Accelerate MapReduce's Failure Recovery through Timely Detection and Work-conserving Logging

by

Huansong Fu

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 12th, 2015

Keywords: Hadoop, MapReduce, Fault tolerance

Approved by

Weikuan Yu, Chair, Associate Professor of Computer Science and Software Engineering
Sanjeev Baskiyar, Associate Professor of Computer Science and Software Engineering
Alvin Lim, Associate Professor of Computer Science and Software Engineering

Abstract

MapReduce has become an indispensable part of the increasing market for big data analytics. As the representative implementation of MapReduce, Hadoop/YARN strives to provide outstanding performance in terms of job turnaround time, scalability, fault tolerance, etc.. Specifically for fault tolerance, YARN is equipped with the speculation mechanism, which can regenerate data properly in the presence of system failures. However, we revealed that the existing speculation mechanism has some major drawbacks that hinder its efficiency in job recovery from system failures, especially for small jobs, which are the paramount counterparts of MapReduce jobs in practical use. As our experiments have shown, a single node failure can cause the job slowdown by up to 9.2 times.

To address this issue, we have conducted a comprehensive study on the fundamental causes of the breakdown of existing speculation mechanism. In order to tackle down those issues, we brought about a set of techniques, including an optimized speculation mechanism, a centralized failure monitor and analyzer, a progress-conserving mechanism for MapTasks and a refined scheduling policy under failures. To evaluate our framework, we conducted a set of experiments that evaluate the performance of both single component and the framework in overall. Our experimental results show that our new framework has dramatic performance improvement dealing with task and node failures compared to the original YARN.

Acknowledgments

First of all, I want to express my sincere gratitude to Dr. Weikuan Yu, my major advisor for his substantial support, insightful instructions and patient encouragements. I have learned so much from him, not only the rich experience in research, but also the motivated and driven attitude towards work. His guidance has helped me in so many ways and it will always be a valuable treasure in my future career.

I also appreciate the earnest help I have received on this thesis from my advisory committee members, Dr. Sanjeev Baskiyar and Dr. Alvin Lim.

I would also like to thank my colleagues in the Parallel Architecture and System Laboratory: Dr. Hui Chen, Dr. Jianhui Yue, Yandong Wang, Bin Wang, Cong Xu, Zhuo Liu, Teng Wang, Fang Zhou, Xinning Wang, Kevin Vasko, Michael Pritchard, Hai Pham, Lizhen Shi, Yue Zhu and Hao Zou, for the numerous help I have received from them and the two wonderful years of journey we have experienced together. Especially I want to thank Yandong for his advice on the fault tolerance project, Cong for his help to settle me down at Auburn, and Bin and Zhuo for their kind guidance throughout my Auburn career. The time spent with you all is surely unforgettable.

I am indebted to my parents, my father Yong Fu and mother Huiming Shi. I cannot thank you enough for your endless love, encouragement and support. I will continue to make you proud.

Last but not least, I would like to acknowledge the sponsors of this research at Auburn University. Particularly, I would like to acknowledge the Alabama Innovation Award, and the National Science Foundation awards 1059376, 1320016, 1340947 and 1432892.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, the world has entered into its "Big Data era." According to a report from International Data Corporation (IDC) [14], the amount of data in the digital universe was 1,227 exabytes in 2010, and is doubling every two years. By the year of 2020, the number will skyrocket to 40,000 exabytes. This means that there are more than 5,200 gigabytes for every human being on earth. Big data not only has the volume, but it has huge commercial potentials, too. According to a recent survey of McKinsey report [17], the value of big data analytics is approximately $300 billions per year for the health care service alone in the US.

With such phenomenal balloning of data capacity, the use of big data analytics tools has been increasingly substantial. Among them, MapReduce-type cluster computing has gained wide popularity since Google introduced it [11] in 2004. Specifically, Hadoop [2] has become the *de facto* standard implementation of MapReduce. Currently, it has been evolved into its second generation, which is often called YARN [1]. YARN is designed to overcome scalability and flexibility issues in the first generation Hadoop. In YARN, resource management and task scheduling are carried out by a global ResouceManager(RM) and per-application ApplicationMasters(AM) respectively. This framework has a number of major modifications [21] and brings significant advantages such as performance improvements, scale-out abilities and programming model's diversity compared to its predecessor.

The popularity of Hadoop is largely due to its fast turnaround time [11]. In order to achieve that in the highly unstable heterogeneous environment, a mechanism called speculation is designed to contribute to the purpose. A global speculator proactively makes a copy of the stragglers that may block the job progress, and whoever finishes first will let the job proceed. Even in the presense of a whole computing node going down, as long as all

the tasks on the node are properly speculated, the job performance will not downgrade too much.

However, we have found that this approach has some major deficiencies, especially for small jobs. Fig 1.1 shows the job slowdown caused by a single node failure. When the job size is large, the slowdown is negeligible, but when the job size is small (1 GB), the slowdown can be as much as 9.2x to the normal job running time. Fig 1.2 shows a closer look at jobs with input size at 1 GB to 10 GB. We depict the slowdown along with the number of tasks in the jobs. In this way we can see that to the jobs which have 1 to 10 GB input data or 10 to 100 tasks, a single node failure can degrade the job performance by an average of 4.8x.

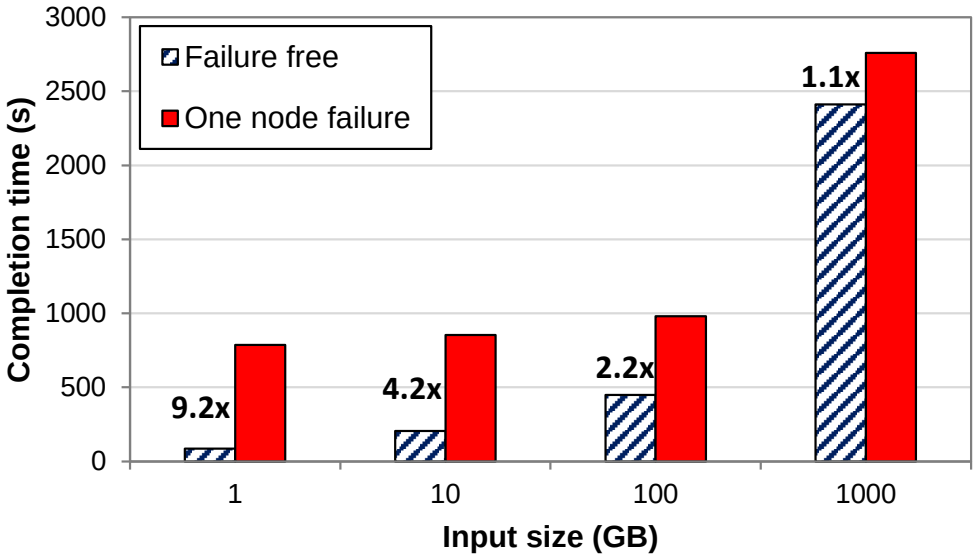


Figure 1.1: Wordcount job performance when one node fails.

How serious can this impact be? Although Hadoop is known for its ability for processing big data, the paramount of jobs in industrial use are actually small size jobs. Fig. 1.3 shows the distribution of Facebook workloads [4]. It clearly demonstrates a heavy tail tendency, in which both the number of tasks and input size follow a power-law distribution. Table 1.1 gives the exact ratio of jobs in terms of different number of tasks. For Facebook workloads, about 90% of jobs have 100 or less tasks, and a majority of portion having 100 Gb or less input data. Thus, it becomes clear that the majority of the current jobs will suffer from

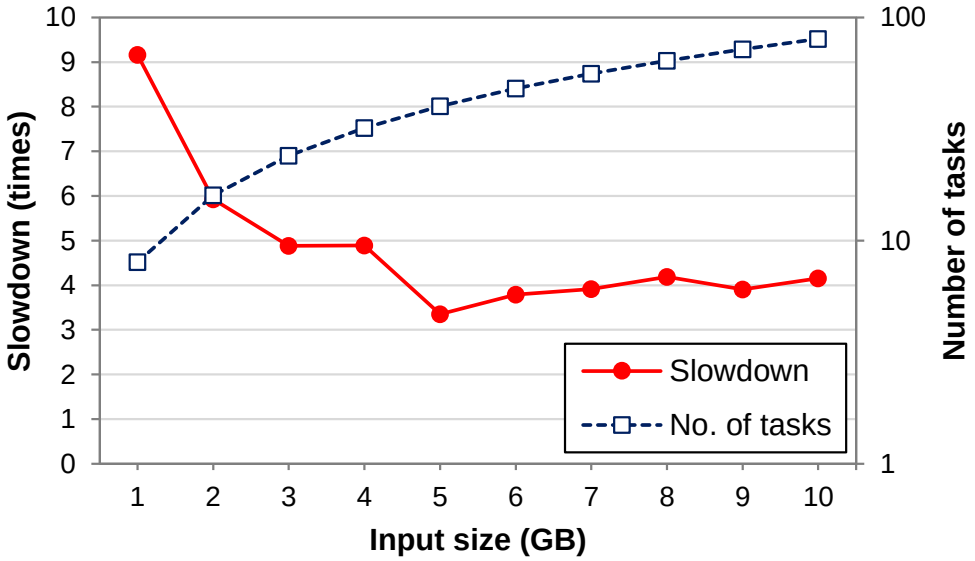


Figure 1.2: Wordcount job performance when one node fails.

the performance degradation as shown in Fig. 1.1 and Fig. 1.2. Note that we simulate only node failures in our experiment, but no other senarios that may cause an unresponsive node (network delay, disk failures, etc.). Even that, just the case of node failure itself is extremely common. According to [10], there is an average of five node failures during one MapReduce job execution. All the evidences indicate an imperative need to revisit the speculation mechanism in current MapReduce framework.

