A Case Study of MapReduce Speculation for Failure Recovery

Huansong Fu
Florida State University
fu@cs.fsu.edu

Yue Zhu
Florida State University
yzhu@cs.fsu.edu

Weikuan Yu
Florida State University
yuw@cs.fsu.edu

ABSTRACT

MapReduce has become indispensable for big data analytics. As a representative implementation of MapReduce, Hadoop/YARN strives to provide outstanding performance in terms of job turnaround time, fault tolerance etc. It is equipped with a speculation mechanism to cope with run-time exceptions and failures. However, we reveal that the existing speculation mechanism has some major drawbacks that hinder its efficiency during failure recovery, which we refer to as the speculation breakdown. In order to address the speculation breakdown, we introduce a failure-aware speculation scheme and a refined scheduling policy. Moreover, we have conducted a comprehensive set of experiments to evaluate the performance of both single component and the whole framework. Our experimental results show that our new framework achieves dramatic performance improvement in handling with task and node failures compared with the original YARN.

1. INTRODUCTION

Nowadays, the society has entered into its "Big Data era." With the ballooning of the digital world's capacity, the use of big data analytics tools has been increasingly substantial. Among them, MapReduce based 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 called YARN [1]. YARN is designed to overcome scalability and flexibility issues in the first generation Hadoop.

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 copy of the straggler task that may block the job progress. The first copy of straggler that finishes first will let the job proceed. Even in the presence 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.

[1]Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Request permissions from Permissions@acm.org.
DISCS-2015, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-3993-3/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2831244.2831245

However, we have found that the existing speculation mechanism has several deficiencies, especially for small jobs. Fig 1 shows the job slowdown caused by a single node failure with a varying input size from 1 GB to 10 GB and an increasing number of tasks. We can see that to the jobs that have 1 to 10 GB input data or 10 to 100 tasks, a single node failure can degrade the job performance by a varying factor from 3.3x to 9.2x.

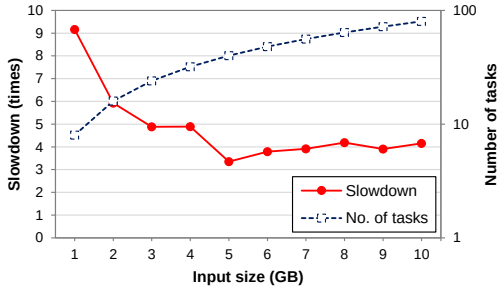


Fig. 1: Wordcount job performance when one node fails.

How serious is this impact? Although Hadoop is known for its ability for processing big data, a significant portion of jobs in industrial use are actually *small size jobs*. For example, the distribution of Facebook workloads [4] demonstrates a heavy tail tendency, in which both the number of tasks and the input size follow a power-law distribution. For Facebook workloads, about 90% of jobs have 100 or less tasks, and a majority portion having 100 Gb or less input data. Thus, a lot of jobs will suffer from the performance degradation as shown above. Note that in our experiments, we simulate only node failures, which are already extremely common. According to [10], there is an average of five node failures during one MapReduce job execution. All these evidences indicate an imperative need to revisit the existing speculation mechanism in MapReduce model.

In order to address the aforementioned problem, we have studied the inability of existing speculation mechanism in handling with failures. Then we introduce a set of techniques, including an optimized speculation mechanism and a fast scheduling policy on failures. Our experimental results show that our new speculation implementation has dramatic performance improvenment in handling with failures compared with the original YARN.

In summary, our work makes the following contributions:

- We systematically reveal the drawbacks of current speculation mechanism and analyze the impact, cause and implication of node failure in great detail.
- We improve the efficiency of YARN's existing speculation against failure by introducing our a speculation scheme FARMS

(Failure-Aware, Retrospective and Multiplicative Speculation).

- We involve YARN with a heuristic failure scheduling algorithm named Fast Analytics Scheduling (FAS) to work with FARMS, which adds strong resiliency to the heterogeneous real-world environment.

