FARMS: Efficient MapReduce Speculation for Failure Recovery in Short Jobs

Huansong Fu\textsuperscript{a}, Haiquan Chen\textsuperscript{b}, Yue Zhu\textsuperscript{a}, Weikuan Yu\textsuperscript{a}

\textsuperscript{a}Florida State University  
\textsuperscript{b}Valdosta State University

Abstract

With the ever-increasing size of software and hardware components and the complexity of configurations, large-scale analytics systems face the challenge of frequent transient faults and permanent failures. As an indispensable part of big data analytics, MapReduce is equipped with a speculation mechanism to cope with run-time stragglers 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.

We use the representative implementation of MapReduce, i.e., YARN and its speculation mechanism as a case study to demonstrate that the speculation breakdown causes significant performance degradation among MapReduce jobs, especially those with shorter turnaround time. As our experiments show, a single node failure can cause a job slowdown by up to 9.2 times. In order to address the speculation breakdown, we introduce a failure-aware speculation scheme and a refined task 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 node failures compared to the original YARN.

Keywords: MapReduce, YARN, Speculation, Failure Recovery

1. Introduction

MapReduce-based distributed computing has gained wide popularity since Google introduced it in 2004 [13]. 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 encountered in the first generation of Hadoop.

The popularity of Hadoop is largely due to its good performance for big data analytics workloads [13]. In order to achieve that in the highly unstable heterogeneous environment, where many task stragglers are common due to various reasons [36, 6], a mechanism called speculation is designed to contribute to the purpose. A global speculator proactively makes a copy of a straggler task that may block the job progress. The completion of either the straggler task or the new copy 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 speculatively duplicated, the job performance will not degrade too much.

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, i.e., a crashed or unresponsive node. It shows results 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.](image)

How serious is this impact? Although Hadoop is known for its ability for processing big data, a significant portion of jobs used in the real-world environment are actually small size jobs, which has been reported by a wide range of studies [9, 4, 3, 7, 26]. The size of MapReduce jobs in production clusters follows the power-law distribution with a vast majority of them containing less than 10 GB of input. For example, the distribution of Facebook workloads [4] demonstrates a heavy tail tendency, where about 90% of jobs have 100 or less tasks, and many with 100 GB or less input data. Thus, a lot of jobs will suffer from the performance degradation as shown above. In our experiments we only inject node failures, which are very common in real-world environment. According to [12], there is an average of five node failures during one MapReduce job’s...
execution, which was using 268 nodes in average. All these evidences indicate a critical need to revisit the existing speculation mechanism in the MapReduce model.

In order to address the aforementioned problem, we have studied the inability of the existing speculation mechanism in handling failure recovery. Then, we introduce a set of techniques, which we name it FARMS (Failure-Aware, Retrospective and Multiplicative Speculation). It includes an optimized speculation mechanism and a fast scheduling policy on failures. Our experimental results show that FARMS has dramatic performance improvement in handling failures compared to the original YARN.

In summary, our work makes the following contributions:

- We systematically reveal the drawbacks of the current speculation mechanism and analyze the cause and effects of node failures.
- We improve the efficiency of YARN’s existing speculation against failure by introducing our speculation scheme FARMS.
- 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 our new speculation mechanism 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 a representative implementation of MapReduce, Hadoop strives to provide outstanding performance in terms of job turnaround time, scalability, fault tolerance, etc [33]. YARN aims to overcome shortcomings of Hadoop and provide lower level support for various programming models, e.g., MapReduce and MPI [29] etc. For simplicity, we refer to YARN MapReduce as YARN in this paper. In 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 the reduce function on them. The final results are stored to the HDFS.

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 so that a correct failover is still ensured. In addition, YARN depends on long timeouts to declare a failure for every task. Such long timeouts are necessary to avoid false positive decisions on failure, but 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 reported by [12, 23, 27, 30, 8, 31]). As a result, YARN’s performance can be seriously affected by failures if it relies solely on the naive task-restarting mechanism. Thus, apart from the fail-over scheme, YARN also has a speculation mechanism which can help accelerate the detection and recovery.

Speculation has been extensively studied with a variety of focuses [36, 6, 3, 5]. Most of these strategies have a core similarity, i.e., they make a speculative copy for the slowest task during the job execution, a.k.a the straggler. For instance, the LATE schedulers [36] that is the default speculation mechanism of Hadoop, estimates the completion time for every task and uses the results to rank those tasks. The task that is estimated to finish the furthest will have a speculative copy on a fast node. After a configurable time interval, the speculator will search again for slow tasks and launch speculative copies for them intermittently. This strategy, along with others with some variations in how to determine stragglers (e.g. using processed data instead of task progress in Mantri [6]), have been adopted by the mainstream 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 has some major drawbacks, which seriously impede its efficiency in the real-world environment, where failures are prevalent. Next, we describe the two issues existing in the speculation mechanism. In the following context of the paper, we use speculate to describe launching a speculative copy for a task.