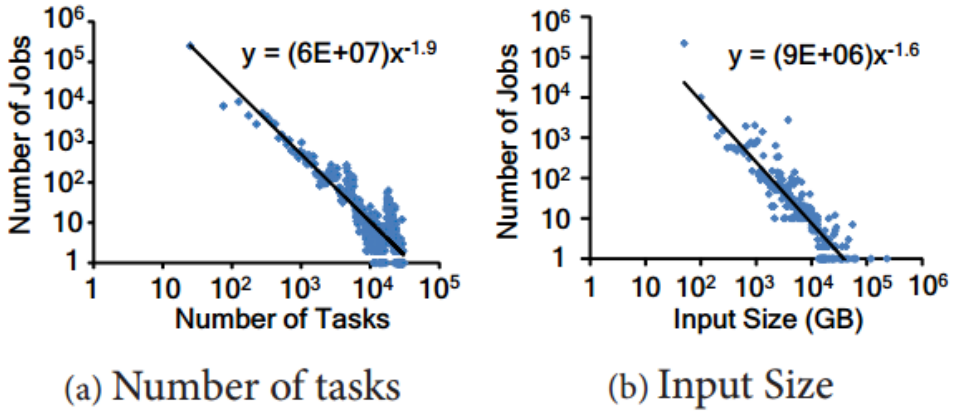


Figure 1.3: Facebook workloads distribution [4].

In order to address the aforementioned problems, we have conducted a comprehensive study on the inability of existing speculation mechanism to deal with task/node failures. We

Table 1.1: Facebook and Bing workloads distribution in number of tasks [4].

Group	**Tasks**	**% of Jobs (Facebook)**	**% of Jobs (Bing)**
1	$1-10$	85%	43%
2	$11-50$	4%	8%
3	$51-150$	8%	24%
4	$151-500$	2%	23%
5	>500	1%	2%

found that existing speculation fails to predict failed tasks because of multiple reasons. Then we brought about a set of techniques, including an optimized speculation mechanism, a refined scheduling algorithm, etc.. Our experimental results show that our new framework has dramatic performance improvenment dealing with failures compared to the original YARN.

In general, our work makes the following contributions:

- We systematically reveal the drawbacks of current speculation mechanism and analyze the impact, dissect and causes of task/node failures in great detail.
- We improve YARN's speculation algorithm by involving it with a Centralized Failure Analyzer and a heuristic failure decision algorithm.
- We design work-preserving techniques for the MapTask which only needs a minimum amount of logging stats, but provides maximum performance improvement for failure recovery.

This thesis is organized as following. Chapter 2 describes our study on current speculation mechanism and demonstrates its drawback with our experiment results. It also includes a survey of related works. Chapter 3 talks about our new service CFA and the associated algorithm FAS. Chapter 4 presents the evaluation results of our framework. Finally we conclude the thesis in chapter 5.

Chapter 2

Background and Motivation

2.1 An overview of MapReduce and Hadoop/YARN

MapReduce [11] is a programming model for writing distributed computing programs with ease. It has a simple abstraction for parallel processing of massive amount of data, which is to extract the input data in small key/value pairs and apply user-defined map/reduce functions on each pair.

YARN [1] is designed not only as the updated version of Hadoop MapReduce, but also as a resource management infrastructure. It is able to support various programming models, including MapReduce and MPI. To achieve that, it is comprised of two categories of components, one ResourceManager and many NodeManagers. The NodeManager associates with one node and abstracts the resources on the node as many *containers* that can serve different applications individually. For example, a container can be used to run a MapTask of a MapReduce program, while other containers can accommodate an MPI process. The ResourceManager manages all resources and allocates containers to running applications based on the scheduling policy.

In Hadoop YARN, each job has one ApplicationMaster, *a.k.a MRAppMaster*, and many MapTasks and ReduceTasks. Its execution includes two major phases: *map* and *reduce*. The MRAppMaster negotiates with the ResourceManager for containers. When granted the permission to launch tasks on certain NodeManagers, it launches MapTasks. Each MapTask reads one input split that contains many <k,v> pairs from the HDFS and converts those records into intermediate data in the form of <k',v'> pairs. That intermediate data is organized into a Map Output File (MOF) and stored to the local file system. A MOF contains multiple partitions, one per ReduceTask.

After one wave of MapTasks, MRAppMaster launches the ReduceTasks, overlapping the reduce phase with the map phase of remaining MapTasks. A ReduceTask is a combination of two stages: *shuffle/merge* and *reduce*. Once launched, a ReduceTask fetches its partitions from all MOFs. The ReduceTask merges the incoming partitions and reduce the number down to a threshold (*mapreduce.task.io.sort.factor*), and then enters into the reduce stage with all partitions organized into a Minimum Priority Queue (MPQ). After that, the ReduceTask traverses all the sorted <k',v'> pairs from the MPQ and applies reduce function on them. The final results are stored into the HDFS.

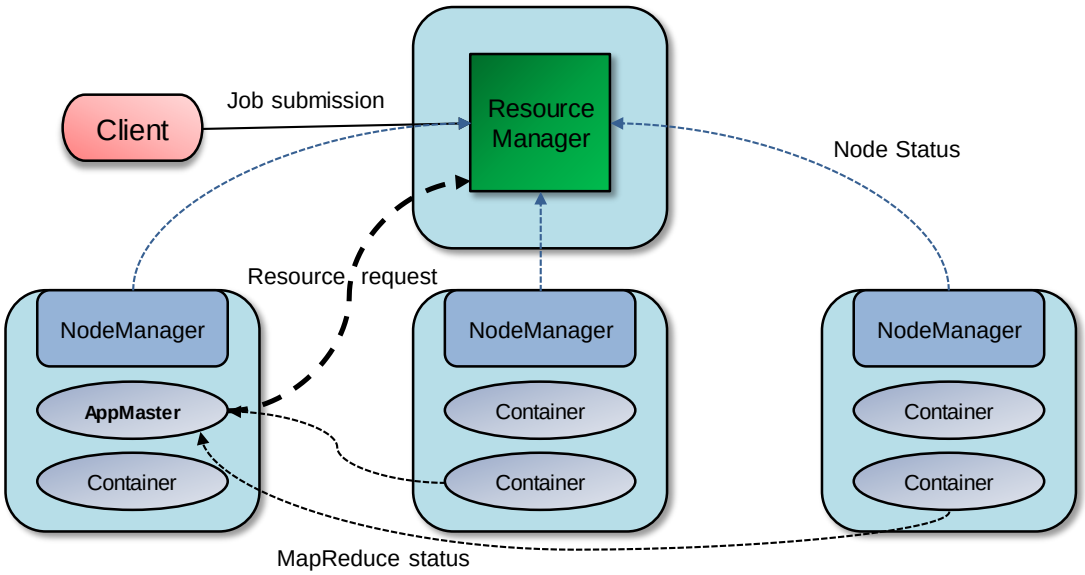


Figure 2.1: An overview of components of YARN [1].

2.2 Fault tolerance and speculation in MapReduce

As the representative implementation of MapReduce, Hadoop strives to provide outstanding performance in terms of job turnaround time, scalability, fault tolerance, etc.. Its aforementioned design can deliver good distributed computing and extreme scaling abilities. In order to achieve strong fault tolerance, YARN is quipped with other two mechanisms, i.e. data *replication* and *regeneration*. Task is properly regenerated upon all kinds of failures (network, disk, node, etc.). Even if the input data is unavailable on such failures, the

rescheduled task will have access to a replica of the data and a correct failover is still ensured. However, simply conducting failover is not good enough. YARN depends on long timeouts to declare failure for every task/component. Such long timeout is necessary to avoid false negative judgment on failure, but could prolong the recovery time when real failure occurs. So, a simple failure can lead to large performance degradation, especially for small jobs who have very short turnaround time. To make things worse, failures are prevalent in commodity cluster as found by [10, 16, 19, 23, 7, 24]. That means in overall, YARN's performance can be seriously affected by those failures. Thus, apart from data replication and regeneration, YARN also has *speculation* mechanism, which can help accelerate the detection and recovery process.

Speculation has been thoroughly studied [29, 6, 3, 5]. All of those strategies have a core similarity, which is to launch a backup copy of the slowest task in the job. Take LATE scheduler [29] for instance, it estimates the completion time for every task and uses the results to rank those tasks. The task that is estimated to finish latest will be speculated on a fast node. The number of speculations running at the same time is capped by a predetermined system parameter. This strategy, along with others such as Mantri [6], are well adopted by industry to prevent the stragglers from delaying the job performance.