- We demonstrate that the implementation of FARMS and FAS improves YARN's performance significantly under failures, especially for small jobs.

This paper is organized as follows. Section 2 details our findings on the existing speculation mechanism with experimental results. Section 3 introduces our solution designs of FARMS and FAS. Section 4 presents the evaluation results of our implementation. We survey related work in Section 5 and conclude the paper in Section 6.

2. BACKGROUND AND MOTIVATION

2.1 Fault Tolerance and Speculation Mechanism of YARN

As the representative implementation of MapReduce, Hadoop strives to provide outstanding performance in terms of job turnaround time, scalability, fault tolerance, etc. In its current version called YARN, each job is comprised of one ApplicationMaster, *a.k.a AM*, and many *Map-* and *ReduceTasks*. 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, AM launches ReduceTasks, overlapping the reduce phase with the map phase of remaining MapTasks. Once launched, a ReduceTask fetches its partitions from all MOFs and applies reduce function on them. The final results are stored into the HDFS.

This design provides good distributed computing and scaling abilities. In order to achieve strong fault tolerance, YARN is equipped with data *replication* and *regeneration* mechanisms. A task is properly regenerated upon various failures (network, disk, node etc.). Even if the original input data is unavailable because of failures, the rescheduled task will have access to a replica of data and a correct failover is still ensured. However, to simply conduct failover is not good enough. YARN depends on long timeouts to declare failure for every task. Such long timeout is necessary to avoid false positive decisions on failure, but it could prolong the recovery when real failures occur. 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, 18, 21, 23, 7, 24]. That means in overall, YARN's performance can be seriously affected by failures if it solely relies on the naive task-restarting mechanism. Thus, apart from it, YARN also has a *speculation* mechanism which can help accelerate the detection and recovery process.

Speculation has been studied previously [28, 6, 3, 5]. Most of these strategies launch a backup copy of the slowest task for a the . The LATE scheduler [28], for instance, estimates the completion time for every task and uses the results to rank those tasks. The task that is estimated to finish the last will be speculated on a fast node. After a given time interval, YARN will search again for slow task to speculate. This strategy, along with others such as Mantri [6], are adopted by industry to prevent the stragglers from delaying the job performance.

2.2 Issues With The Existing Speculation

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

2.2.1 Intra-node only

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 every single task of one job is *converged* on one single node and for some reason the node becomes unresponsive (such as node crash or connection lost), the speculator will not speculate any of those tasks since they have relatively the same progress. The speculator cannot tell which task is slower so the whole job will halt until each of the tasks gets a timeout and then be executed from scratch again. Clearly, those timeouts should be avoided by early speculations as soon as YARN recognizes that the tasks on one node are all slow. However, in the existing speculation, the speculating decision is only made by *intra*-node task-progresses but not *inter*-node status.

One may argue that the task convergence can seldom occur because it counters with the distributed computing nature of MapReduce. However, we found this phenomenon is not rare but indeed extremely common among small size jobs. The reason is that although MapReduce framework provides locality of tasks which can help distribute the tasks evenly across different nodes, in practical implementation such as YARN, its scheduler 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 causes task convergence and downgrades the effectiveness of speculation.