2.2.1. Converged Task Stragglers

To start with, speculation is simply making 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 the node becomes unresponsive due to node crash or lost connection, the speculator will not speculate any of those tasks since they have relatively the same progress. The speculation algorithm of the speculator cannot decide which task is slower, so the whole job will halt until each of the tasks gets a timeout and then starts from scratch again. Clearly, those timeouts can be avoided by early speculation as soon as YARN recognizes that those tasks on the same node have stalled. However, this
is not feasible in the current YARN speculator because its specu-
lation decision is based only upon comparison of progress or
processed data of task across all participating nodes. Thus, it is
unable to discover the intra-node converged task stragglers.

One may argue that the converged task stragglers can rarely
occur for most MapReduce jobs because it works against the
distributed computing nature of MapReduce. However, we find
that this phenomenon is not rare but indeed extremely common
among small size jobs. The reason is a design feature of YARN
MapReduce. Although the MapReduce framework provides
data locality that can help distribute the tasks evenly across dif-
ferent nodes, in practical implementation such as YARN, its
scheduler does not follow the same principle strictly. With its
default scheduling policy (capacity scheduler), the scheduler re-
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 con-
vergence and downgrades the effectiveness of speculation.

2.2.2. Prospective only Speculation

Another critical issue of the existing speculation relates to
the correlation between the map and reduce phases in the MapRe-
duce model. The existing speculation mechanism only specu-
lates running tasks. If a task is finished, it will be excluded from
the candidates for speculation. The progress comparison of the
existing speculation algorithm still uses the completed task’s
progress (100% of course) to determine if other running tasks
are to be speculated, but no longer considers making specula-
tive copies for the completed tasks themselves.

Intuitively, it is a reasonable strategy since completed tasks
should have no way of delaying a job. However, MapReduce
computing typically requires the use of intermediate data (the
MOF) that is produced by the completed MapTasks. Clearly, it
will be a problem if that intermediate data is lost, because the
job will be held up until it finally finds out that the intermediate
data is permanently lost. The MOF partitions of one MapTask
are often consumed by multiple ReduceTasks and one failed
node often contains MOFs from multiple MapTasks. Thus, a
failed node can cause a large number of delayed ReduceTasks.
As a result, completed tasks can also become stragglers, which
the current speculation mechanism is unable to address. In
other words, the existing speculation mechanism can only make
prospective copies of tasks, but not retrospective ones. The is-
ue of prospective-only speculation implies that a task should
be considered to be subject to failure even if its progress has
reached 100%. This issue results in serious degradation of job
performance, which we will demonstrate in later sections.

2.3. The breakdown of the existing speculation

With the ever-increasing size of software and hardware com-
ponents and the complexity of configurations, large-scale ana-
alytics systems face the challenge of frequent transient faults and
permanent failures. Conventionally, such faults and failures are
categorized based on the tolerance level an application may ex-
hit against them. Soft errors are transient faults upon which a
process may experience slow performance or erroneous results.
Hard errors are usually permanent failures caused by network
disconnection, disk and node failures. Both soft and hard er-
ors can cause the slowdown of all tasks on a single node. In
this section, we use hard errors as a study case to demonstrate
the impacts of speculation myopia. We test the default YARN
and use the Wordcount benchmark as an example. During each
job, we inject a failure of a task-hosting node at different stages
of map progress. We avoid crashing the master node or AM-
hosting node because that will fail the job entirely. Detailed
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 and its completion time. The dotted baseline in-
dicates the normal job execution time without failures. First of
all, Fig. 2(a) shows the results of 1GB jobs that have node fail-
ure at different progresses of their map phase. The results are
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 unrelated issue behind this. YARN needs to clean up the
container after a task attempt is finished. If the attempt is run-
ning on a failed node, YARN will keep trying to connect to the
corresponding NodeManager, which is currently unavailable,
and finally throws a timeout exception after a fixed number of
retries, which is decided by the IPC connection configuration
settings (ipc.client.connect.*). By default, it will take about 30
minutes to throw a NodeManager failure. During this time, the
job will not end successfully, even both map and reduce’s pro-
gresses may have already reached 100%.

This long timeout for the 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
rule out the issue by modifying YARN’s default retry policy and
then conduct the same set of tests (note that we have also ruled
out this issue with the experiment results shown in Section 1
and Section 4). In Fig. 2(b), we can still observe significant
performance degradation compared to the running time of nor-
mal job. The culprit here is the issue of converged task strag-
glers as we have discussed earlier. When the node containing
all the MapTasks has become unresponsive, the speculator will
not speculate any of those MapTasks but wait for 600 seconds
until they get timeout.

However, this delay of MapTask timeout still explains only a
portion of our test cases. We can see that, if node failure oc-
curs on 40% to 60% of the overall map progress, some jobs end
only slightly slower than the no-failure case. This is because as
map phase proceeds, different MapTasks’ progress rates can be
uneven, meaning that some MapTasks can be much faster than
others, finally becoming fast enough that the progress variation
can trigger the speculation of the slowest task. Thus, when a
node failure occurs during this time, MapTasks on the failed
node are stalled, but the speculative copies of some MapTasks
will continue on other nodes. As job proceeds even further,
when the progress rates of those speculative copies are large
enough, they will in turn trigger the speculation of other Map-
tasks on the failed node, and let the job proceed normally there-
after. Hence, although the job is still slower than usual, the
avoidance of long timeouts result in a much better performance
than other failure tests.

