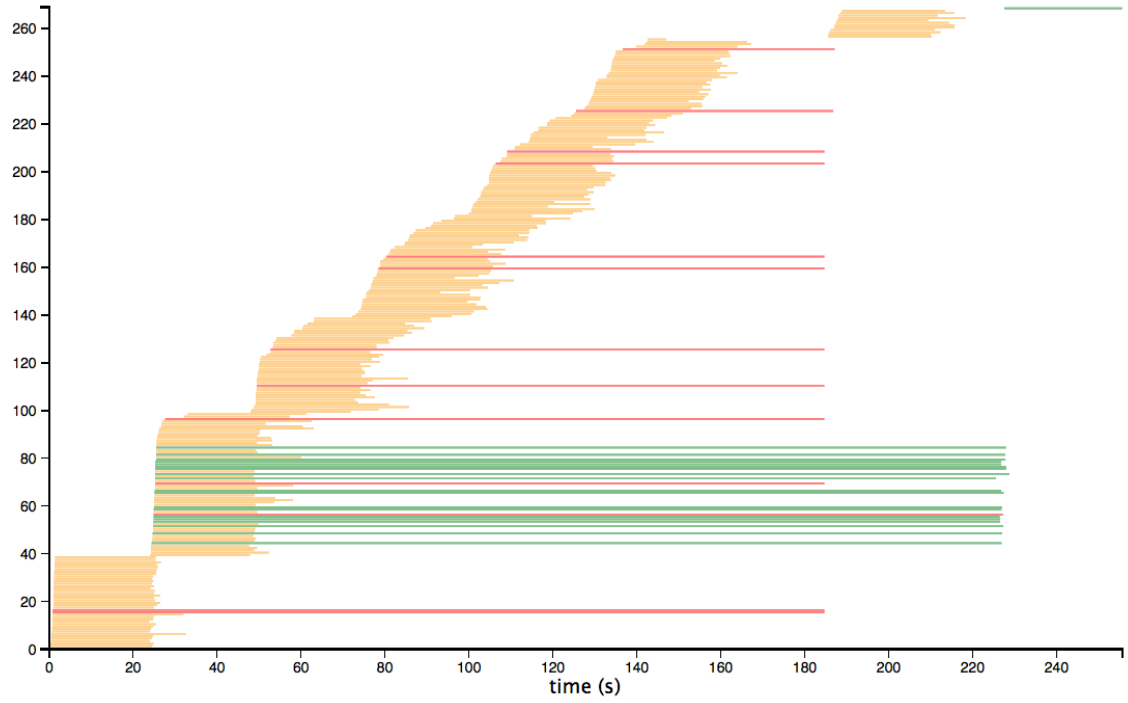

Figure 7.5: FIFO*'s execution of job 1

Job 1

Figure 7.5 shows the execution of all tasks from Job 1 in FIFO*. The yellow lines represent Map task, green lines represent Reduce tasks, and red lines represents irregularly finished tasks. The Map phase is clearly divided into waves. The first wave consists of 40 tasks (equals to the total number of Map slots in the clusters) and is rather regular: tasks start and finish at similar time. Failure is injected at 150s, and is discovered at around 185s. Since the $expiry\_time$ is set at 30s, there is a 5s difference due to the fact that Hadoop does not continuously check for failed Task Tracker. In fact, it only checks for failed Task Tracker every $expiry\_time/3 = 30/3 = 10$ seconds. Upon Task Tracker failure discovery, all the Map tasks on that Task Tracker are declared failed and re-executed. Although the discovery of failure happens at around 185s, there is one red task that lasts till the end of other Reduce tasks. This red task is actually the Reduce task that was running on the failed Task Tracker. For running failed tasks, Hadoop keeps them in a certain queue, launches a Task Clean Up task of the same type before moves the failed tasks to failed queue. Since there is no free Reduce slot at the moment of failure discovery, the failed Reduce task has to wait until one of the Reduce tasks finishes. This awaiting time can be significantly long if the Reduce tasks last long.