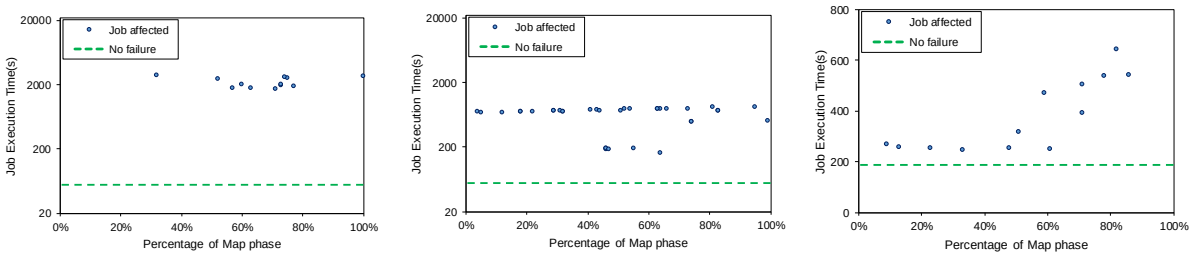
2.2.2 Prospective only

Another critical issue of current speculation relates to the correlation between map and reduce phases in MapReduce model. The existing speculation is made only among the running tasks. If a task is finished, it will be excluded from the candidates for speculating. The progress comparison of the existing speculation algorithm still use the completed task's progress (100% of course) to determine if other running tasks are to be speculated, but no longer consider speculating the completed tasks themselves.

Intuitively, it is a reasonable strategy since completed tasks should have no way of straggling a job. However, MapReduce computing typically requires the use of intermediate data that is produced by the completed MapTasks. Clearly, it will be a problem if those intermediate data are lost because 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. The retrospective speculation implies that a task should be considered to be subject to failure even if its progress has reached 100%. If speculation fails to consider those completed tasks, they will face a speculation breakdown along with serious degradation of job performance, as we will see in later section.

2.3 The breakdown of the existing speculation

Next we demonstrate how the above issues can lead to the existing speculation mechanism's breakdown, causing disastrous performance degradation of MapReduce jobs. We take YARN as a



(a) 1GB jobs of original YARN. (b) 1GB jobs of modified YARN. (c) 10GB jobs of modified YARN.

Fig. 2: Running time of MapReduce jobs in presence of node failure on different spot.

study case and use Wordcount as the representative benchmark. During each job, we generate failure of the task-hosting node at different progress spot. But we avoid crashing the master node or AM-hosting node because those will fail the job entirely. Other experimental setup can be found in Section 4.

2.3.1 *Speculation breakdown in small jobs*

Fig. 2 shows the execution time of individual jobs. Each dot represents a job with node failure and its turnaround time, and the dotted baseline below indicates the normal job execution time without failures. Firstly, Fig. 2(a) shows the results of 1GB jobs that have node failure at different progresses of map phase, The result is as astonishing as it stands. Most of the jobs take time that is orders of magnitude longer than the failure-free case, but there is an unusual issue behind this. YARN needs to clean up the container after a task attempt is finished. 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 fixed number of retires, which is decided by the IPC connection configuration settings (*ipc.client.connect.**). In default, it will take about 30 minutes for connection retry. During this time, the job will not end successfully even both map and reduce's progresses may have already reached 100%.

But this long timeout for container cleanup is not the sole reason that hurts the job performance. Fig. 2(b) is the result of our control group 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 retry problem with the figures in Section 1 or Section 4). We can still observe large performance degradation compared with the normal job running time. The culprit here is the first issue of existing speculation, the intra-node only 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 timeout.

However, this delay of MapTask timeout still explains only a part of our test cases. We can see that if node failure occurs 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 becomes fast enough that can trigger the speculation of the slowest task. 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 the progress rates of those speculation copies are high enough, they will in turn trigger the speculations of MapTasks on the failed node, and let the job proceed normally thereafter. So, although the job is still slower than usual, the avoidance of long timeouts make their performance much better than other failure tests.

But if node failure occurs on even later phase of the overall map progress, the disadvantage of the second issue, the prospective-only speculation starts to appear. As many MapTasks are now completed, the ReduceTask that are trying to fetch those MOFs will have fetch failures since the MOFs are unavailable on the failed node. After the time for fetching one MapTask output exceeds the limit (determined by *mapreduce.reduce.shuffle.read.timeout*), 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 also can be declared failed and rescheduled. Additionally, 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, intra-node only speculation is also broken down during this period for not MapTasks but ReduceTasks. Recall that in MapReduce workflow, reduce phase does not require the completion of map phase, ReduceTask can start running when one wave of MapTasks is finished. So in the end half of the map phase, some ReduceTasks may have already been launched. 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 compare to. The entire job will halt until the ReduceTask gets a timeout (600 seconds in default, too). Thus, these cases (mostly during 50% to 100% of map phase) have similarly bad performance.

