2015 IEEE 29th International Parallel and Distributed Processing Symposium

Cracking Down MapReduce Failure Amplification through Analytics Logging and Migration

Yandong Wang   Huansong Fu   Weikuan Yu

Dept. of Computer Science and Software Engineering
Auburn University, Auburn, AL 36849, USA
{*wangyd,hsfu,wkyu*}@*auburn.edu*

***Abstract*—MapReduce is popular for big data analytics because it offers easy-to-use map and reduce user interfaces while hiding the complexity of system scalability and fault resiliency issues. While a large body of literature has focused on improving the performance and scalability of MapReduce, the issue of fault resiliency has thus far received little attention. In this paper, we take on an effort to investigate the fault resiliency of MapReduce using YARN (the next-generation Hadoop) as a case study. We reveal that the failures of a MapTask, a ReduceTask or a compute node can cause distinctly different impact to MapReduce programs. Particularly, YARN MapReduce is not able to gracefully handle failures that involve ReduceTasks, causing prolonged task execution, delayed job completion, and, more severely, failure amplifications due to the cascading effects to other tasks. These problems together cause the performance collapse of MapReduce jobs. In this paper, we introduce a new fault-tolerant framework that can crack down failure amplification and gracefully handle failure scenarios. It is designed with two key fault handling techniques: *analytics logging* and *speculative fast migration*. Analytics logging is a light-weight mechanism that logs the key progress information of MapReduce tasks; speculative fast migration handles node failures by proactively re-executing MapTasks, migrating ReduceTasks, and collective merging with a pipeline of shuffle/merge and reduce stages. Our performance evaluation demonstrates that these techniques can eliminate failure amplification and deliver fast job execution compared to the existing task re-execution mechanism in MapReduce.**

I. INTRODUCTION

MapReduce [7] is a popular model for numerous organizations to process enormous amounts of data, perform massive computation, and extract critical knowledge for business intelligence. MapReduce provides easy-to-use map and reduce interfaces and hides the complexity of system scalability and fault resiliency issues from the users. Hadoop [2] is an open-source implementation of MapReduce. It has evolved into its next generation called YARN [1].

The capability of MapReduce in handling large amounts of data and critical tasks has also made it a research target for governmental, academic and industrial organizations. For example, many previous works have focused on improving the performance and scalability of MapReduce [7], [20], [6], [13], the scheduling of task and jobs for fairness and fast response time [23], [18], [22], and the redundancy and availability of data [15], [14].

Compared to the paramount interest on these topics, performance improvement, scheduling optimization, and data availability, the fault resiliency of MapReduce tasks has received much less attention. For nearly a decade, MapReduce implementations have been relying on the same fault handling mechanisms: speculative execution and task re-execution. However, the execution of MapReduce programs consists of two stages (map and reduce), each with a different type of tasks and distinct execution behaviors. We have examined the existing task re-execution mechanism in MapReduce and revealed that it does not work effectively for different task failures. In our experiment, we inject errors to fail a varying number of Map- or ReduceTasks and then measure the job's recovery time. Fig. 1 shows the impact of different task failures. YARN can quickly recover from a large number of MapTask failures, but it takes an order of magnitude longer time to recover from the failure of a single ReduceTask than the same from 200 MapTasks. To make things worse, from the trace reported by Kavulya *et al.* in [10], the average number of ReduceTasks per job is 19, with many containing more than 145. Along with the MapTask failures, TaskTimeout and other exceptions, they result in 3% of the total jobs ended as failed or cancelled, and many other jobs delayed. These facts suggest that it is necessary to revisit the fault handling mechanism in MapReduce. With so many concurrent inter-dependent activities happening in a large-scale MapReduce program, such a long-lasting effect of a ReduceTask failure may have inflicted other system components and caused other side-effects that need to be identified.

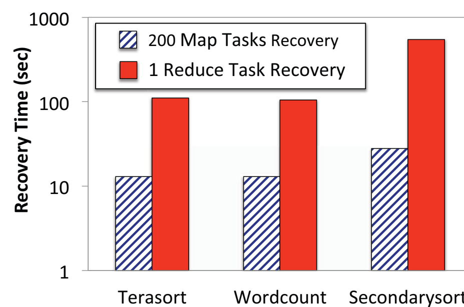


Fig. 1: The recovery time for a single ReduceTask failure and the failure of many MapTasks.