Figure 7.6 illustrates the execution of job 1 in PLS. The failed Reduce task is re-launched earlier thanks to the preemption of 1 Reduce task from job 2 (as we can see later). The re-execution of failed Map tasks is also more regular: Preemption allows failed running tasks to

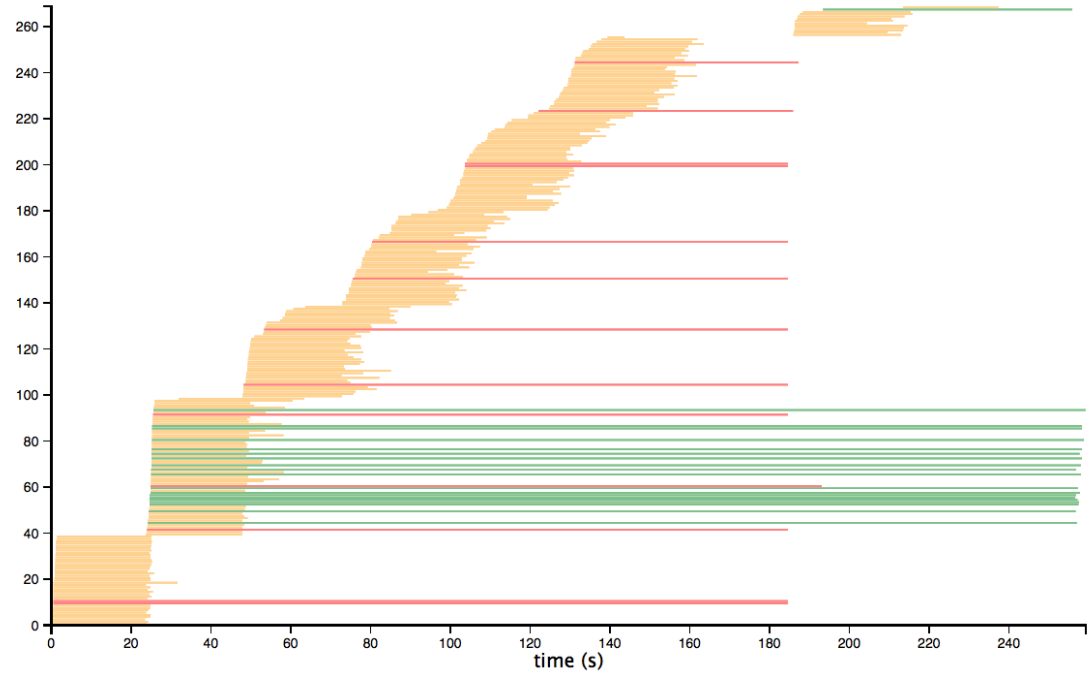

Figure 7.6: Preemption's execution of job 1

be relaunched quickly rather than having to wait for tasks to finish. The early launching of failed tasks (especially Reduce tasks) in this implementation greatly improves the execution time.

Figure 7.7 illustrates the execution of Job 1 in Kill-PLS. There is only one difference when compared with PLS: the red Reduce task finishes faster. This is because Kill primitive is faster in acquiring slot than preemption.