2.3.2 *Speculation breakdown in larger jobs*

What if the data size is larger? Now the effects of task convergence is eliminated, but the cost of node failure on the map phase is still significantly high. As we can see from Fig. 2(c), which is the results of the same test but with 10GB of input, the running time of most failure cases are nonetheless more than twice as much as the failure free case. 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 detects that it is slower than others. But we found that the majority of jobs still suffer various performance degradation. The causes are similar to the 1GB test but with slight variations:

- Intra-node speculation may be invalid for multiple ReduceTasks, too. We already show that the failure of only one ReduceTask will cost us 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 speculated, which is contingent on the YARN's speculation algorithm. In Fig. 2(c),

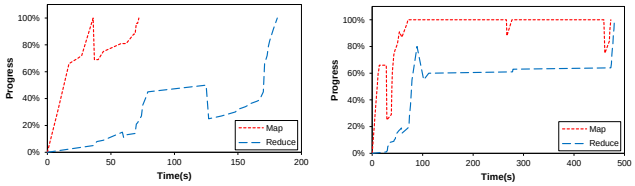
the jobs that have more than 600 seconds execution time are mostly due to this cause.

- The other test cases in Fig. 2(c) that spend less than 600 seconds but a lot more than no-failure case suffer from the 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.
- The fact that speculations are conducted intermittently is not effective. In Fig. 2(c), most of the jobs in early phase suffer from this cause. A node fails and all tasks on it wait one-by-one to be speculated. Depending on the speculation interval, the jobs have various delays on their completion time.

2.4 Complexity of Timeout Setting

We have already seen that the default timeouts are too long for MapReduce framework to detect failures and thus can prolong the speculation and failover process. One may assume that the problem can be easily solved by simply decreasing the timeouts. However, those long timeouts are necessary for it to adapt to heterogeneous environment since the networking situation is unknown and unstable. If the timeout is too small, tasks could be falsely declared failed when the network is just experiencing some temporal congestion. Fig. 3(a) shows a 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. about 80s, some ReduceTasks are shuffling), or even backslide if the delays exceed the timeout and the corresponding tasks are declared failed (at about 130s).

Thus, simply changing configurations is not feasible for unstable networks, let alone if there are more unstable factors such as failed nodes in the environment. Fig. 3(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, causing progress backslides. Then, 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 the 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 (290s and 480s). Until those missing MOFs are reproduced by the corresponding MapTask speculations, the reduce phase can continue and the job is completed quickly after that.



(a) Unstable network (b) Unstable network and node

Fig. 3: The progress of map and reduce of jobs with modified timeout.

3. DESIGN AND IMPLEMENTATION

In this section, we will unfold our designs and some important implementation features in order to tackle down the aforementioned issues of the 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 (FARM-Speculation, or FARMS). Fig. 4 shows the demonstration, where the existing speculation is shown on the left and FARM is shown on the right. Each small box represents a running task and its brightness indicates the task's progress (darker box indicates later phase of a task). In the existing speculation, a straggler is speculated upon periodical progress comparison and its copy will be attached to the task scheduling queue.

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. Actually, because the speculator only coordinates the tasks progress at task level, it does not need the node level status for the computation. Thus, a simple node failure can straggle the whole job. Our solution is straightforward. In FARMS, we leverage the failure information that is collected by a global monitor that runs with the ResourceManager.Tasks are associated with their host nodes, so the affected tasks can be speculated collectively and incrementally when a malfunctioning node is detected.