2.3 Issues with the existing speculation

However, we have found that the existing speculation mechanism have some major drawbacks, which seriously impede their use in the real-world environment, where failures are prevalent.

2.3.1 Intra-node speculation

To start with, speculation is simply to make a copy of the *slowest* task. But what if all the tasks are slow? For example, if tasks are converged on one single node and for some reason the node is unresponsive, the speculator will not speculate any of those tasks

because they are equally "slow", since the comparison of progress is only made *intra-node* but not *inter-node*. The whole job will then halt until each of the tasks gets a timeout. One may argue that this convergence of tasks seldom happens. However, this phenomenon is not rare but indeed extremely common among small size jobs because of a reason that is related to YARN's fundamental design. Although MapReduce framework provides locality of tasks which can help distribute the tasks evenly across different nodes, in YARN, the ResourceManager does not follow the same principle restrictively. With its default scheduling policy (capacity scheduler), it requests several containers at once from one NodeManager and when it gets enough containers for the job, it stops requesting. When the job is small (so it does not need many containers), the MapTasks will have a very high probability of residing on the same node. This design of ResourceManager is good for YARN's extreme scalability, but unfortunately downgrades the effectiveness of speculation.

2.3.2 Prospective-only speculation

Secondly, current speculation is made only among the running tasks. If a task is finished, it will be excluded from the candidates of speculation. Intuitively, it is a reasonable strategy since completed tasks should have no way of straggling the job. However, MapReduce workflow typically requires the use of intermediate data that is produced by the completed MapTasks. It will be a problem if those intermediate data are lost and the job will be held up until it finally finds out that the intermediate data is permanently lost. Thus, completed tasks can also become stragglers but the current speculation mechanism is unable to address that. In other words, the existing speculation mechanism can only make *prospective* copies of tasks, but not *retrospective* ones.

2.3.3 Non-parallel speculation

Moreover, speculations are carried out only one at a time, and another one after a certain time period (15 seconds by default in YARN). This is a desired feature for not overwhelming

the cluster with needless speculated tasks. But it also delays the necessary ones that are surely stragglers, such as the tasks on a failed node. Waiting for those tasks to be speculated one by one can seriously degrade the performance, especially for small jobs. For example, if there are 12 MapTasks on the failed node, there will be at least 180 (12×15) seconds for the last MapTask to wait idly before it can get speculated. That is, comparing to the short turnaround time of small jobs such as those shown in Fig. 1.1, too much to endure. During that time, the job is held up by those tasks and only after the completion of the last speculation task, the job can then proceed.

2.3.4 Non-work-conserving speculation

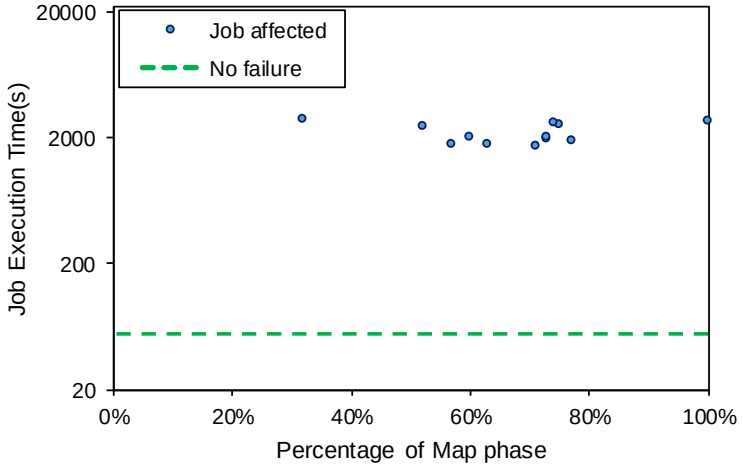
Speculation takes care of delayed/failed tasks in a way that everything starts over from scratch. But sometimes it is just ineffectual to simply discard progress. Considerable amount of progress can be reused for the newly launched speculation. Take MapTask as an example, although the turnaround time of MapTask is relatively short, the ratio of its failure is significantly higher than other type of task failure, as reported by [16]. It can be highly beneficial to integrate a mechanism that conserves task progresswith the speculation framework, so that the new speculation copies will cost a lot less time than starting over.

2.4 The breakdown of existing speculations

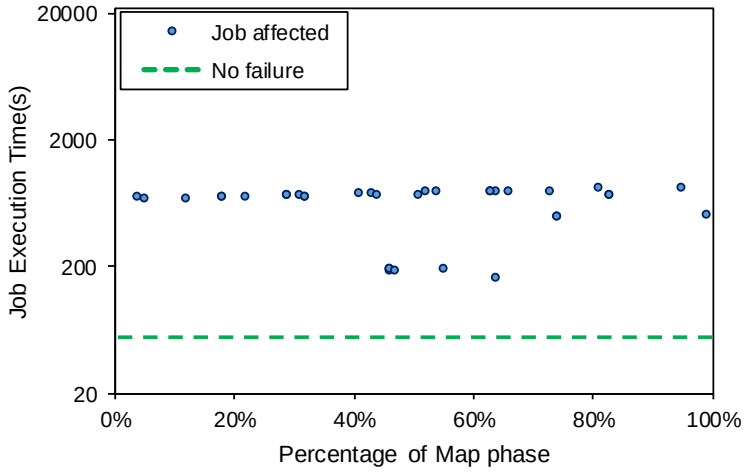
Next, we will demonstrate in detail how the above issues can impact the performance of existing speculation mechanism. We take YARN as a study case, and show how it can be affected in the presence of node and task failures.

2.4.1 Node failure

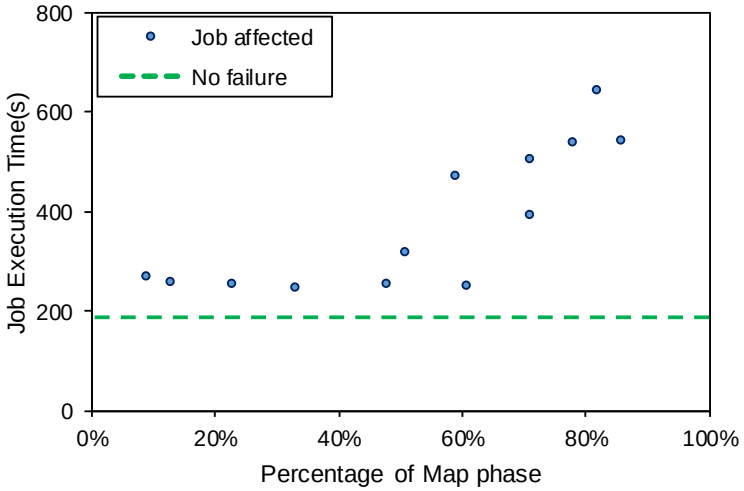
Fig. 2.2 shows the execution time of individual jobs in our test. Each dot represents a job with node failure and its turnaround time, and the normal job time is indicated by the baseline below. Firstly, Fig. 2.2(a) are the results of 1GB jobs under node failure at



(a) Small jobs of original YARN.



(b) Small jobs of modified YARN.



(c) Large jobs of modified YARN.

Figure 2.2: Running time of MapReduce jobs under node failure at different spot.

different progresses of map phase, (detailed experimental setup can be found in Section 4). The results is as astonishing as it stands, most of the jobs take time that is orders of magnitude longer than the failure-free case. There is an unusual issue behind this. YARN needs to clean up the container after a task attempt is done. But if the attempt was running on a failed node, YARN will keep trying to connect the corresponding NodeManager, which is currently unavailable, and finally throws a timeout exception after a period of time. The configuration parameters that affect the time duration are the IPC connection settings (*ipc.client.connect.**). In default, it will take about 30 minutes for connection retry. During this time, the job will not finish successfully, even both map and reduce's progresses may have already reached 100%.

But this long timeout for container cleanup is far from the sole reason that hurts the job performance. Fig. 2.2(b) is result of the control group that is without the issue of container cleanup. We excluded the problem by modifying YARN's default retry policy and then conducted the same set of tests (note that we have also excluded this problem with the introduction Fig. 1.1 and 1.2). But we could still observe large performance degradation compared to the normal job running time. The culprit here is with comparison-based speculation as we discussed before. When node contains all the MapTasks and it becomes unresponsive, the speculator will not speculate any of those MapTasks and will wait for 600 seconds for them to get timeouts.

However, this delay of MapTask timeout explains only a part of the running cases. We can see that if node failure happens on 40% to 60% of the overall map progress, some jobs ended only slightly slower than no-failure case. This is because that as map phase proceeds, different MapTasks' progress rates can be uneven, meaning that some MapTasks can be much faster than others, finally to an extent that triggers the speculation of the slow tasks. So when a node failure occurs during this time, the MapTasks on the failed node are stalled, but the speculations on other node, if there are any, will continue. When those speculations'

progress rates are high enough, they will in turn trigger the speculations of MapTasks on the failed node, and proceed normally thereafter.