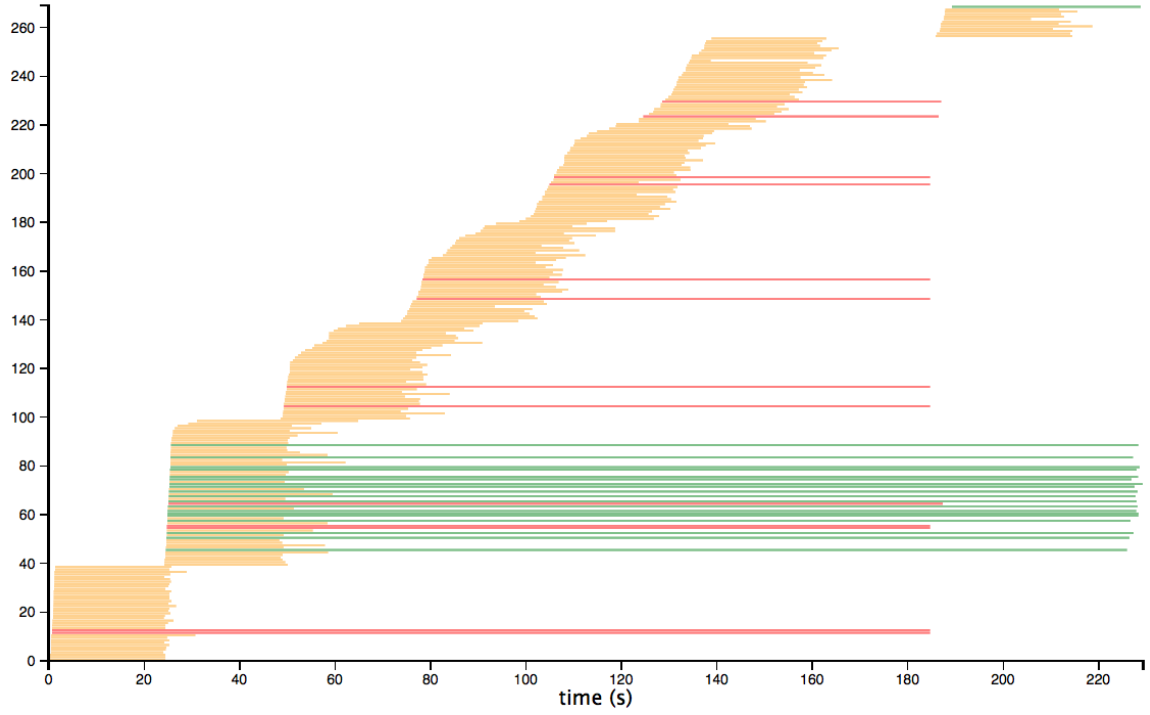

Figure 7.7: Kill-PLS's execution of job 1

Job 2

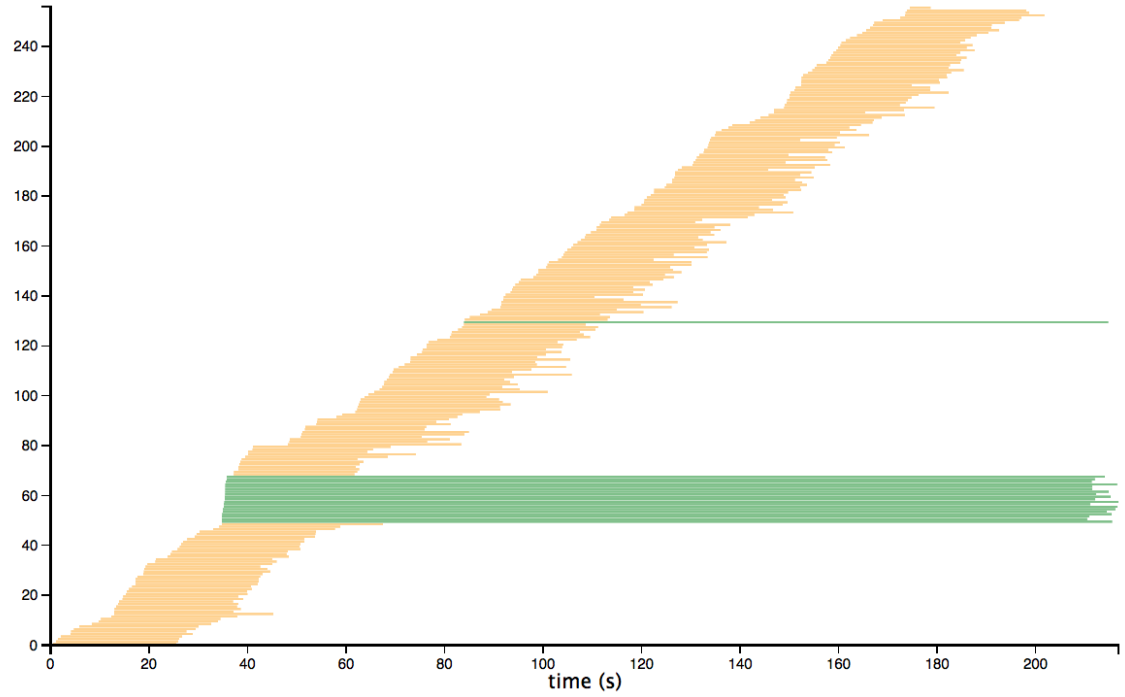

Figure 7.8: Fifo*'s execution of job 2

While the execution of job 2 in FIFO* (Figure 7.8) does not contain any red task, PLS's execution of job 2 contains 1 red task. It is the Reduce task that was chosen to be preempted. This results in another "second wave" of Reduce task that starts at around 120s. (FIFO has only one "second wave" Reduce task that starts after the completion of any of the Job 1's Reduce tasks).

It is also possible to observe some irregularly short Map tasks from Job 2 in PLS (those that end around the 50s). These are the tasks that were preempted to give slot for failed Map tasks from Job 1. Since the preemption mechanism used with Map task is splitting rather than "Pause and Resume", those tasks are considered "normally completed" and are colored yellow. Some other short tasks are observed about 20 to 30 seconds after that: they are the second half of the split Map tasks. We award priority to once-preempted tasks to be executed on node with local data over normal tasks to avoid the waste in transferring unused data.

The execution of job 2 under Kill-PLS resembles that of PLS, except for the fact that there are some red tasks. These are tasks that were killed to save slot for failed tasks from job 1.