Secondly, in FARMS, we continue to list the completed tasks in the speculation candidates. We add transitions that can speculate the completed tasks that associated with a failed node. When the speculation task attempts have completed, ReduceTasks will be notified to fetch MOFs from the new task attempts instead of the original ones. But note that the speculation of completed tasks are not based solely upon successful detection of unresponsive node, the fetch failure of certain MOFs is also taken into consideration. The difference is on the granuarity of speculation. If a single fetch failure is notified by YARN, we speculate that particular completed task. To avoid unnecessary speculative copies, the node-based speculation and the task-based speculation are mutually excluded. When one task is speculated by either way, it will not be speculated again by another cause.

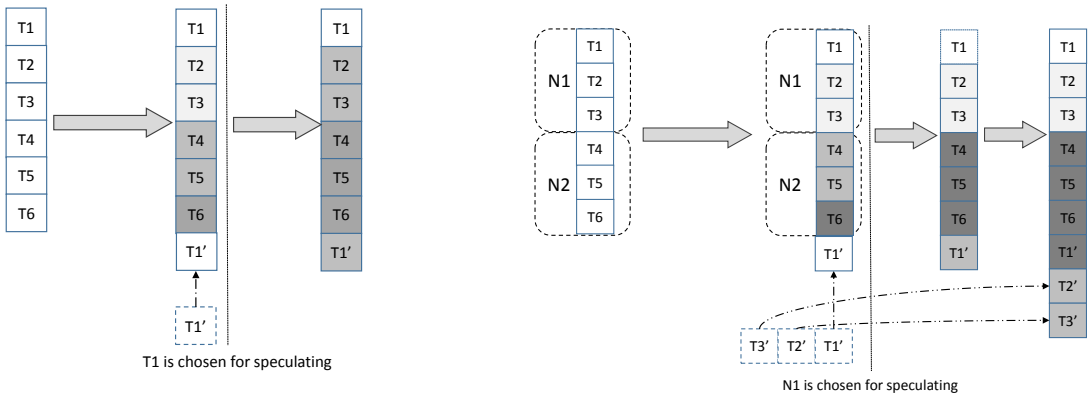
Thirdly, we change the single speculation to batch speculation, meaning that when we decide to launch additional speculations, they can be all launched at once. But note that such speculations can be costly sometimes because if we make false-negative decision on the node exceptions, there will be a lot of unnecessary additional resource consumption that comes with the speculation. Although we have optimized our decision algorithm (will introduce in Section. 3.2) but we still want to minimize the cost. Thus, we incorporate an multiplictive 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 Fast Analytics Scheduling

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 critically 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 tolerated.

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.

(i) The decision made in most cases should decrease the job ex-



(a) Existing speculation

(b) FARMS

Fig. 4: Comparison between existing speculation and FARMS.

ecution time.

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

(iii) Even when the decision misjudges the situation, the impact to job performance should be cheap.

(iv) It can be self-adjusted to different failure patterns.

To meet (i), we need to keep the rescheduling of tasks aggressive enough so it can gain at least some 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 (ii) and (iii), we need the threshold to be dynamically adjusted according to the specific cluster conditions. So we have to limit the number of positive 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 (iv). By doing so, it can have optimal performance under different failure scenarios.

Our overall algorithm is detailed 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 dead only after waiting for a relatively 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 of other nodes. This is necessary when the node failure is permanent because all the computations need to be re-conducted 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, we 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. Through 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. The latter one in some sense falls into the same timeout mechanism of default YARN, which is already shown 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 free from node failures. However, we do not recommend that because commercial cluster trace [3] shows that the problematic nodes are very common and they are typically distributed evenly across a cluster because usually the faulty nodes have already been blacklisted during some periodic checks before the job execution.