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.111

IEEE computer society

In view of this phenomenon, we take on an effort to investigate the fault resiliency of MapReduce using YARN as a case study. While an efficient resiliency mechanism is critical, YARN as an implementation of MapReduce relies on the conventional task re-execution to recover from various failure scenarios, with little regard to the need of differentiated recovery for distinct types of task failures. The failures of a MapTask, a ReduceTask or a compute node can cause distinctly different impact to a MapReduce program. More severely, because of the dependencies of ReduceTasks on the intermediate data from MapTasks, the failure of one ReduceTask can cascade into a series of failures of other ReduceTasks, a phenomenon we refer to as *failure amplification*. These problems together cause the performance collapse of MapReduce jobs.

To address these issues, we introduce a new fault resiliency framework that can crack down failure amplification and gracefully handle all failure scenarios. This framework is designed with two novel fault handling techniques: *analytics logging* and *speculative fast migration*. We have conducted an extensive set of tests to evaluate the effectiveness of our framework compared to the existing task re-execution mechanism in the original YARN. Our results demonstrates that the framework accomplishes efficient recovery performance under various failure scenarios. In summary, we make the following contributions on improving the fault resiliency of MapReduce for big data analytics.

- We examine the execution behaviors of MapReduce jobs under a variety of failure scenarios, and reveal several key shortcomings of the existing recovery mechanism such as prolonged task execution, delayed job execution and failure amplification.
- We have designed and implemented analytics logging as a viable lightweight approach for periodic preservation of data analytics of long-running ReduceTasks, and speculative fast migration as a proactive approach to re-executing MapTasks, migrating ReduceTasks, and pipelining the shuffle/merge and reduce stages.
- We have integrated analytics logging and speculative fast migration into one framework to crack down failure amplifications and ensure efficient failure recovery. Our evaluation demonstrates that these techniques can eliminate failure amplification and deliver fast job execution compared to the existing task re-execution mechanism in MapReduce.

The rest of the paper is organized as follows. Section II provides the background and motivation. We then describe analytics logging in Section III, followed by Section IV that details speculative fast migration. Section V provides experimental results. Section VI reviews related work. Finally, we conclude the paper in Section VII.

II. Background and Motivation

In this section, we start with a brief description of YARN and its MapReduce frameworks. Then we detail two issues that lead to poor failure recovery performance, including (1) delayed job execution, and (2) failure amplification.

A. Overview of YARN MapReduce Architecture

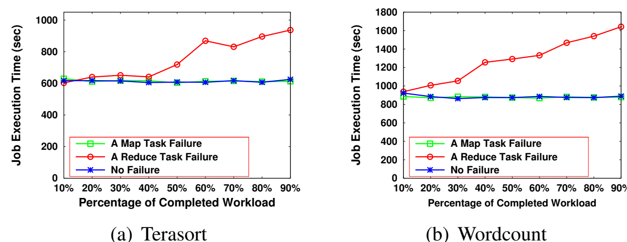
Designed as a resource management infrastructure, YARN aims to simultaneously support various programming models, such as MapReduce and MPI. It consists of two categories of components, including one ResourceManager and many NodeManagers. Each NodeManager abstracts the resources on the node as many *containers* that can serve the purposes of different applications. The ResourceManager manages all resources and allocates containers to running applications.

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

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

Fault resiliency: during the execution of a MapReduce program, current design relies on failover to provide fault tolerance. When the MRAppMaster detects the failure of running tasks, it re-launches those tasks on healthy nodes without distinguishing recovery tasks from normal tasks. The failed MapTasks can be successfully recovered as long as the input data is still available from HDFS. The successful recovery of ReduceTasks requires the presence or regeneration of MOFs produced by MapTasks.

B. The Issue of Delayed Job Execution



(a) Terasort (b) Wordcount

Fig. 2: Delayed execution of MapReduce programs.

MapReduce adopts a strategy via data regeneration to guard against transient faults such as out-of-memory, network congestion etc. Once a task is detected as failed, its running

status and past progress are discarded. The scheduler simply re-launches another attempt of the same task, repeating the work achieved previously. This works very well for short-lived MapTasks whose amount of work are generally much smaller compared to the ReduceTasks, but it is not as effective for ReduceTasks because of their long-running behaviors as documented by many MapReduce workload studies [23], [18]. We have conducted a set of experiments via running three representative MapReduce benchmarks, including *Terasort*, and *Wordcount* on a cluster of 21 nodes with SSDs as the storage. (Details about the testbed and system configuration are in section V-A). Fig. 2 shows the performance impact from a single task. We inject failures when a job reaches a varying percentage of progress, and compare its performance with that under failure-free scenarios. As shown in the figure, the failure of a MapTask has negligible impact on job completion, but a ReduceTask failure can degrade the execution time of Terasort and Wordcount by more than 43.2% and 50.3%, respectively. When more progresses have been made, a ReduceTask failure will cause even longer delays.