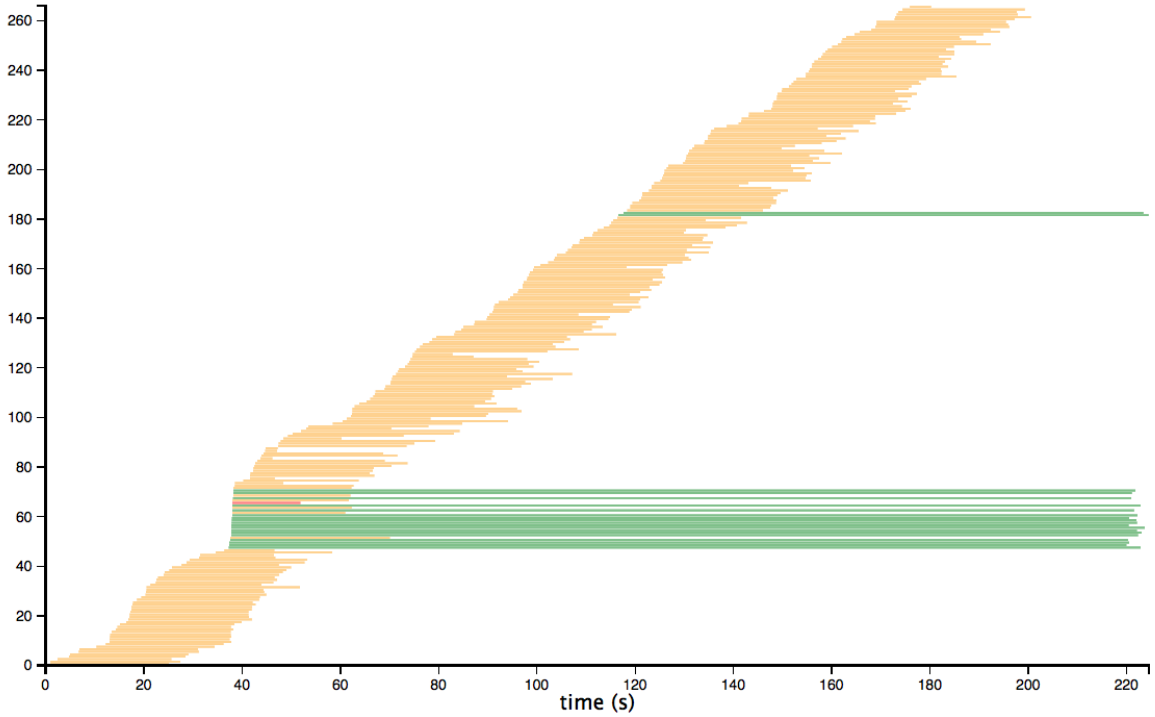

Figure 7.9: PLS's execution of job 2

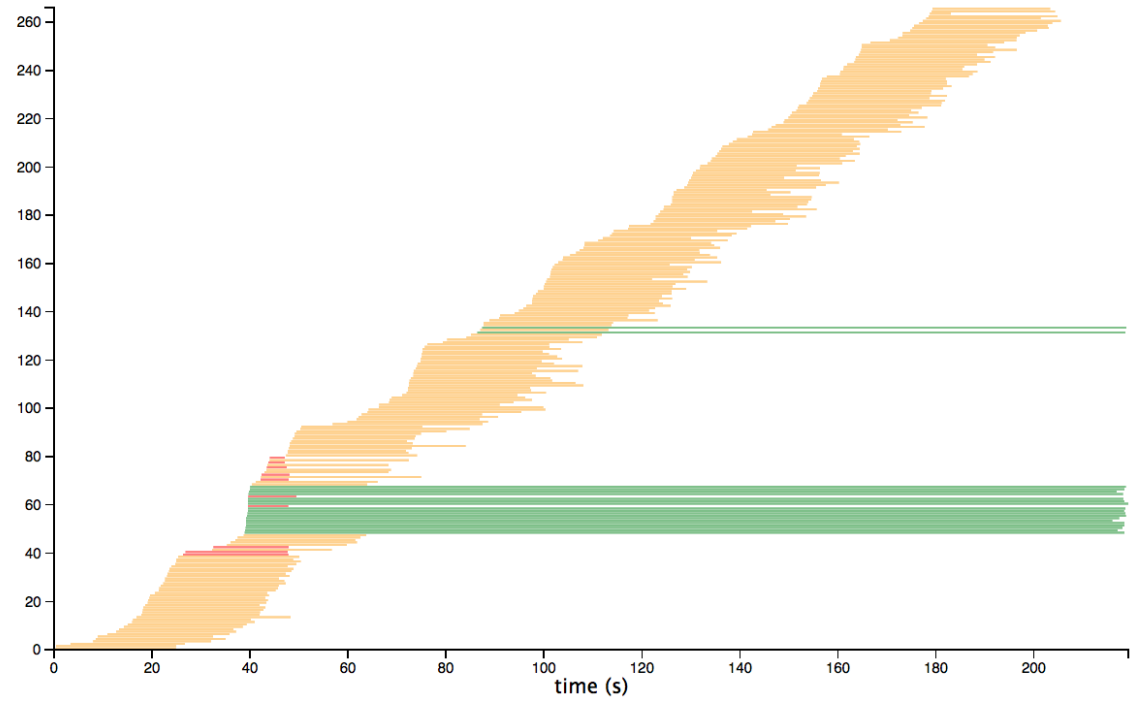

Figure 7.10: Kill-PLS's execution of job 2

8 Conclusion

The unprecedented growth in data center technologies and services in the recent years opens new opportunities for data-intensive applications in the Cloud. Many MapReduce-like frameworks have been introduced as services. MapReduce in the Cloud provides a cost-effective way for smaller businesses to take advantages of this simple yet powerful programming paradigm. However, users have to pay the cost of for failures, which have become a norm rather than an exception. Thus, Fault-tolerance for MapReduce becomes a topic that attracts much interest from both academy and industrial institutes.

This thesis addresses the problem by investigating the fault-tolerance mechanism of Hadoop, a popular implementation of MapReduce. It presents some results to illustrate Hadoop's behavior under different situations. The intent was to confirm the mechanism and analyze the drawbacks of how Hadoop handles failures.