Algorithm 1 Enhanced Failure Recovery with FAS

```
1: N ← {The compute node}
2: T_N ← {Tasks running on N}
3: threshold ← {Time threshold to decide a node failed}
4: Fail_cur ← {Number of nodes currently already speculated}
5: Fail_max ← {Maximum number of node failures allowed}
6: P_a = 1.5 {Heuristic parameter used to increase the threshold}
7: P_b = 0.5 {Heuristic parameter used to decrease the threshold}
8: if N's lost time > threshold then
9:   if Fail_cur ≤ Fail_max then
10:    Fail_spec = Fail_spec + 1
11:    for all task ∈ T_N do
12:      schedule new attempt of task on other node.
13:    end for
14:    continue to monitor the node status...
15:    if N resumes responsive then
16:      t = time length of the node's loss of connection
17:      threshold = (average of t of the last 5 runs) ×P_a
18:    else
19:      threshold = threshold × P_b
20:    end if
21:  end if
22: end if
    {When the job is done, document the longest connection lost time t of
    N during the job.}
```

4. EXPERIMENT EVALUATION

4.1 Experiment Environment

Hardware Environment: All experiments are finished on a cluster of 21 server nodes that are connected through 1 Gigabit Ether-

net. Each machine is equipped with four 2.67 GHZ hex-core Intel Xeon X5650 CPUs, 24GB memory and one 500GB hard disk.

Software Environment: 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.

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

4.2 FARMS Evaluation

We have examined the FARMS against node failures to see if it can tackle down their negative impacts. Since node failure has very different impacts on the job execution (Section 2) due to varying job size, we have conducted two sets of experiments on jobs that have small or relatively larger input size. For small size jobs, we ran the three benchmarks with 1GB of input and we crash a node that hosts the MapTasks at a different progress spot during the overall map phase. For larger size jobs, the input size is 10GB and the node to crash is picked randomly.

Fig. 5 and Fig. 6 show the performance comparison between the original YARN and ours using FARMS against node failure at different phases. At each spot, we test it at least three times and get the average. 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. FARMS speeds up the job execution time by almost an order of magnitude. It manages to keep the job completion time to be comparable with the no failure case. For larger jobs, it can also tackle down the failure delay significantly. Moreover, the original YARN has very distinct performance due to different failure occurrence spot and benchmark type. But FARMS smooths out the variation and provide constancy and predictability for job executor.

4.3 Overall Evaluation

The evaluation of FARMS demonstrates the advantage of FARMS in handling node failures. But we also want to know how FAS can help FARMS fit in the real-world environment. Thus, we run different size of Terasort, Wordcount and Secondarysort jobs in sequence. We referenced [4] to set the size of jobs, as shown by Table 1. We let the job arrives at random times following a Poisson distribution.

Table 1: Ratio of test group in data size.

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

We then generate task failure, node crash and network delays, each with a frequency as introduced in previous sections. We conduct tests with the exact same setup (job group, failure injection method and interval) for both original YARN and ours. Fig 7 shows the results of the overall evaluation. We can see that combining FARMS and FAS provides performance that is almost comparable with the no-failure YARN. For smaller jobs that are basically intact from node failures, all three cases are similar, but ours slightly outperforms original YARN with failures and, surprisingly, is even slightly better than original YARN without failures. This shows how the aggressive speculation can benefit small jobs. For larger jobs that are more often affected by node failures, original YARN performs a lot worse under failure but ours manages to keep their performance comparable to the no failure case. This shows that although not too much improvement can be gained for large jobs, the FARMS+FAS implementation would not hurt their performance. In overall, our performance is 15.3% better than the original YARN under our experimental setup.