If the node failure occurs on even later phases of the overall
map progress, the disadvantage of the second issue we have dis-
cussed, i.e. the prospective-only speculation, becomes relevant.
As many MapTasks are now 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 time
for fetching a MapTask output exceeds the limit (determined
by mapreduce.reduce.shuffle.read.timeout), the MapTask will
be declared failed and a new task attempt will be scheduled.
But it has seriously stalled the overall job progress because Re-
duceTasks are idle during that time. To make things worse, if
the fetch failures experienced by a single ReduceTask exceeds
another hard limit, that ReduceTask will also be declared failed and
rescheduled. Additionally, if the corresponding MapTasks
have not been timely speculated, the rescheduled ReduceTask
can experience a second fetch failure and thus be scheduled for
a third time.

On the same time, converged task stragglers still appear in
this phase, although they are ReduceTasks but not MapTasks.
Recall that, in the MapReduce workflow, the reduce phase does
not require the completion of the map phase. ReduceTask can
start executing when one wave of MapTasks finish. So, in the
second half of the map phase, some ReduceTasks have already
been launched. If the job has only one ReduceTask (often the
case for small jobs) and the ReduceTask is on the failed node,
it will certainly not be speculated since it has no other Reduc-
task to compare to. The entire job will halt until the Reduc-
task gets a timeout (600 seconds by default, too). Thus, these
jobs (mostly during 50% to 100% of map phase) also have poor
performance.

2.3.2. Speculation breakdown in larger jobs

What if the data size is larger? Now, the effect of converged
tasks is eliminated, but the cost of node failure on the map phase
is still significantly high. Fig. 2(c) demonstrates the results of
the same test with 10GB of input. The execution time of most
jobs with failure are nonetheless more than twice as much as
the no-failure case. Note that, right now the number of Map-
tasks is large enough so that they are assigned evenly to differ-
ent nodes. Thus, the speculator can successfully speculate the
MapTasks that reside on the failed node as soon as it detects
that it is slower than others. However, the majority of jobs still
suffer various performance degradation. We analyze the causes
as follows.

- Converged task stragglers may occur in multiple Reduc-
tasks, too. We have shown that the failure of only one
ReduceTask will cost 600 seconds of the ReduceTask time-
out. If a crashed node contains multiple ReduceTasks, the
progress of remaining ReduceTasks may not be slow
enough for them to be speculated. In Fig. 2(c), the jobs
that have more than 600 seconds of execution time are
d mostly due to this cause.

- The other jobs in Fig. 2(c) that spend less than 600 sec-
onds but a lot more than the 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 com-
pleted tasks is still unbearable compared to normal job
execution time.

- The fact that speculative tasks are conducted intermit-
tently is not effective. In Fig. 2(c), those jobs encoun-
tering node failures in the early phase have speculation
working successfully. But the tasks on the failed node
have to wait in line to be speculated. Depending on the
number of affected tasks, those jobs have various delays
on their completion time.

2.4. Issue with Shorter Timeouts

We have shown that the default timeouts are too long for
MapReduce framework to detect failures and can prolong
the speculation and failure recovery process. A natural solution is
that we can solve the problem by simply decreasing the length
of timeouts. However, the long timeouts are necessary for MapRe-
duce to adapt to heterogeneous environment since the network-
ing condition is unknown and may be unstable. If the timeout
is too short, tasks could be falsely declared failed when the net-
work is just experiencing temporal congestion. To investigate
the feasibility of short timeouts for MapReduce jobs, we con-
duct experiments with modified timeout setting. First of all,
we changed YARN’s timeout for MapTask and ReduceTask to
be 5 seconds and run the jobs in an unstable network where a

![Fig. 2: Running time of MapReduce jobs in presence of node failure on different spot.](image)
A lot of networking delays, varying from 1 to 8 seconds, are injected randomly. Fig. 3(a) shows the results of this case. We can see that, the progresses of both map and reduce are seriously affected. They either stall at the network delays when they need network transferring (about 80s, some ReduceTasks are in shuffle phase), or even backslide if the delays exceed the timeout and the corresponding tasks are declared failed (at about 130s). Furthermore, we then inject a failed node into the scene and the result is shown in Fig. 3(b). Many MapTasks (at about 30s) are quickly declared failed due to network delays, which cause progress backslides of the overall map progress. The progresses are further impeded by a node failure (at about 100s), after which one ReduceTask is declared failed immediately, but the overall reduce progress cannot proceed because the ReduceTask needs the MOFs on the failed node. So it keeps fetching those MOFs until a fetch failure is thrown. The ReduceTask continues to request other lost MOFs and undergoes two more fetch failures (290s and 480s). Only until those missing MOFs are reproduced by the corresponding speculative copies of those affected MapTasks, the reduce phase can continue and the job is completed quickly after that.