We proposed the design of a new feature for Hadoop: the waste-free preemption function. By allowing a task to preserve its state and release the slot in a timely manner, Hadoop can have more control over resources. This in turn will help increase the performance of Hadoop under the occurrence of failure. The preemption feature was implemented not only with the intention of improving performance under failure, but also to open new possibilities for further improvement under other circumstances.

Finally, we evaluated the effectiveness of the new feature considering some basics metrics such as execution time and data locality. In this work, we compare the original Fifo scheduler with a preemptive version. Experiments show that our new feature improves the execution time of Hadoop jobs when failure occurs. Although the preemption mechanism imposes some overhead, if wisely used, it can greatly improve Hadoop's performance.

Future work

Our work suffers from some problems regarding the usability, as well as efficiency. Firstly, the Reduce Pause and Resume mechanism should be further extended to allow preemption during Reduce phase. This allows a more fine-grained control over the Reduce tasks. Secondly, it would be interesting to evaluate the cost of preempting a Reduce task, in comparison with Killing. Killing instantly releases the slot, but wastes the effort, while Preemption may take sometime before the slot is released. Also, upon resuming, Reduce tasks also require some extra time to load up the data from local storage. These two delays add up and in some cases,

it is more beneficial to decide to Kill rather than Preempt a Reduce task.

Thirdly, the scheduler can try to provide locality to Reduce tasks, in the sense that placing a Reduce task on the node that has the most data (or has the potential to have the most data) would be better. This can be achieved by calculating the number of Map Output on each node, and the possibility to have another Map Output on each node for every pending Map task.

Finally, it would be interesting to evaluate the efficiency of the preemption feature with different schedulers other than the default FIFO. The Fair scheduler can make use of this feature in providing fair share to pools: traditionally the Fair scheduler has to kill tasks to rescue slots for under-shared pools. Waste-free preemption will be helpful in this situation, especially when the share is constantly changing (due to the arrival of new jobs).

Bibliography

Ahmad, F., S. Lee, M. Thottethodi, & T. Vijaykumar (2012). Puma: Purdue Mapreduce benchmarks suite.

Amazon. Amazon elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

Amazon. Amazon web services. http://aws.amazon.com/.

Ananthanarayanan, G., S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, & E. Harris (2011). Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, pp. 287–300. ACM.

Apache. Apache Hadoop Welcome page. http://http://hadoop.apache.org.

Bicer, T., W. Jiang, & G. Agrawal (2010). Supporting fault tolerance in a data-intensive computing middleware. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12. IEEE.

Borthakur, D. (2010). Facebook has the world's largest Hadoop cluster! http://hadoopblog.blogspot.fr/2010/05/facebook-has-worlds-largest-hadoop.html.

Chandra, A., R. Prinja, S. Jain, & Z. Zhang (2008). Co-designing the failure analysis and monitoring of large-scale systems. *ACM SIGMETRICS Performance Evaluation Review 36*(2), 10–15.

Chen, Y., A. Ganapathi, R. Griffith, & R. Katz (2011). The case for evaluating mapreduce performance using workload suites. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pp. 390–399. IEEE.

Czajkowski, G. (2008). Sorting 1pb with mapreduce. *Google, Blog. Available at googleblog. blogspot. com/2008/11/sorting-1pb-with-mapreduce. html. Last accessed on January 7*, 2012.

Dailymail (2011). Web chaos: Amazon Cloud failure crashes major websites Playstation network goes AGAIN. http://www.dailymail.co.uk/sciencetech/article-1379474/.

Dean, J. (2006). Experiences with mapreduce, an abstraction for large-scale computation. In *PACT*, Volume 6, pp. 1–1.

Dean, J. (2009). Large-scale distributed systems at google: Current systems and future directions. In *Keynote speech at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*.

Dean, J. & S. Ghemawat (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*(1), 107–113.