C. The Issue of Failure Amplification

YARN relies on running ReduceTasks to detect the lost MOFs from a crashed node. When a ReduceTask fails many times to fetch any MOF, the scheduler will take it as faulty and preempt it. Unfortunately, such design can cause many undesirable issues, amplifying the adverse impact of a single ReduceTask failure.

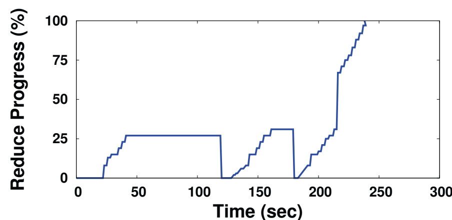


Fig. 3: The temporal repetition of a ReduceTask failure.

Temporal Failure Amplification: The simple task re-execution mechanism can lead to penalty amplifications upon the failure of a long-running ReduceTask. Fig. 3 illustrates the temporal amplification of such failures. We profile the progress of reduce phase of a Wordcount job (for illustration purpose, this job is configured with 1 ReduceTask). At 48th second, a node crashes and the progress stalls. Then, it takes a timeout (about 70 seconds) for the scheduler to detect the crashed node and start the recovering process at the 129th second. Unfortunately, the recovered ReduceTask tries to retrieve lost MOFs from the crashed node, encountering many data fetching failures and causing YARN to declare a second failure at the 180th second.

Spatial Failure Amplification: Also due to aforementioned connection issue, the failure of a single node that contains MOFs desired by running ReduceTasks can trigger cascading failures of healthy ReduceTasks. Fig. 4 shows the existence of such scenarios. We profile the reduce progress along with

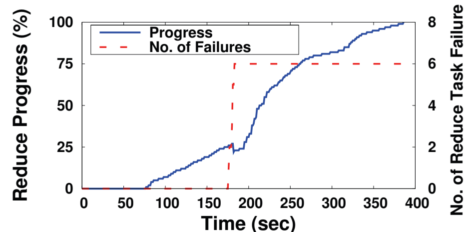


Fig. 4: A single node failure can infect healthy ReduceTasks.

the number of failed ReduceTasks during the execution of a Terasort job that has 20 ReduceTasks. A single node crash at the 176th second results in failures of six more healthy ReduceTasks running on other nodes, even though the crashed node does not host any ReduceTasks.

D. Proposed Solutions

In view of the aforementioned issues and our findings of distinct recovery behaviors of ReduceTasks, we plan to investigate the feasibility of logging the progress of MapReduce analytics for fast migration and recovery of ReduceTasks. Accordingly, we propose to design an Analytics Logging and Migration (ALM) framework to address the delayed job execution and failure amplification in MapReduce.

As shown in Fig. 5, our ALM framework is designed with two component techniques *Analytics LoGging* (ALG) and *Speculative Fast Migration* (SFM) for effective job recovery under various failure scenarios. ALG aims to conserve the progress of MapReduce tasks. It strives to do so in a non-intrusive, light-weight manner with minimized interruption or interference to the normal task execution. As shown in the figure, we focus on ReduceTasks (both shuffle/merge and reduce phases) for ALG in this paper. It will leverage both local file systems and HDFS for logging the progress of analytics work.

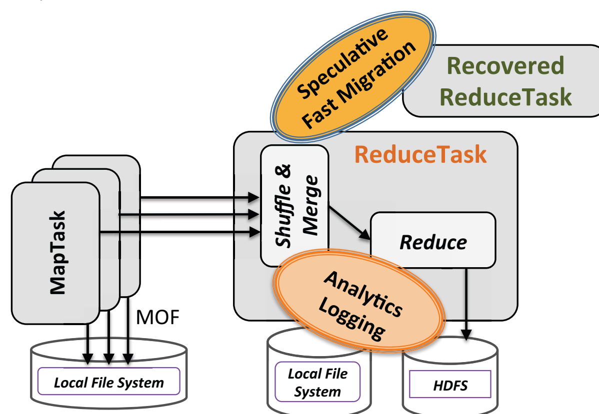


Fig. 5: Diagram of Analytics Logging and Migration.