The above analysis shows how the short timeout can be affected by network jitters and node failures. We further examine the feasibility of short timeout in dealing with failure recovery using more choices of timeout length. As discussed in the previous section, failures happening at different progresses have distinct impacts on the job. Thus, we inject the failures at 0%, 50% and 100% of the map progress to the jobs. Fig. 4 shows the average job execution times when tuning different lengths of task timeout. Note that we do not consider the ReduceTask failures here so the results have precluded the jobs where one or more ReduceTasks are being hosted on the failed node. The figure shows that the shorter timeouts can only help to reduce the performance degradation of failure happening at 0% of map progress, where the speculation fails to work because of task convergence. It has limited benefits for failures at 50% because it cannot resolve the performance degradation caused by the prospective-only speculation. Additionally, it has nearly no effect on helping the failures at 100% of map progress. Thus, we conclude that although shorter timeouts can avoid the performance degradation of early failures, it cannot help with later failures and more importantly, it comes with the price that the MapReduce framework would be much less effective to defend against transient faults which are even more frequent than node failures. We cannot simply rely on timeout tuning to solve the problem. Instead, we have to address the internal limitations of the existing speculation mechanism.

2.5. Proposed Solution

To restore the broken down speculation and accelerate MapReduce failure recovery, we propose a hybrid solution, including a run-time failure analyzer, a new speculation algorithm and a new scheduling policy. The failure analyzer, being initiated as a YARN component, will detect failure occurrence as early as possible, supplying run-time failure awareness for the YARN speculator. The central design is the new speculation scheme named FARM, which takes advantage of the failure analytics results, will bundle all affected tasks and speculate them in a collective manner. The new scheduling policy will incorporate the speculation algorithm, providing fast recovery from node failures, while restraining additional overheads incurred by the speculation in minimal using an accurate detection method of node failures.

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 with failure recovery.
3.1. Failure Awareness of YARN Speculator

As discussed before, the breakdown of the existing speculation roots in its unawareness of failures. We need the YARN speculator to be aware of the failures so it can facilitate failure recovery by launching early speculative copies. YARN has an application history service and configurations like yarn.api.records that can act like an information bank for post-execution analysis. But they do not feature a service that is dedicated to run-time failure analysis, nor any efficient associations between the speculator and failure analysis. YARN needs a standalone server whose responsibility is to gather only the information of system exception/failure and guide the failover. To this end, we have designed and implemented a Centralized Fault Analyzer (CFA) that can collect and monitor system anomalies at run-time, and provide failure analysis to YARN’s speculator for the speculation decision.

Our framework is shown in Fig. 5. The CFA is initialized with ResourceManager as a CompositeService. CFA keeps two levels of records: job-level and system-level. At job-level, CFA keeps a record of the job logs such as job ID, task IDs, container assignments, etc. At system-level, CFA keeps track of the node health status, such as the time duration of a node connection loss. Since CFA and ResourceManager are collocated on the same master node, information about the running application can be retrieved from the ResourceManager without going through the network. System-level logs are recorded by individual component who discovers it. CFA gathers those logs and provides its analytics result back to the speculator, which resides on a slave node, along with the AppMaster. The speculator will consume the results and adjust the speculation accordingly. Because all logs are general information gathered in every several seconds (five seconds in our implementation), the size of the logs is trivial. We will demonstrate in Section 4 that CFA’s extra I/O is lightweight and it incurs minimal overheads.

Failure to CFA itself can be a tricky issue. However, our design of CFA guarantees its availability upon failure. Because 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.

3.2. FARMS

We design a new speculation mechanism that is Failure-Aware, Retrospective and Multiplicative Speculation (FARM-Speculation, or FARMS). Fig. 6 shows the demonstration of FARMS, where the existing speculation is shown on the left and FARMS 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 speculative copy will be attached to the task scheduling queue. FARMS improves such traditional design in the following ways.

First of all, 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 computation. Thus, a simple node failure can delay the whole job. Our solution is straightforward. In FARMS, we leverage the failure information that is collected by CFA. Thus, FARMS knows the association between tasks and their host nodes. When a failed node is detected, FARMS can list all the affected tasks as its speculation candidates.

Secondly, in FARMS, we continue to list the completed tasks in the speculation candidates. We add transitions that can speculate the completed tasks that are 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. But note that, the speculation of completed tasks are not based solely upon the successful detection of unresponsive node. The fetch failure of certain MOFs is also taken into consideration, though there is difference in the granularity of speculation. If a single fetch failure is notified by YARN, we speculate that particular completed task. To avoid unnecessary speculative copies, the speculation triggered by unresponsive node and the speculation triggered by intermediate file fetch failure are mutually excluded. When one task is speculated by either way, it will not be speculated again by another cause.

Thirdly, FARMS speculates stragglers in a collective manner, meaning that when we decide to launch speculation upon the detection of a node failure, all the affected tasks can be speculated at once but not one by one. But we are aware of that such speculation can be costly sometimes because if we make false-negative decision in the node’s unresponsiveness, there can be a lot of unnecessary speculative tasks, along with additional resource consumption that is also unnecessary. Although we have optimized our decision algorithm (will be introduced in Section 4.4) but we still want to minimize the cost. Thus, we incorporate a multiplicative speculation mechanism into FARMS. Upon the detection of an unresponsive node, the number of tasks to speculate increases in exponential order. The condition to keep making speculation copies is contingent upon 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.