But if node failure occurs during even later phase of the map progress, the disadvantage of the prospective-only speculation starts to appear. As many MapTasks have now been completed, the ReduceTasks that are trying to fetch those MOFs will have fetch failures since the MOFs are unavailable on the failed node. After the number of fetch failures for one MapTask exceeds a limit, the MapTask will be declared failed and a new attempt will be scheduled. This seriously stalls the overall job progress because ReduceTasks are idle during this time. To make things worse, if the fetch failures experienced by a single ReduceTask exceeds another limit, that ReduceTask can also be declared failed and rescheduled. Subsequently, if the corresponding MapTasks are not timely speculated, the rescheduled ReduceTask will have a second fetch failure and thus be scheduled for a third time.

On the other hand, comparison-based speculation is also broken down in this phase. As we know, reduce phase does not require the completion of map phase, ReduceTask can start running when one wave of MapTasks is finished. But if the job has only one ReduceTask (often the case for small jobs) and it is on the crashed node, it will certainly not be speculated since it has no other ReduceTask to compared to. The entire job will halt until the ReduceTask gets a timeout (600 seconds by default, too).

What if the data size is larger? As we further neglect the effects of MapTasks convergence, the cost of node failure on the map phase is still significantly high. As we can see from Fig. 2.2(c), which is the results of the same test but with 10GB of input, the running time of most failure cases nonetheless is more than twice as much as the no-failure one. Note that, right now the number of MapTasks is large enough so they were assigned evenly to different nodes. Thus, the speculator can successfully speculate the MapTask that reside on the failed node as soon as it detected that it is slower than others. But we found that the majority of jobs still suffer various performance degradation. The causes are similar:

- Single-filed speculation is not effective. In Fig. 2.2(c), most of the jobs in early phase suffer from this cause. The jobs are slightly slower than the failure-free case.

- Comparison-based speculation may be invalid for multiple ReduceTasks, too. We already show that the failure of only one ReduceTask will cost us for waiting 600 seconds as the ReduceTask timeout. In fact, if a crashed node contains multiple ReduceTasks, there is a chance that the progress of remaining ReduceTasks is not "slow" enough for them to get speculations. This is contingent on the specific speculation algorithm. In Fig. 2.2(c), our experimental results show that such jobs will take much longer than others to finish.

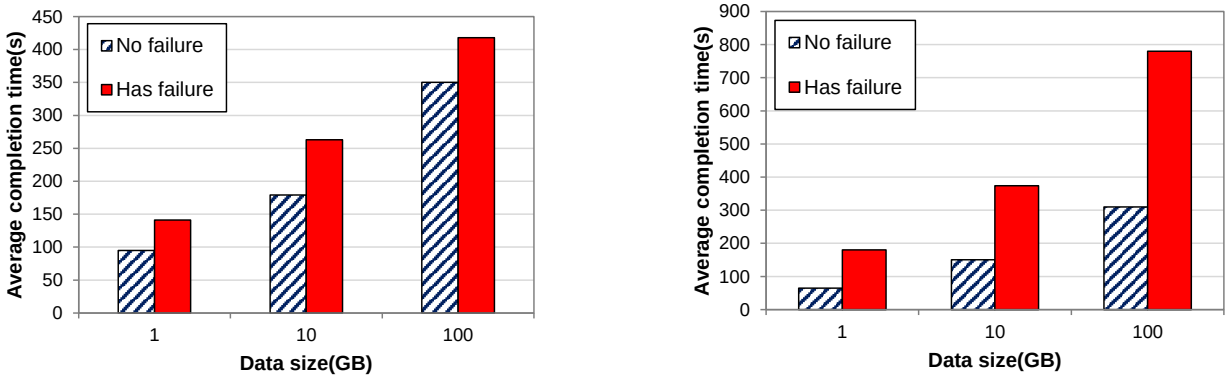
- Many test cases in Fig. 2.2(c) also suffer from the problem of prospective-only speculation. We can see that even if the input size is larger, the cost of resuming the completed tasks is still unbearable compared to normal turnaround time.

Performance variation: Another bad outcome of the speculation breakdown is that it has impacts on the predictability of job performance, which is always a concern in the heterogeneous environment [22, 29, 27]. As we can observe from Fig. 2.2, the node failure can lead to very unstable job performance. Since a common failure can cause such distinct performance variation, job executor may never schedule their jobs properly.

2.4.2 MapTask failure

As we discussed before, the failure handling of Map- and ReduceTasks in YARN adopts a straightforward re-execution mechanism. When tasks fails, all of its running status and progress are discarded. This approach works well when the tasks are small in terms of running time, but is unsuitable for large task whose turnaround time is relatively long. Especially for map-heavy workloads such as Wordcount, Pagerank or K-means Clustering, as investigated by [15], the penalty of MapTask failures is unbearable. Fig 2.3 shows such impact. We evaluate the efficiency of YARN's recovery efficiency under two failure scenarios, i.e., *disk*

failure and *bad records*. Both types are extremely common in the real-world deployment [23, 8, 10, 26]. For disk failure, we disable the disk I/O on one node, run Wordcount jobs and compare the average performance with normal runs that is without failure. The results are shown in Fig. 2.3(a). The disk failure downgrades the job completion time by nearly 50% for 1 Gb jobs, and about 15% for the 100 Gb jobs. Then we introduced the other failure type, i.e., one bad record in each map input split. A bad record results in failure of the corresponding MapTask and rescheduling of its new attempt. Fig. 2.3(b) shows the results, which are similar to the disk I/O test on the left. But as we can see, bad records have even worse performance degradation. All test cases cost 100% or more turnaround time than the case of no failure.



(a) Disk failure

(b) Bad records

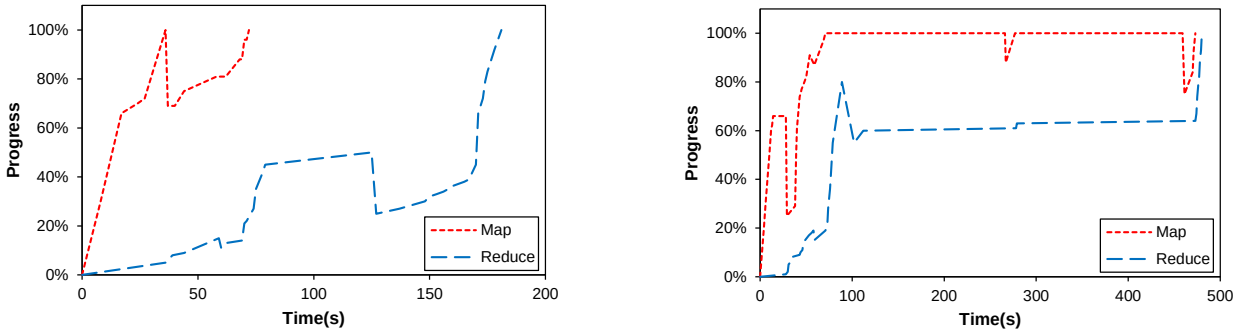
Figure 2.3: Issues with MapTask recovery.

2.4.3 Change timeouts?

We have already seen that the default timeouts are too long for MapReduce framework to detect failures and thus can prolong the speculation/failover process. One may assume that the problem can be solved by simply decreasing the timeouts. However, those long timeouts are necessary for it to adapt to heterogeneous environment since the network situation is unknown and unstable. If we set the timeouts to some values that are too small, tasks could be falsely declared failed when the network is just experiencing some temporal

congestion. Fig. 2.4(a) shows an example of that. We changed YARN's timeout for MapTask/ReduceTask to 5 seconds and run it in an unstable network where a lot of networking delays, varying from 1 to 8 seconds, are generated randomly. We can see that the progress of both map and reduce are seriously affected. They either stall at the delays when they need network transfer (e.g. around 100s, some ReduceTasks are still shuffling), or even backslide if the delays exceed the timeout and the corresponding tasks are declared failed.

Thus, simply changing configurations is not feasible for unstable networks. Let alone if there are more unstable factors such as unstable nodes in the environment. Fig. 2.4(b) is another example of MapReduce jobs with both network delays and node failure. It shows that many MapTasks have failed because of network delays, and the progresses are further impeded by a node failure at about 100s, after which one ReduceTask is declared failed immediately but the reduce phase cannot proceed because it needs an MOFs on the lost node. So it keeps fetching the MOFs until a fetch failure is incurred. Then it continues to request other lost MOFs and undergoes two more fetch failures.



(a) Unstable network

(b) Unstable network and node

Figure 2.4: The progress of map and reduce.

2.5 Related Work

Quiane-Ruiz *et al.* in [18] introduced RAFTing MapReduce for preserving the computation status of MapTasks and replicating the MOFs to reduce side. This design avoids the

re-computation of MapTasks on the failed node, but it requires pre-assignment of ReduceTasks and additional network overheads. Moreover, it addressed only one negative factor of node failure which is the lost of MOFs, without taking care of failures happened on early map phase and looking into the detection method of the failures. Thus, it would still suffer from the performance degradation discussed in this paper and have problem dealing with real-world failure scenarios. Furthermore, assign recovered MapTasks strictly on previous nodes have potential issue of repeating failures since the node that contains the failed MapTasks has higher probability of failing the recovered MapTasks for the same cause again.