In parallel with ALG, SFM leverages analytics logs generated by ALG and other resources (such as healthy containers) in the YARN cluster to orchestrate migration and collective recovery of failed ReduceTasks. It can help MapReduce programs to tolerate any failure and also crack down

its temporal and spatial amplifications. Furthermore, to fully exploit the capability of SFM, we have further enhanced the recovery scheduling policy in MRAppMaster based on a speculative mechanism. It speculatively re-executes MapTasks and generates map output files in advance, thereby preventing a ReduceTask from becoming a straggler when it is recovering from a previously failed ReduceTask.

In the next two sections, we will describe the ALG and SFM techniques in detail.

III. Analytics Logging

Though logging has been extensively studied for file systems [5], little research has been carried out to explore its feasibility for MapReduce tasks. Meanwhile, many system-level heavy-weight checkpointing mechanisms [9] that interrupt the execution of processes and take snapshots of the entire memory image can incur substantial overhead for tasks with several GBs of heap memory.

We introduce Analytics LogGing (ALG), which is a *non-intrusive* and *task-level* light-weight logging mechanism, running asynchronously alongside the execution of ReduceTasks. It aims to log only the key information that can help a recovering ReduceTask avoid conducting unnecessary reduce computation and data deserialization, and meanwhile minimize the interruption and interference to task execution. In addition, due to the independent characteristic of MapReduce tasks, ALG does not require any global coordination to maintain job-level consistency before logging. The analytics logs are completely managed within a task.

Due to distinct behaviors of different stages of ReduceTask, we need to have appropriate logging strategies for *shuffle/merge* and *reduce* stages within a ReduceTask.

A. Logging in the Shuffle/Merge Stage

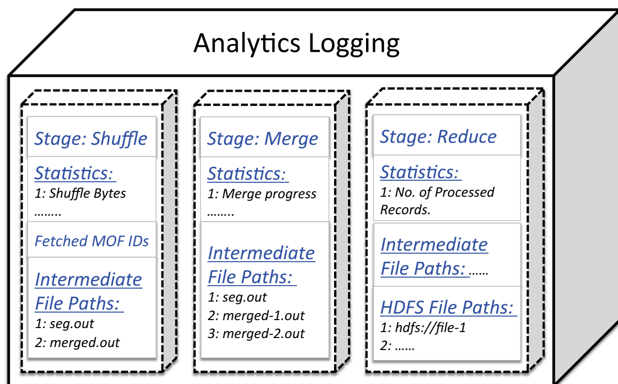


Fig. 6: Format of progress logs at different stages.

A ReduceTask periodically fetches segments from MOFs on remote nodes. Depending on the segment size and remaining available memory, it determines whether to store it in memory or spill to disks. Meanwhile, a couple of merging threads are running in the background to carry out in-memory and on-disk merging. Accordingly, to preserve the running status of this stage, the key analytics progress that needs to be logged includes the IDs of remote MOFs, and the paths of local intermediate files, such as on-disk segment files and intermediate merged outputs. Other statistics are also included to form an indistinguishable record for the resumed execution of a recovering ReduceTask. The format of a log record at the shuffle stage is shown by the left column in Fig. 6.

Once triggered, ALG invokes a temporary in-memory merging thread to merge all in-memory segments and flush merged results to local disks. Such merging is necessary to control the number of temporary files and amortize the merging overhead in the later execution of ReduceTasks. In addition, by launching a temporary thread instead of forcing the original in-memory merger to conduct above merge/flush, ALG minimizes the interruption to the on-going data shuffling that could be waiting for the completion of in-memory merging. Moreover, in order to accomplish the status logging, ALG needs to record the paths of persistent intermediate files processed by the on-disk merger. Waiting for the completion of on-disk merging can be very time-consuming and delay the logging process. Therefore, ALG pauses the on-disk merging to record such file paths. On-disk merging is resumed thereafter.

When a ReduceTask proceeds into the merge stage, all the segments have been successfully shuffled to the ReduceTask side. Thus, it is no longer necessary to maintain the IDs of MOFs. Therefore, the paths of intermediate files are the only critical information to keep. The updated format of a log record at the merge stage is shown by the middle column in Figure 6. When a large number of in-memory segments exist in the merge queue, it can incur a noticeable overhead to merge them. We have found that if we increase the logging frequency during the shuffle stage, very few in-memory segments will be maintained in the final merge queue, thus very little performance penalty for the creation of logs.