3.3. Fast Analytics Scheduling

Finally, we propose a new scheduling procedure on failures based on FARMS. We name it as the Fast Analytics Scheduling.
ing (FAS). As mentioned before, there is an important trade-off between speeding up the failure detection and keeping low resource consumption. In FAS, we use a dynamic threshold to determine if a failure should be speculated or not. If the node has been unresponsive for a time duration longer than the threshold, FAS deems it as a positive result. Otherwise, it is a negative result. The positive result indicates a failed node and tasks on that node will be added to the list of speculation candidates. The negative result will be tolerated and we will wait to see its further responsiveness.

Before we go into details, the design choice of the scheduling policy need to be sorted out. In general, the algorithm should meet the following principal requirements.

(i) The failure detection made in most cases should decrease the job execution time.

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

(iii) Even when the detection is wrong, its impact to job performance should be trivial.

(iv) The algorithm should fit into the real-world environment.

To meet (i), we need to keep our speculation aggressive enough for gaining some performance improvement. That means, the threshold cannot be too large. Otherwise, a big threshold for node failure would cause similar issue with the long task timeouts, which as we have demonstrated before, can greatly hurt the performance. However, to meet (ii), we need the threshold to be dynamically adjusted according to the specific conditions of the running environment. Thus, we have to limit the number of positive results and so does the number of speculative tasks imposed to the job execution. Then, to meet (iii), our speculation needs to be cautious in spawning speculative tasks. This is accomplished by the multiplicative speculation mechanism, as discussed before. Finally, we need to tune the key parameters in our algorithm to meet the requirement (iv). By doing so, it can have optimal performance when we deploy it with real-world MapReduce systems. Those principles indicate us that, to balance the trade-off between effectiveness and efficiency of speculation, the threshold to determine an unresponsive node is crucially important. For that matter, we introduce a Temporal Window Smoothing algorithm, which assigns various weights to the recent several responsiveness of a node. This algorithm is detailed in the following section.

3.3.1. Temporal Window Smoothing

To decide the threshold that is used for the FAS procedure (Fig. 7), we need to consider the recent unresponsiveness of the node. A node is down if it has lost disconnection permanently. But as discussed in previous section, we need to timely detect the anomaly to conduct proper speculation. A plausible approach is to calculate the average of some number of recent node disconnection duration. But this approach overlooks the unstable environment that the node may reside on. Network latency/throughput does not remain plateau, it can be slow for some time and become fast later. Thus, instead of using the average value [18], we adopt a Temporal Window Smoothing mechanism that can take the historical node unresponsiveness into account with varying weights, where the earlier unresponsiveness has less impact on the determination of the threshold.

To be more specific, in order to capture the temporal locality between the last \( L \) failures and the next failure at node \( i \), we define the length of our smoothing window as \( L \). We then use \( R_n \) to represent the real unresponsive time duration for the last failure at node \( i \) while \( P_{n+1} \) denotes the predicated unresponsive time duration for the next failure. Given any node \( i \) and a smoothing window with a length of \( L \), \( P_{n+1} \) can be estimated as follows:

\[
P_{n+1} = \frac{\sum_{k=1}^{L-1} (2^{L+1-k} \times R_{n+1-k})}{\sum_{k=1}^{L-1} (2^k)}
\]  

(1)

To give an example, if we set \( L \) to five, given the last five unreponsiveness of a node \( i \), denoted as \( R_{n-4}, R_{n-3}, R_{n-2}, R_{n-1}, R_n \), the prediction is:

\[
P_{n+1} = \frac{(2^4 \times R_{n+1-4}) + (2^3 \times R_{n+1-3}) + (2^2 \times R_{n+1-2}) + (2^1 \times R_{n+1-1}) + (2^0 \times R_{n+1})}{16 + 8 + 4 + 2 + 1} = \frac{16R_{n+1}}{31}
\]

which can be simplified to:

\[
P_{n+1} = \frac{16}{31}R_{n+1}
\]
$R_n$, the threshold should be set as follows:

$$P_{n+1} = \frac{2^1 \times R_{n-4} + 2^2 \times R_{n-3} + 2^3 \times R_{n-2} + 2^4 \times R_{n-1} + 2^5 \times R_n}{2^1 + 2^2 + 2^3 + 2^4 + 2^5}$$  \hspace{1cm} (2)

The parameter $L$ can be tuned based on the trade-off between the prediction accuracy and the computing overhead, i.e., the extent to which the historical behaviors will affect the future responsiveness. The base (in this case, it is 2) can also be tuned based on the characteristics of the nodes. A higher base should be set if the next responsiveness is more temporally related to the last few failures.