Dinu *et al.* in [13] conducted a comprehensive study of impacts of node failure in MapReduce framework. They revealed that a single node failure can significantly downgrade MapReduce applications' performance. Specifically, they found that the failure of the node contained ReduceTasks can infect other healthy tasks and nodes, causing drastic performance degradation. Our previous work [28] have revealed issues similar to them, which we referred to as "failure amplification", and more importantly, we also provided techniques to address the issues. But both works did not look into the failures happened on map phase. This paper is orthogonal to those works by analyzing and addressing map phase failures.

Ananthanarayanan *et al.* [3] digged into the straggler problem in small size jobs in MapReduce. They demonstrated that to aggressively speculate a clone for every task is a good way to ameliorate the performance degradation that stragglers may impose on the MapReduce applications. Although their design can also be helpful for solving the node failure problem found in this paper, it has obvious downside that cloning every task will incur a lot more unnecessary resource consumption and network overheads, especially for a shared MapReduce cluster that is already heavily loaded as discussed in [9, 20, 25]. In addition, those extra overloads are needed in every job execution, despite the nodes were being faulty, just delaying, or not having any problem at all. Without handling failures respectively, relying on such aggressive speculations for fault resiliency is unpractical.

Chapter 3

Design and Implementation

In this chapter, we will unfold our designs and some important implementation features in order to tackle the aforementioned issues of existing speculation.

3.1 Overview of FARMS

We start with our new speculation design that is **F**ailure-**A**ware, **R**etrospective and **M**ultiplicative **S**peculation (FARMS).

First of all, we have to understand that the existing speculation's inability to address the aforementioned issues roots in its unawareness of the failures. Because the speculator can only coordinate the tasks progress at task level, it does not understand the node level status very well. Thus, a simple node exception can straggle the whole job. Our solution is straightforward. In FARMS, we leverage the failure information that collected by a global analyzer(Section 3.2). The demonstration is shown in Fig. 3.1. In the figure, the existing speculation is shown on the left and our new speculation is shown on the right. Each box represents a running task and its brightness indicates the task's progress. Tasks are associated with their host nodes, and are speculated collectively and incrementally when a malfunctioning node is detected.

Secondly, in FARMS, the completed tasks are not excluded from speculation candidates. We add transitions that can speculate the completed tasks that associated with a failed node. When the speculation task attempts have been completed, ReduceTasks will be notified to fetch MOFs from the new task attempts instead of the original ones.

Thirdly, we change the single speculation to batch speculation, meaning that when we decide to launch speculations, they can be launched all at once. But note that such

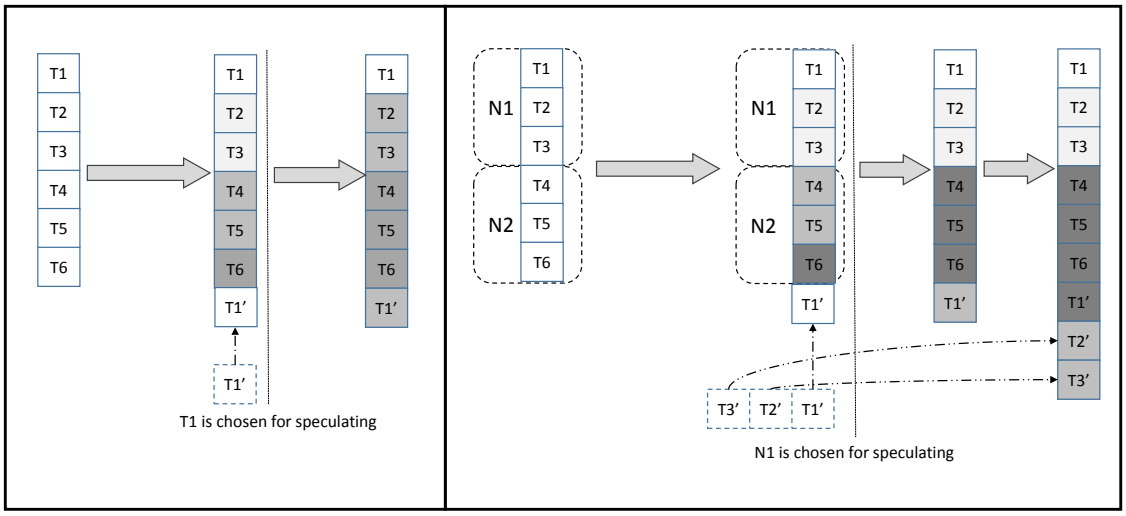


Figure 3.1: Traditional speculation and our optimized speculation.

speculations can be costly sometimes because if we make false-negative decision on the node exceptions, there will be a lot of unnecessary additional resource consumption that comes with the speculation. Although we have optimized our decision algorithm (Section. 3.4), we still want to minimize the cost. Thus, we incorporate an exponential speculation mechanism to FARMS that can multiplicatively make speculation copies of the stragglers. Upon the detection of node exceptions, the number of tasks to speculate increases in exponential order. The condition to keep making speculation copies is contingent on the liveness of the corresponding node. For example, if one node is unresponsive, we first speculate 2 tasks. We monitor the progress of the problematic tasks and if they remain slow or unresponsive, we speculate another 4 tasks.

3.2 Centralized Failure Analyzer

As discussed before, we need FARMS to be aware of the system exceptions to facilitate the new speculation process. But actually, as a comprehensive framework that considers fault-tolerance as one of its design goals, YARN does not lack awareness of its own real-time running information. It has an *application history service* for storing generic and per-framework information of both running and completed applications, and also a package

named *yarn.api.records* that keeps a comprehensive set of records of the status of system components. However, their primary purpose is to serve simply as an information bank, which is referenced only for some post-running analysis. Moreover, the information is too generic and not very useful for analysis of specific types of failure. Simply put, YARN needs a standalone server whose responsibility is to gather only the information of system problems and guide the failover.

Fig. 3.2 shows the overview of our framework. We have a Centralized Fault Analyzer (CFA), which is initialized with ResourceManager as a *CompositeService*. It can request application information such as job IDs, task IDs, container assignments, etc. from the ResourceManager. System problem is recorded by individual component who discovers it. We choose HDFS to store the problem information because of its high availability and global accessible feature. The types of problem include NodeManager heartbeat lost, MOFs fetch failure, connection failure, etc.. CFA gathers the information and provides the analysis result back to individual components via writing to HDFS. Those components will consume the results and adjust their functions accordingly. The extra I/O is lightweight and it incurs minimal overheads, which we will demonstrate in Section 4.

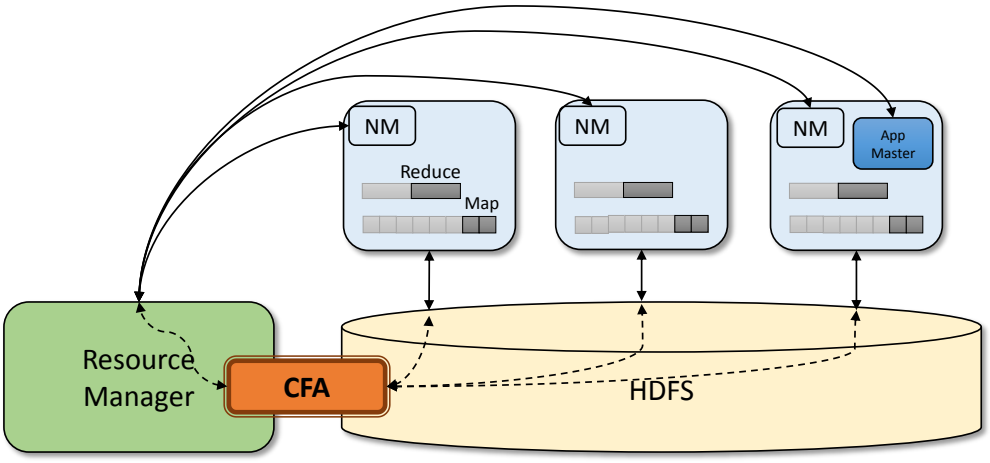


Figure 3.2: Information exchange with the involvement of CFA.

Failure of CFA: Failure can also happen to CFA itself. However, our design of CFA guarantees its availability upon failure. Since nearly all useful information is stored onto

HDFS with replica, we simply depend on the fault resiliency of HDFS itself. If ResourceManager finds CFA unresponsive, it will restart it and the new CFA will extract the previous status from HDFS.

3.3 Work-conserving speculation of MapTask

Different from node failure, MapTask failure is often detected immediately after it experiences something wrong, e.g., I/O exception, spill failure, etc.. As soon as such an exception is thrown, the MapTask will be declared failed and a new attempt is scheduled. Thus, it does not have the aforementioned issue, i.e., a trade-off between tolerating transient failure with timely failover from permanent failure. However, unlike handling node failure in YARN which is on the extreme of tolerating transient failure, handling MapTask failure is on the other extreme of just providing quick failover regardless of its real status. The MapTask failure can occur because of distinct reasons. In some situations, it is better to reallocate the new attempt on another node because the original node may already be problematic and that could well be the reason the original attempt was failed in the first place. But in other situations, the original node may have nothing wrong but the MapTask failed for transient causes such as disk I/O faults.