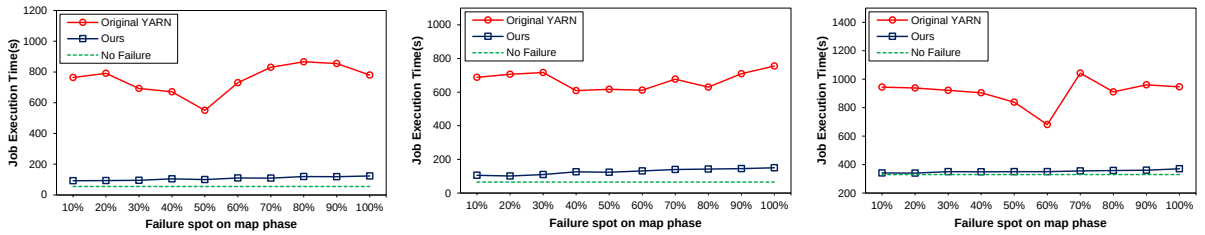
5. RELATED WORK

Speculation mechanism was introduced with the inital versions of many of the representative parallel computing paradigms such as MapReduce [11] and Dryad [17]. Since then, it has been extensively studied with a variety of viewpoints [28, 6, 5, 3]. But we find that none of these works has addressed the issues of speculation in handling with failures as discussed in this paper.

To name a few, LATE [28] scheduler takes node heterogeneity into account. It deliberately places the speculative copies onto fast nodes but not slow ones. But the question about when to make a speculation on failure-related stragglers remains unsolved. Also, its intermittent speculative strategy can cause significant amount of performance loss upon node failure because the job only proceeds till all speculative tasks are completed. Mantri [6] searches for the causes of stragglers and build its optimized speculation algorithm based on the straggler categories. It identifies in part the impact of failure-related stragglers. However, it only considers recomputation as the worst outcome incurred by failure, but does not address the delayed execution of speculation upon failure. GRASS [5] improves speculation's performance of the error-bound and deadline-bound approximation jobs by using two distinct scheduling strategies, a.k.a. Greedy Speculative and Resource Aware Speculative scheduling. But neither of the two strategies can serve the purpose to failure cases.

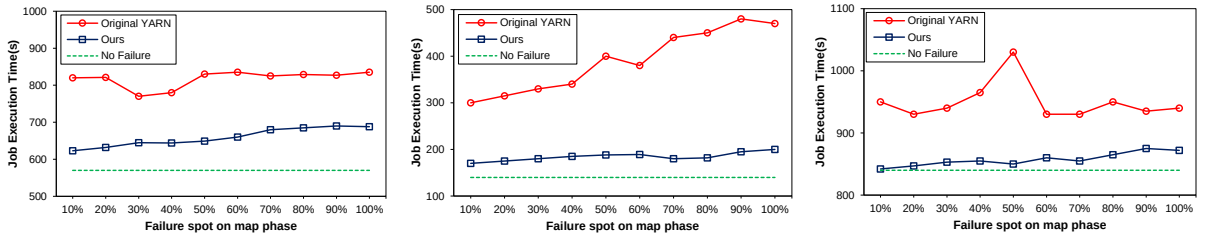
Among the studies of speculation, DOLLY [3] has similar research focus compared with our work. It digs into the straggler problem of small size jobs of MapReduce framework. They demonstrates that to aggressively launch a clone for every task is a good way to ameliorate the performance degradation that stragglers may impose on the MapReduce applications. Although their design can also be helpful for solving the performance breakdown of node failure found in this paper, it has an 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, 22, 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 with failures respectively, relying on such aggressive speculations for fault recovery is unpractical.

Besides speculation, our work has also set foot in the issue of MapReduce's fault tolerance. The existing efforts of this area include to analyze code bugs to prevent failure occurence [16, 27, 15], localize the failure timely and accurately [19], enhance data placement to achieve higher data availability [8], etc. Although the failure resiliency has gained so much attentions, we must be clear that strong failure resiliency does not imply optimal job performance. Failures can cost significant degradation of job turnaround time even if the job can eventually complete successfully, as shown in this paper. There are studies like [20, 13, 12, 26], along with our work, have revealed that due to the fact that failures are norm rather than exception in the real-world production deployment, to recover speedily from failures can be also essential. Similar to our work, Piranha [14] also recognizes the delays of small jobs in Hadoop framework but it focuses more on scheduling optimization.