### 3.3.2. Overall FAS Procedure

The procedure of FAS is illustrated in Fig. 7. We take a simple heuristic method that guides the speculation of FARM to recover jobs from failures. As discussed before, we use the temporal window smoothing to deduce a failed node. If the positive result is correct (i.e., the node is failed), all the computations need to be re-conducted anyway so those early speculative tasks should not be considered as overheads. But in the case of a false positive result (i.e., the node is not failed), the speculative tasks may cause some overheads. However, those speculative tasks still act as the competitors of the original tasks on the temporarily unresponsive node. The two groups of tasks can compete for completion. If the original node is resumed too late, the speculative tasks can still help the job progresses faster. If the node resumed soon, we issue kill to the speculative tasks, and the threshold is adjusted accordingly. We have examined the overheads that come with false results as will be shown in Section 4.

![Fig. 7: Workflow of FAS.](image)

Intuitively, it is good to blacklist the node that experiences frequent failures so the majority of jobs will be free from node failures. However, this approach is not sufficient to mitigate the performance degradation caused by node failures. The current blacklisting mechanism is to rule out the problematic nodes through some periodic system checks when no jobs are running, so the next running job would not use the blacklisted nodes. Thus, blacklisting may reduce the total number of failures occurrences in the long term, but it cannot prevent failures from happening. Even if we blacklist nodes at run-time, i.e., rule out failed node so the following tasks would not be scheduled on that node, it does not help either because the node failure probably have already caused stragglers and the corresponding performance loss.

### 4. Experiment Evaluation

#### 4.1. Experiment Environment

**Hardware Environment:** Our experiments are conducted on two private clusters. The first one is 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. The second one also has 21 nodes. Each node is featuring with dual-socket, 10 Intel Xeon(R) cores and 64 GB memory. The nodes are connected through a 10 Gigabit Ethernet interconnect.

**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. The key parameters of the whole software stack are listed in Table 1, along with the tuned value. To minimize the data usage, we use 2 replicas which are the minimal to recover lost data of node failure. The minimal and maximum memory allocation (yarn.scheduler.minimum-allocation-mb and yarn.scheduler.maximum-allocation-mb) decides how many containers can be launched on one node. The more containers will have more tasks affected during a node failure.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapreduce.map.java.opts</td>
<td>1536 MB</td>
</tr>
<tr>
<td>mapreduce.reduce.java.opts</td>
<td>4096 MB</td>
</tr>
<tr>
<td>mapreduce.task.io.sort.factor</td>
<td>100</td>
</tr>
<tr>
<td>dfs.replication</td>
<td>2</td>
</tr>
<tr>
<td>dfs.block.size</td>
<td>128 MB</td>
</tr>
<tr>
<td>io.file.buffer.size</td>
<td>8 MB</td>
</tr>
<tr>
<td>yarn.nodemanager.vmem-pmem-ratio</td>
<td>2.1</td>
</tr>
<tr>
<td>yarn.schedulenodeallocation</td>
<td>1024 MB</td>
</tr>
<tr>
<td>yarn.schedulermodeallocation</td>
<td>6144 MB</td>
</tr>
</tbody>
</table>

**Benchmarks:** We have selected a wide range of representative MapReduce benchmarks from two MapReduce benchmark suites. The first one is the built-in benchmarks of YARN, including Terasort, WordCount, and SecondarySort. The other one is the well-known HiBench MapReduce benchmark suite v5.0 [21]. Developed by Intel, HiBench includes a wide range of MapReduce benchmarks with different emphasis of MapReduce characteristics such as map-heavy (K-means, Wordcount, Scan etc.) and reduce-heavy (Join, Terasort, Pagerank etc.) For other experiments that do not mention the benchmark type, we use the built-in Wordcount results (Section 4.2.2, Section 4.4, figures in Section 1 and 2).

**Performance metrics:** We measure the Job Execution Time for the performance evaluation of FARMS. And to evaluate the FAS algorithm, we use the job execution time plus Additional Tasks Rate, which is the number of speculative tasks that should not be launched, i.e., the original task attempts are not on a failed node. To inject a node failure in the test, we simply kill
all JAVA processes on that node. We compare FARMS against the original YARN, which adopts the LATE scheduler [36] as its default speculator. Throughout the results, we use YARN to indicate the results of original YARN and Ours to indicate our framework.

4.2. FARMS Evaluation

We examine the effectiveness of FARMS in tackling the speculation breakdown and performance degradation under node failures. Since node failure has very different impacts on the job execution (Section 2) with different job sizes and failure occurrence time, we conduct different sets of experiments with varying job sizes and timing of failure injection. The first set is small-size jobs that have 1GB of input data. The second set is median-size jobs that have 10GB of input data. The third set is very large size jobs that have 100GB and 1TB of input data. For small size jobs, we run different benchmarks and crash a node that hosts the MapTasks at 10 different spots during the job’s map phase. For median size jobs, we crash a random node and inject failure at various time spots. For even larger jobs, the performance difference between various time spots is less pronounced, so we only collect one of them to report.

Fig. 8 and Fig. 9 show the performance comparison between the original YARN and our framework against node failure happening at different map progress spots. At each spot, we test it five times and get the average. We show the highest and lowest execution time using error bars. Since the prolonged finishing delays caused by YARN’s retry policy is too large and can be tuned by simple re-configuring, in our experiment, we have precluded that 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 even manages to keep the job completion time to be comparable with the no failure case. For median size jobs, it can also tackle down the job delay significantly, although by a smaller factor. Moreover, the original YARN has very distinct performance at different failure spots because when the spots differ, the causes for delay also differ (Section 2). As shown by the error bars, FARMS smooths out the variation. Next, we will discuss more about the performance variation.