Thus, our approach to failure handling of MapTask is to speculatively launch two attempts of the failed MapTask on its original node and a new node. We log useful information such as the spill path and offset of processed input split of every MapTask, when the task is spilling intermediate data to the disk. So the recovered MapTask can avoid repeating the completed work done by previous attempt. In order to avoid making useless speculation copies, we will first consult the CFA about the status of previous node. If the node is not in doubt, two speculations of the MapTask will then be scheduled. Otherwise, an additional speculation is not needed, and we will spawn them on a fast node, just as the current YARN does.

3.4 FAS - A New Scheduling Policy

Finally, we propose a new scheduling process that leverages the analytics information. We name it as the *Fast Analytics Scheduling (FAS)*. As discussed before, the trade-off between speeding up failure detection and truncating resource consumption is really important. In FAS, we use a dynamic threshold to determine if a failure should be speculated or not. The positive results will be added to the speculated list, and the negative ones will be ignored.

Before we go into details, some design principles of the new algorithm need to be sorted out. In general, the algorithm should meet the following requirements.

1. The decision made in most cases should decrease the job execution time.

2. The decision should be as accurate as possible, avoiding too much unnecessary additional resource consumption.

3. Even when the decision misjudges the situation, the impact to job performance should be cheap.

4. It can adjust to different failure patterns.

To meet requirement 1), we need to keep the rescheduling of tasks aggressive enough for it to gain any performance improvement. That means, the threshold cannot be too large, otherwise there will be no difference from the default timeout mechanism and will hurt the performance as well. But to meet requirement 2) and 3), we need the threshold to be dynamically adjusted according to the specific network & hardware conditions. So we have to limit the number of decisions made and so is the speculations imposed to the job execution. Finally, we need to tune the parameters in our algorithm for the requirement 4). By doing so, it can have optimal performance under different failure scenarios.

Our overall algorithm is described in Algorithm 1. We take a simple heuristic method that deduces the status of the node. As discussed before, in order to fit the heterogeneous environment better, YARN declares the node death only after waiting for a long period of

timeout (600 seconds by default). But we do not need to wait that long to provide equal or better fault tolerance. We can just speculate those tasks on the node while still waiting for it to resume responsive again. So if a node is detected malfunctioning, we aggressively speculate all the tasks on it. However, this approach incurs additional resource consumption on other nodes. This is necessary when the node failure is a permanent one because all the computations need to be conducted again anyway, but is not so when the failure case is a transient one. Thus, we keep monitoring the node status after the speculations are launched. If finally it is revealed that the failure is a transient one, CFA will measure the time duration that the node remains lost. After that, the threshold to which we declare a node failure will be dynamically adjusted according to the most recent lost time of that node. By repeating this step, each threshold is well adapted to the specific node environment. However, if the network is temporarily down for a very long time, we do not want the threshold to be too high because it would still be better to proactively schedule them rather than waiting for a long time and then schedule, which in some sense, falls into the same timeout mechanism of default YARN, which is already shown to be extremely inefficient. So we decrease the threshold every time it makes a right prediction, keeping it short enough to gain performance improvement. Intuitively, it is good to blacklist the node that experiences frequent failures so the majority of jobs will be running on a more healthy cluster. However, we do not allow that because commercial cluster trace [3] shows that the problematic nodes are very common and they are typically distributed evenly across a cluster. This is partly due to that the faulty nodes have already been blacklisted in the periodic checks.

Algorithm 1 Enhanced Scheduling Policy with FAS

$N \leftarrow$ {The compute node}
$T_N \leftarrow$ {Tasks running on N}
$threshold \leftarrow$ {Time threshold to decide a node failed}
$Fail_{cur} \leftarrow$ {Number of nodes currently already speculated}
$Fail_{max} \leftarrow$ {Maximum number of node failures allowed}
$P_a = 1.5$ {Predetermined parameter a for updating the threshold}
$P_b = 0.5$ {Predetermined parameter b for updating the threshold}
if N's lost time $> threshold$ **then**
  if $Fail_{cur} \leq Fail_{max}$ **then**
    $Fail_{spec} = Fail_{spec} + 1$
    for all $task \in T_N$ **do**
      schedule new attempt of $task$ on other node.
    end for
    continue to monitor the node status...
    if N resumes responsive **then**
      $t =$ time length of the node's loss of connection
      $threshold =$ average of last recent five values of $t \times P_a$
    else
      $threshold = threshold \times P_b$
    end if
  end if
end if
if MapTask t_m on N failed **then**
  if N is problematic **then**
    schedule new attempt of t_m on other node.
  else
    schedule new attempt of t_m on N.
  end if
end if
{When the job is done, document the longest connection lost time t of N during the job.}

Chapter 4

Evaluation

4.1 Experimental Setup

Hardware Setup: All experiments are finished on a cluster of 21 server nodes that are connected through 1 Gigabit Ethernet. Each machine is equipped with four 2.67 GHZ hex-core Intel Xeon X5650 CPUs, 24GB memory and one 500GB hard disk.

Software Setup: We use the latest release of YARN 2.6.0 as the code base with JDK 1.7. One node of the cluster is dedicated to run ResourceManager of YARN and NameNode of HDFS. The key parameters of the whole software stack are listed in Table 4.1,along with the tuned values.

Table 4.1: List of key YARN configuration parameters.

Parameter Name	**Value**
mapreduce.map.java.opts	1536 MB
mapreduce.reduce.java.opts	4096 MB
mapreduce.task.io.sort.factor	100
dfs.replication	2
dfs.block.size	128 MB
io.file.buffer.size	8 MB
yarn.nodemanager.vmem-pmem-ratio	2.1
yarn.scheduler.minimum-allocation-mb	1024 MB
yarn.scheduler.maximum-allocation-mb	6144 MB

Benchmarks: Through out the whole experiments we have selected three representative MapReduce applications, including Terasort, WordCount, and Secondarysort.

4.2 Fast failover with CFA

We have tested our framework against the task and node failures to examine its efficiency to tackle down the performance degradation. Since node failure has very different impacts on

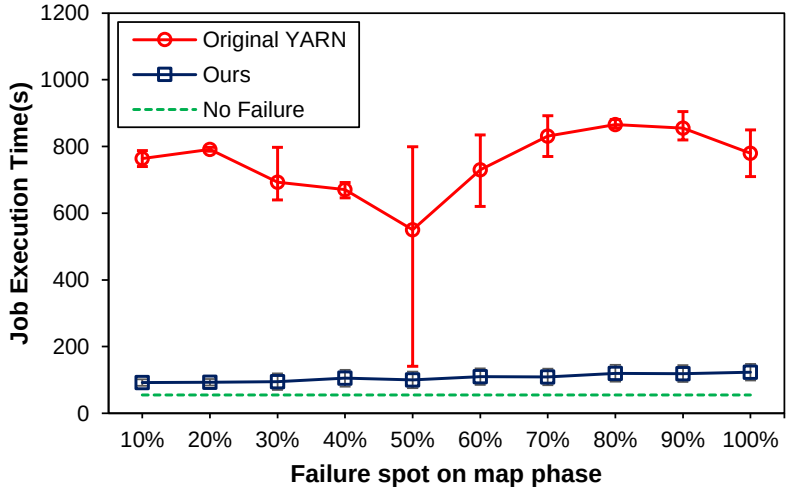
the job execution (Section 2) due to different job size, we conducted two sets of experiments on jobs that have small or large input size. For small size jobs, the input size is 1GB and we crash a node that hosts the MapTasks at a different spot in the map phase. For large size jobs, the input is 10GB and the node to crash is picked randomly.

Fig. 4.1 and Fig. 4.2 shows the comparison between the original YARN and our framework against node failure at different phases. At each spot, we test it at least three times and get the average. The highest and lowest execution time is also shown using an error bar. Since the prolonged job finish delay caused by YARN's retry policy is somewhat "unusual" because it can be tuned by simple re-configuration, so in our experiment we have neglected this issue by modifying YARN's default retry policy and still regard it as the "Original YARN" case.

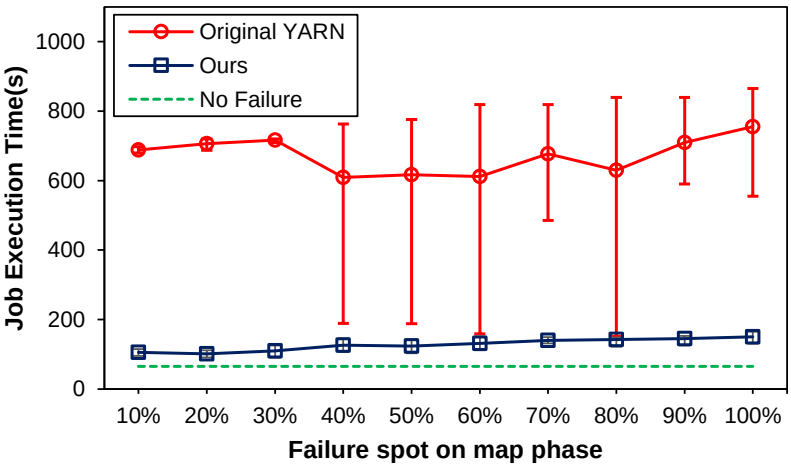
From the figures, it is clear that for small size jobs, the performance improvement is striking. Our framework speeds up the job execution time by almost an order of magnitude. For large size jobs, it can also tackle down the failure delay significantly. For all cases, our timely detection and fast speculation produce results that are comparable to the no failure case, and they also show much less variations, which can provide constancy and predictability for the job executor.