(a) Terasort (b) Wordcount (c) Secondarysort

Fig. 5: Failure recovery of 1GB job.



(a) Terasort (b) Wordcount (c) Secondarysort

Fig. 6: Failure recovery of 10GB job.

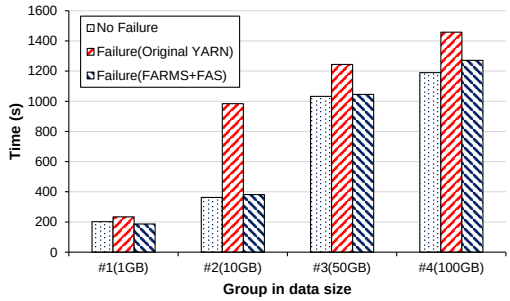


Fig. 7: Overall performance improvement.

Quiane-Ruiz *et al.* in [20] introduces RAFTing MapReduce for preserving the computation status of MapTasks and replicating the MOFs to reduce side. This design avoids the recomputation of MapTasks on the failed node, but it requires pre-assignment of ReduceTasks and additional network overheads. Moreover, it addresses only one negative factor of node failure which is the loss of MOFs, without taking care of failures happened on early map phase and looking into the means of failure's detection. Thus, it would still suffer from the performance degradation discussed in this paper and have problem dealing with real-world failure scenarios. However, we do think that the idea to conserve MapTask output may be beneficial to speculation as well to avoid unnecessary recomputations.

Dinu *et al.* in [13] conducts a comprehensive study on the impacts of node failure in MapReduce model. They revealed that a single node failure can significantly downgrade the performance of MapReduce applications. Specifically, they found that the failure of the node containing ReduceTasks can infect other healthy tasks and nodes, causing drastic performance degradation. Our previous work [26] has revealed issues similar to them, which we referred to as "failure amplification", and more importantly, also provided techniques to address the issues. But both works did not look into the failures occurring on map phase. Dinu's subsequent work RCMP [12] studies on how to conduct recomputation upon failures at the job-level. Our paper is orthogonal to those works by addressing map phase failures and leverage an optimized speculation mechanism to expedite the job performance at the task-level.

6. CONCLUSION AND FUTURE WORK

This paper details issues of the existing speculation mechanism that has long been neglected in the representative implementation of MapReduce model, i.e., YARN. It has revealed that existing speculation has flaws for failure recovery of small size jobs that have led to serious job execution delay. It demonstrates an extensive study about how the issues can cause breakdown of the existing speculation in presence of failures. Based on the findings, a new speculation mechanism called FARMS is proposed and a refined failure scheduling policy to leverage FARMS is designed. The results of a comprehensive evaluation show that our framework has dramatic performance improvement in handling with task/node failures than the original YARN and can adapt to an unstable environment very well. In the future, we plan to further explore the inefficiency of speculation, especially during reduce phase. We also plan to incorporate proper work-conserving mechanism for the speculations.

Acknowledgments

We are thankful to the anonymous reviewers for their insightful comments. This work is funded in part by National Science Foundation awards 1561041 and 1564647.

7. REFERENCES

[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] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: a cost-effective alternative to full cluster geo-replication. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pages 31–43. USENIX Association, 2015.

[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] F. Dinu and T. Ng. Rcmp: Enabling efficient recomputation based failure resilience for big data analytics. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 962–971. Ieee, 2014.

[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] K. Elmeleegy. Piranha: Optimizing short jobs in hadoop. *Proceedings of the VLDB Endowment*, 6(11):985–996, 2013.

[15] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, et al. What bugs live in the cloud?: A study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[16] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. Eio: Error handling is occasionally correct. In *FAST*, volume 8, pages 1–16, 2008.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[18] 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.

[19] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese. Gestalt: Fast, unified fault localization for networked systems. In *Proc. USENIX ATC*, 2014.

[20] 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.

[21] 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.

[22] 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.

[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] 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.

[27] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–265, 2014.

[28] 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.