4.2.1. Performance Variation

As mentioned before, we plot the highest and lowest execution times with error bars in Fig. 8 and Fig. 9. As shown the figures, node failure can cause distinct impacts even at the same spot of failure occurrence. For small jobs, the variations are more obvious during the middle phase of the job than the initial or later phases. This is because as job approaches to halfway of its map progress, both types of speculation issues are possible to take effect as we have discussed in Section 2. Additionally, there is also a possibility that neither of them occurs, which is reflected by some results that have just marginally larger execution time than the no failure case. More types of causes of delays lead to larger performance variation. On the contrary, our optimization eliminates all the issues of the existing speculation in handling failures. The experimental results show insignificant variation in terms of the job execution times, which provides constancy and predictability for job executor in the real-world deployment.

For median size jobs, the performance penalty caused by failures is much less, but the variation is even more pronounced. Here, since there is no such issue of task convergence on one single node, the jobs are no longer suffering from the converged task stragglers and thus be able to avoid the long task timeout. This results in some test cases ending up having slightly worse performance compared to the no failure case. However, many jobs still suffer from the long waiting time for lost MOFs on the failed node, which has earlier appearance during the map phase than small jobs. This is because, for median size jobs, many MapTasks will finish a lot faster than others. Such imbalance in MapTask progresses means that a node failure is likely to cause MOFs lost at earlier map stage. Thus, the variation between highest and lowest execution remains large and, because the performance loss is smaller, appears to be more significant. In contrast, our optimization can always discover the failure and recover the lost MOFs within a fixed period of time and hence manage to keep the variation insignificant.

Table 2: Job execution time of 100 GB jobs (in seconds).

<table>
<thead>
<tr>
<th>Job</th>
<th>Terasort</th>
<th>Wordcount</th>
<th>Secondarysort</th>
</tr>
</thead>
<tbody>
<tr>
<td>No failure</td>
<td>597</td>
<td>318</td>
<td>1338</td>
</tr>
<tr>
<td>Failure-YARN</td>
<td>678</td>
<td>489</td>
<td>1445</td>
</tr>
<tr>
<td>Failure-Ours</td>
<td>617</td>
<td>355</td>
<td>1344</td>
</tr>
</tbody>
</table>

Table 3: Job execution time of 1 TB jobs (in seconds).

<table>
<thead>
<tr>
<th>Job</th>
<th>Terasort</th>
<th>Wordcount</th>
<th>Secondarysort</th>
</tr>
</thead>
<tbody>
<tr>
<td>No failure</td>
<td>5911</td>
<td>2175</td>
<td>7669</td>
</tr>
<tr>
<td>Failure-YARN</td>
<td>6078</td>
<td>2352</td>
<td>7949</td>
</tr>
<tr>
<td>Failure-Ours</td>
<td>5922</td>
<td>2199</td>
<td>7611</td>
</tr>
</tbody>
</table>

4.2.2. Performance of Even Larger Jobs

Besides gaining big performance improvement on small or median size jobs, we also want to evaluate our framework for very large jobs to see whether they still benefits from the optimization. To that purpose, we conduct the same tests but increase the sizes to 100 GB and 1 TB. We observe that for those jobs, the variation of performance at different failure spots is not as obvious as the small jobs. This is because in large jobs, failures at any spots of map phase almost always result in some MOFs loss. Thus, we only report one of the results for each case, which is the job having a failure at 50% of its map progress. The results are shown in Table 2 and 3. Among these experiments, for jobs that have relatively shorter execution time, e.g., the Wordcount jobs with both 100GB and 1TB of input, our optimization can greatly reduce the performance degradation caused by node failure and achieve performance that is comparable with the failure-free case. For other jobs that take much longer (e.g., Terasort and Secondarysort jobs with 1TB of input), the performance degradation is less significant.
and so is the improvement of our optimization. Moreover, the figure also clearly shows that our optimization does not hurt the performance of very large jobs. The additional speculative tasks launched by early speculation may cause some overheads during early phase. But as discussed before, those tasks are needed regardless of using optimization or not, while it is better to execute them earlier than later.

### 4.3. Evaluation with HiBench

We have deployed the HiBench benchmarks with its default configurations, e.g., the input size, number of data parallelism, etc. Note that in HiBench, each benchmarks is consist of multiple MapReduce jobs and has complex data dependency between the jobs. A random failure may cause undesirable outcome such as the loss of data that are needed by dependent jobs, which results in the failure of the entire benchmark test. Thus, we uniformly inject the failure at the 50% map progress of the last job in each benchmark. Fig. 10 shows the results of HiBench experiment. It shows that for all benchmarks, our framework is able to boost the performance in the failure case. Moreover, for map-heavy jobs such as Kmeans and Wordcount, the failures cause a lot more disastrous performance degradation than reduce-heavy jobs and our optimization can provide performance that is just slightly worse than the failure-free case. But even for the reduce-heavy jobs such as join and sort, our framework is still capable of quickly recovering from the failure and boost the job performance notably. Note that we have also tested our framework with HiBench in the failure-free scenarios. The results show that our framework have negligible additional overheads to the failure-free jobs. This shows that FARMS can handle the occurrence of failure without disturbing the normal job execution. The failure decision algorithm in FAS does not impose much additional overheads in a stable environment. In order to further evaluate the correctness and overheads of FAS in an unstable environment, we will present the corresponding evaluation of FAS in the next section.