4.3 FAS evaluation

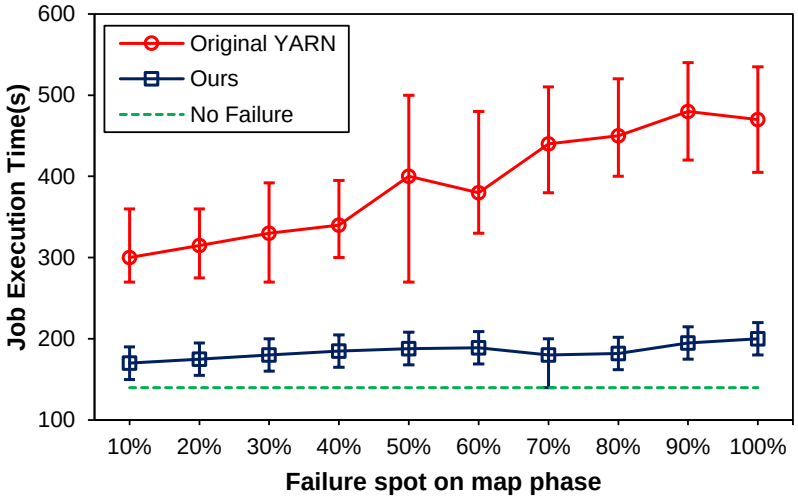
To see if our framework can adapt to real-world environment where transient network congestion are common, we generate several evaluation cases with different experimental settings. First, we test our framework against one single problematic node. The time duration t for the connection lost that we simulate is generated according to the Poisson Distribution where $P(X = t) = \frac{e^{-\lambda}\lambda^k}{k!}$. The real-world node failure rate is empirically set according to the statistics from [12, 10], from which we know that the average of node failure per job is 5 and the average number of nodes in one job is 268. Since we use 21 machines, we set the probability of one random node failure per test to 0.4 and the probability of one



(a) Terasort

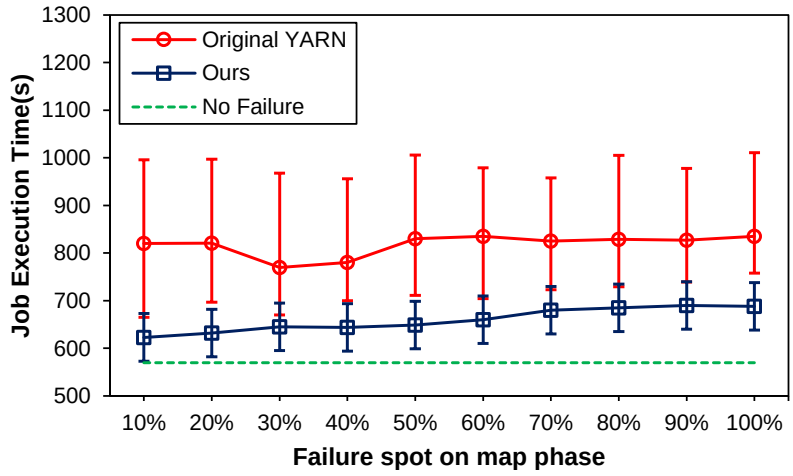


(b) Wordcount

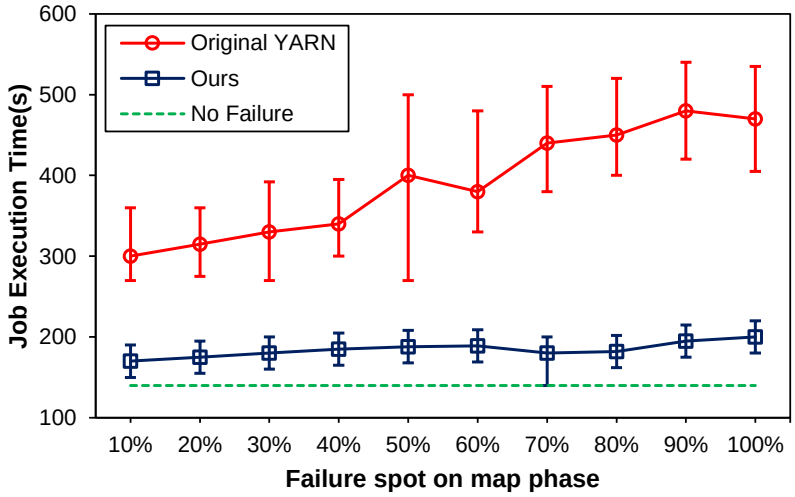


(c) Secondarysort

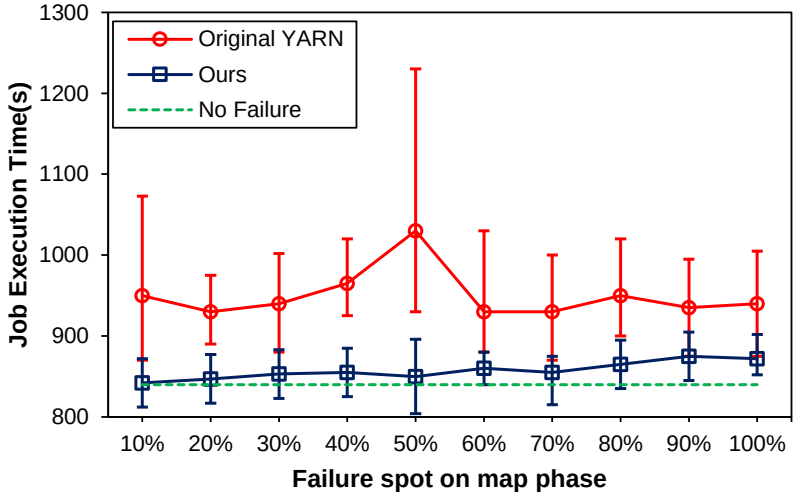
Figure 4.1: Failure recovery of small size job.



(a) Terasort



(b) Wordcount



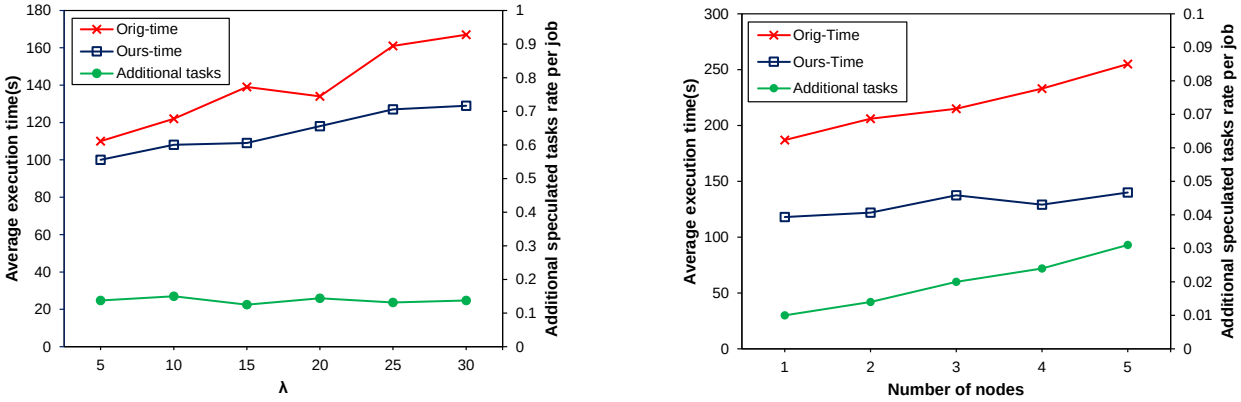
(c) Secondarysort

Figure 4.2: Failure recovery of large size job.

particular node failure to 0.02. We conducted one set of experiment for each value of λ and for each set, we randomly generated a network delay or a node failure in each test. Twenty tests were conducted for each set. The maximum λ in our experiment is set to 30.

Note that we do not want λ to be too large because then the difference between a transient failure and a permanent one will be eliminated. To normalize our tests and make the results comparable, the delay and failure all occur at the same phase of the job. The same experimental setup is used for both the original YARN and our framework. Since our metrics include both job performance and resource consumption, we plot the average job running time and the wrongfully speculated tasks per job for every experiment setup.

Fig. 4.3(a) shows the results of the experiments introduced above. The performance of our framework is significantly better than the original YARN in all experiment sets. Note that the more node failures per set, the more performance improvement we will get from FAS. But even if there is no node failure, our framework can still outperform the original YARN because the delays are balanced by the early speculations. Also, in overall, these early speculations incurred by the network delay remain at very low level.