Dinu, F. & T. Ng (2012). Understanding the effects and implications of compute node related failures in hadoop. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pp. 187–198. ACM.

Fox, A., R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, & I. Stoica (2009). Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28*, 13.

Google. Google app engine. https://cloud.google.com/.

Gottfrid, D. (2007). Self-service, prorated supercomputing fun! http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/.

Grid5000. Grid5000 Home page. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home.

Jindal, A., J.-A. Quiané-Ruiz, & J. Dittrich (2011). Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, pp. 21:1–21:14. ACM.

Ko, S. Y., I. Hoque, B. Cho, & I. Gupta (2010). Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 181–192. ACM.

Lai, E. (2013). Companies are spending a lot on Big Data. http://sites.tcs.com/big-data-study/spending-on-big-data/.

Lampitt, A. (2012). Big data visualization: A big deal for eBay. http://www.infoworld.com/d/big-data/big-data-visualization-big-deal-ebay-208589.

Liu, L., Y. Zhou, M. Liu, G. Xu, X. Chen, D. Fan, & Q. Wang (2012). Preemptive hadoop jobs scheduling under a deadline. In *Semantics, Knowledge and Grids (SKG), 2012 Eighth International Conference on*, pp. 72–79. IEEE.

Logothetis, D., C. Olston, B. Reed, K. C. Webb, & K. Yocum (2010). Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 51–62. ACM.

Microsoft. Microsoft azure. https://azure.microsoft.com/.

Pastorelli, M., M. Dell'Amico, & P. Michiardi (2014). Os-assisted task preemption for hadoop. *arXiv preprint arXiv:1402.2107*.

Pinheiro, E., W.-D. Weber, & L. A. Barroso (2007). Failure trends in a large disk drive population. In *FAST*, Volume 7, pp. 17–23.

Rao, B. T., N. Sridevi, V. K. Reddy, & L. Reddy (2012). Performance issues of heterogeneous hadoop clusters in cloud computing. *arXiv preprint arXiv:1207.0894*.

Rezaei, A. & F. Mueller (2013). Sustained resilience via live process cloning. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 1498–1507. IEEE.

Roe, C. (2011). The Growth of Unstructured Data: What to do with all those Zettabytes? http://www.dataversity.net/the-growth-of-unstructured-data-what-are-we-going-to-do-with-all-those-zettabytes/.

Schroeder, B. & G. A. Gibson (2010). A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on 7*(4), 337–350.

Schroeder, B., E. Pinheiro, & W.-D. Weber (2009). DRAM errors in the wild: a large-scale field study. In *ACM SIGMETRICS Performance Evaluation Review*, Volume 37, pp. 193–204. ACM.

Seo, S., I. Jang, K. Woo, I. Kim, J.-S. Kim, & S. Maeng (2009). HPMR: Prefetching and pre-shuffling in shared mapreduce computation environment. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1–8. IEEE.

Silberschatz, A., P. B. Galvin, G. Gagne, & A. Silberschatz (1998). *Operating system concepts*, Volume 4. Addison-Wesley Reading.

Tan, J., X. Meng, & L. Zhang (2013). Coupling task progress for mapreduce resource-aware scheduling. In *INFOCOM, 2013 Proceedings IEEE*, pp. 1618–1626. IEEE.

TCS. Size matters: Yahoo claims 2-petabyte database is world's biggest, busiest. http://www.computerworld.com/s/article/9087918/.

Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, & R. Murthy (2010). Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 996–1005. IEEE.

Wang, Y., J. Tan, W. Yu, X. Meng, & L. Zhang (2013). Preemptive reducetask scheduling for fair and fast job completion. In *Proceedings of the 10th International Conference on Autonomic Computing, ICAC*, Volume 13.

Zaharia, M., D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, & I. Stoica (2010). Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pp. 265–278. ACM.

Zhu, H. & H. Chen (2011). Adaptive failure detection via heartbeat under hadoop. In *Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific*, pp. 231–238. IEEE.