#### 4.4. FAS Evaluation

To see if FAS can adapt to real-world environment where transient network congestion is common, we generate several experimental cases with a variety of settings. First, we test our framework against one single network delay or one failed node. We inject such network delay by delaying the packet with a time duration $t$. The time duration $t$ is generated according to the Poisson Distribution. We vary the average delay, denoted as $\lambda$, in the Poisson Distribution equation and generate a number of random delays. Then we conduct the same set of tests as the previous section but with those delays being injected. The node failure is injected at a certain rate, which is empirically set according to the real-world statistics from [14, 12]. From those reports, the average of node failure per job is 5 and the average number of nodes in one job is 268. In our experiment, we use 21
4.5. Overall Evaluation

The evaluation of FARMS demonstrates the advantage of FARMS in handling node failures and the evaluation of FAS demonstrate the effectiveness of our detection mechanism of node failure. We further examine the overall performance using FARMS+FAS for different benchmarks. We referenced [4] to set the size of jobs, as shown by Table 4.

Table 4: Ratio of test group in data size.

<table>
<thead>
<tr>
<th>Group</th>
<th>Size</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 GB</td>
<td>85%</td>
</tr>
<tr>
<td>2</td>
<td>10 GB</td>
<td>8%</td>
</tr>
<tr>
<td>3</td>
<td>50 GB</td>
<td>5%</td>
</tr>
<tr>
<td>4</td>
<td>100 GB</td>
<td>2%</td>
</tr>
</tbody>
</table>

Then, we inject various types of failures and exceptions, e.g., task failure, node crash and network delays, each with a frequency as introduced in the real-world experience of MapReduce system [13, 12, 14, 30]. We conduct tests with the exact same setup (job group, failure injection and interval) for both original YARN and ours. Fig. 12 shows the results of our 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 aggressiveness of FARMS can help small jobs for speeding up their turnaround times. For larger jobs that are more often affected by node failures, original YARN performs badly under failure but ours manages to keep its performance comparable to the no failure case. This shows that although we cannot gain much improvement for large jobs, the FARMS+FAS implementation would at least not hurt their performance.

5. Related Work

Speculation mechanism is introduced with the initial versions of many of the representative parallel computing paradigms such as MapReduce [13] and Dryad [22]. Since then, it has been actively studied with a variety of viewpoints [36, 6, 5, 3]. But
we find that none of these works has addressed the deficiency of speculation in handling with failures, as discussed in this paper. To name a few prior arts, LATE [36] 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 designs its optimized speculation algorithm based on the straggler categories. It identifies in part the impact of failure-related stragglers. However, it only considers data recomputation as the worst outcome caused by failure, without addressing the delayed execution of speculation upon failure. GRASS [5] improves speculation 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 the most similar research motivation but it has very different focus and approach compared with our work. It is motivated by straggler problem of small size jobs of MapReduce framework. However, unlike our paper which reveals the relation between failure and stragglers, its focus is more on general performance impact of stragglers. They demonstrate 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 could also be helpful for solving the performance degradation 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 [11, 28, 32]. In addition, those extra loads 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 speculation 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 occurrence [20, 35, 19], localize the failure timely and accurately [24], enhance data placement to achieve higher data availability [10], 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 [25, 16, 15, 34], along with our work, have reveal 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 [17] also recognizes the delays of small jobs in Hadoop framework but it focuses more on scheduling optimization.

Quiane-Ruiz et al. in [25] 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 to reduce side. 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 [16] conducts a comprehensive study on the impacts of node failure in MapReduce model. They reveal that a single node failure can significantly downgrade the performance of MapReduce applications. Specifically, they find that the failure of the node containing ReduceTasks can infect other healthy tasks and nodes, causing drastic performance degradation. Our previous work [34] has revealed issues similar to them, which we refer to as “failure amplification”, and more importantly, also provide techniques to address the issues. But both works do not look into the failures occurring on map phase. Dinu’s subsequent work RCMP [15] 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

In this paper, we have detailed issues of the existing speculation mechanism that has long been neglected in the representative implementation of MapReduce model, i.e., YARN. We have revealed that the existing speculation has fundamental flaws for failure recovery of shorter jobs that have led to serious job execution delay. We have demonstrated a comprehensive study on how those issues can cause breakdown of the existing speculation in presence of node failures. Based on the findings and implications, a new speculation mechanism called FARMS is proposed. We have also designed a refined scheduling policy to leverage FARMS. We have implemented the framework and evaluated it through an extensive set of experiments. The experimental results show that our framework has dramatic performance improvement in handling node failures than the original YARN and can adapt to an unstable running 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 speculation.

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.