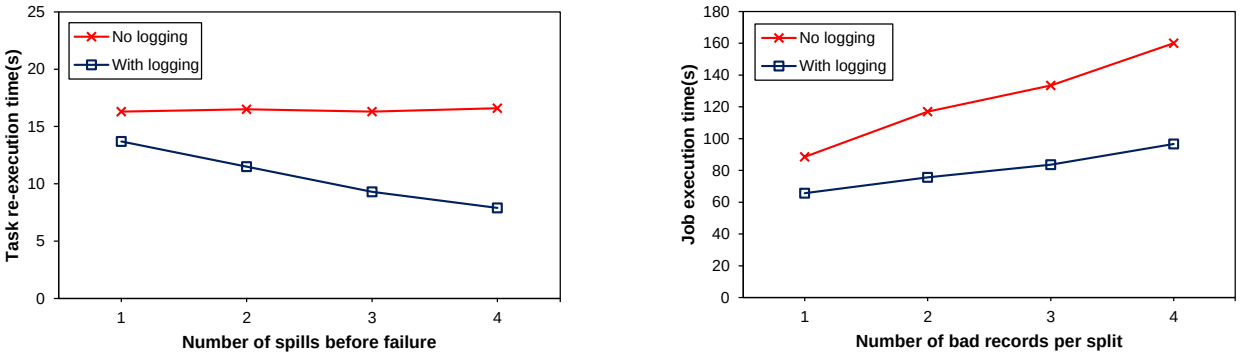
(a) Only network delays

(b) Network delays and node failures

Figure 4.3: Using FAS for fast and efficient recovery against system variations and failures.

It is understandable that a single problematic node does not cause much trouble because even if we speculate it every time despite of its real status, there will not be excessive additional overheads imposed on the whole cluster's computation power. Thus, we conducted another evaluation that generates network connection delay and node failure on multiple

nodes. In this test, we use Wordcount as our benchmark and 10 Gb of input. The results are shown in Fig. 4.3(b). Our framework still has significantly better performance compared to the original YARN when there are multiple network delays or node failures. The performance is not as steady as in Fig. 4.3(a) because the factors that straggle the larger size jobs, as discussed in Section 2. But there is still an obvious trend that when the number of nodes increases, the average running time of original YARN increases along because of more chance of having real node failure. Also, if number of nodes is larger, the number of our additional speculations slightly increases, but still remain under a tolerant level.



(a) Single MapTask failure

(b) One bad record per split

Figure 4.4: Performance improvement with work-conserving speculation.

MapTask Logging: We also conducted experiments to demonstrate the benefits of preserving MapTask progress. Firstly to see its impact to a single MapTask execution, we inject failure by incurring a disk write exception to the MapTask. Fig. 4.4(a) shows the performance improvement of MapTask logging to the new attempt of the failed MapTask. The actual benefits gained from it depends on the task progress when failure happens. When the MapTask progress is higher (so there are more re-usable intermediate local files), the re-execution of the task needs substantially smaller amount of time.

We then demonstrate the benefits of MapTask logging to the total job execution. Fig. 4.4(b) shows the improvement of performance of a MapReduce job under different number of bad records per input split, which leads the MapTask failure for a different number of times. It is clear that the preserving of completed progress of MapTasks have a significant

boost for the overall job execution time. With more bad records within the input files, the more performance improvement we will gain from it.

4.4 Overall evaluation

The above experimental results show the advantages of our framework in terms of the turnaround time of individual task/job. But we also want to know the overall performance improvement of our framework for a certain workload which is composed of multiple jobs. Thus, we combine Terasort, Wordcount and Secondarysort benchmarks and run them in sequence. The size of the jobs are determined according to Table 4.2 which is based on the Facebook workload characteristics from [4]. The job arrives at random times slots following a Poisson distribution.

Table 4.2: Group of test cases.

Group	Size	Ratio
1	1 GB	85%
2	10 GB	8%
3	50 GB	5%
4	100 GB	2%

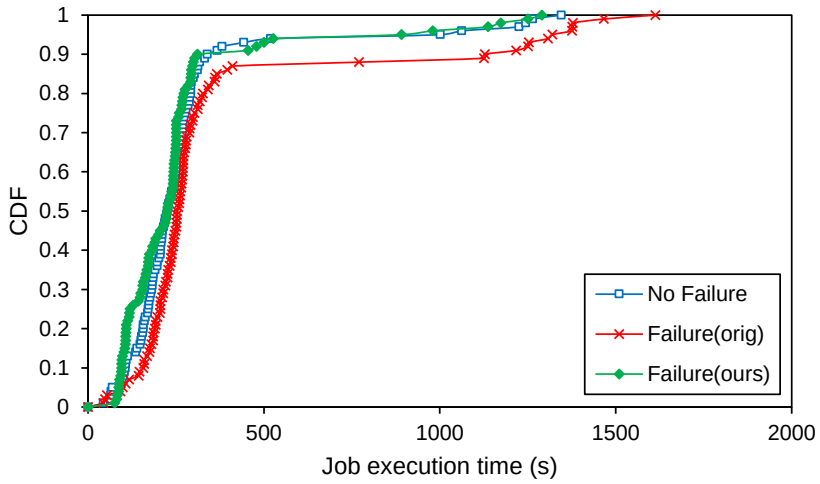


Figure 4.5: Overall performance improvement.

We then generate AM failure, task failure, node crash and network delays, each with a frequency ratio as introduced in previous sections. We repeat tests with the exact same setup (job group, failure injection method and interval) for both the original YARN and our framework. Fig 4.5 shows the results of the overall evaluation. We can see that our framework provides overall performance that is comparable to the no failure case.

For smaller jobs that are intact from node failures, all three cases are similar, original YARN with failure is slightly worse here because of the network delay and disk failure. And for jobs that are affected by node failures, original YARN performs a lot worse but our framework manages to keep their performance comparable to the no failure case. In overall, our performance is 15.3% better than the original YARN under our experimental setup.

Chapter 5

Conclusion and Future Work

This article describes the issues with the existing speculation mechanism that has long been neglected in the representative implementation of MapReduce framework, i.e., YARN. It has revealed that existing speculation has many disadvantages in front of small size jobs that have led to serious job execution delay. The disadvantages have been summarized, including the intra-node speculation, prospective-only speculation, non-parallel speculation and non-work-conserving speculation. The article also demonstrates in detail how they can cause breakdown of existing speculation in front of failures by showing the results of motivation experiments.

Based on the findings, a new design of MapReduce's speculator is proposed, including an optimized speculation mechanism, a centralized failure monitor and analyzer, a progress-conserving mechanism for MapTasks and a refined scheduling policy under failures. These techniques combined form a new job-handling framework and provide methods to tackle the breakdown of existing speculation in the presence of system failures.

The study also demonstrates the results of a comprehensive set of experiments that were conducted in order to evaluate the performance improvement of the new framework. The experimental scenarios include the fast failover test for two different job size, the real-world simulation test, the MapTask work-conserving test and the overall performance test. The results show that in all cases our framework has dramatic performance improvement dealing with task/node failures than the original YARN.

In the future, we will also try to incorporate the logging mechanism for ReduceTask. By combining the logging mechanism of Map-/ReduceTasks and our new speculator design, we

aim to incorporate a complete failure resiliency mechanism into the MapReduce framework.

Bibliography

[1] Apache hadoop nextgen mapreduce (yarn). `http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[2] Apache hadoop project. http://hadoop.apache.org/.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.

[4] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association, 2012.

[5] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: trimming stragglers in approximation analytics. *Proc. of the 11th USENIX NSDI*, 2014.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[8] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[10] J. Dean. Experiences with mapreduce, an abstraction for large-scale computation. In *PACT*, volume 6, pages 1–1, 2006.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, OSDI '04, pages 137–150, San Francisco, California, USA, 2004. USENIX Association.

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

[13] F. Dinu and T. E. Ng. 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*, HPDC '12, pages 187–198, New York, NY, USA, 2012. ACM.

[14] J. Gantz and D. Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2007:1–16, 2012.

[15] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[16] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 94–103. IEEE, 2010.

[17] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[18] J.-A. Quiane-Ruiz, C. Pinkel, J. Schad, and J. Dittrich. Rafting mapreduce: Fast recovery on the raft. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 589–600, Washington, DC, USA, 2011. IEEE Computer Society.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[20] J. Tan, X. Meng, and L. Zhang. Delay tails in mapreduce scheduling. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 5–16, New York, NY, USA, 2012. ACM.

[21] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[22] R. L. Villars, C. W. Olofson, and M. Eastwood. Big data: What it is and why you should care. *White Paper, IDC*, 2011.

[23] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204. ACM, 2010.

[24] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed systems meet economics: pricing in the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 6–6. USENIX Association, 2010.

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

[26] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[27] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–9. IEEE, 2010.

[28] Y. Wang, H. Fu and W. Yu. Cracking Down MapReduce Failure Amplification through Analytics Logging and Migration. In *29th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2015)*, Hyderabad, India, May 2015.

[29] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.