B. Logging in the Reduce Stage

In the reduce stage, all intermediate files are organized in a Minimum Priority Queue (MPQ). The file with the minimum <k',v'> pair is positioned at the root (head). Then ReduceTask sequentially extracts the <k',v'> pairs out of the queue, applies user-defined reduce function on it, and then writes the output into an HDFS file. Analytics logging in this stage requires ALG to record two things: the structure of MPQ and the positions of all intermediate files in the MPQ. This way, ALG can preserve the progress in the reduce stage.

To preserve the structure of MPQ, for every intermediate file, ALG logs the file path and the *offset* of the file for the next <k',v'> pair. The format of the log is shown by the right column in Fig. 6. Such information is adequate to recover the MPQ later. More importantly, we need to safely store the completed output from the reduce function. To this end, ALG flushes the HDFS result file asynchronously without stalling the execution of the ReduceTask. To curb the performance overhead and execution interference, the HDFS result file is generated with local and rack replicas. Furthermore, a ReduceTask in the reduce stage has completed a major portion of its work. Different from previous shuffle/merge stage, the

log record in the reduce stage is also stored into HDFS. This ensures the availability of the log from HDFS so that the recovered ReduceTask can avoid repeating data deserialization and the accomplished reduce analytics.

IV. Speculative Fast Migration

Fig. 7 provides an overview of speculative fast migration. For illustration purpose, we use an example of four compute nodes, each with one MapTask and one ReduceTask. During the job execution, ALG enables a ReduceTask to log the analytics progress, with the logs for the reduce stage stored onto HDFS. This is shown as Step 1 for the ReduceTask (R3) on the third compute node. When the tasks on the third compute node are not responsive, the YARN ApplicationMaster signifies a node failure (Step 2). The ApplicationMaster then speculatively prioritizes the migration of tasks from that node (Step 3), and recover them as M3' and R3' on other nodes. When the migrated ReduceTask (R3') is launched, it then leverages the logged records on HDFS to resume the previous progress of R3 (Step 4).

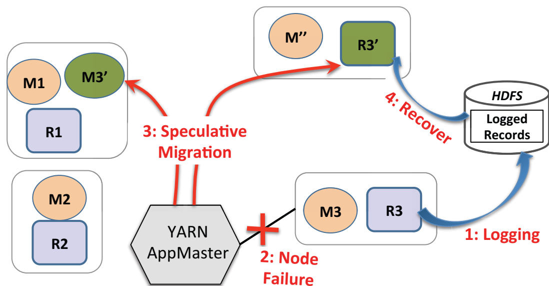


Fig. 7: The flow of Speculative Fast Migration

Leveraging the analytics logs: Since the progress of ReduceTasks has been logged periodically before a failure, we design SFM to take advantage of logged records. This avoids repeating redundant deserialization and the analytics work that has been logged. A recovering ReduceTask looks up the previously generated log files for one that records the progress in the reduce stage. If there is one, it then restores the MPQ and resumes the processing of remaining <k',v'> pairs in the MPQ. Our performance evaluation demonstrates that leveraging logs inside SFM can accelerate the recovery by up to 28.3% (see Fig. 15).

As discussed in Section II, the recovery of ReduceTasks may significantly delay the job execution. if not handled efficiently. We have designed Speculative Fast Migration (SFM) to accelerate the recovery of migrated ReduceTasks. It includes a *Fast Collective Merging* mechanism and a *Speculative Recovery Scheduling* policy.

A. Fast Collective Merging

Intermediate data merging has been widely recognized as a major bottleneck in the ReduceTask execution [17], [13], [12]. The conventional wisdom has focused on providing merging algorithms inside the final ReduceTask. Such design can be insufficient because the recovering ReduceTask, while late by nature, still needs to go through the heavy merging process.

We introduce Fast Collective Merging (FCM) to leverage all participating nodes in the job and have them work together to rapidly rescue the migrated ReduceTask. The key idea of FCM is to ask each node to merge local intermediate data before supplying them to the recovering ReduceTask. FCM distributes the merging process to all participating nodes and reduces the size of MPQ at the recovering ReduceTask.

When starting the recovery, FCM is designed to notify each participant node to organize its local segments into a local merge queue (named as Local-MPQ) and pre-merges these segments. Meanwhile, the recovering ReduceTask constructs another MPQ for global merging (Global-MPQ). Each segment in the Global-MPQ represents the output from a remote participating node. When all the required Local-MPQs and the Global-MPQ are established, FCM starts a pipeline of merging process from the Global-MPQ to all the Local-MPQs. In the pipeline, the recovering ReduceTask continuously draws the <k',v'> pairs from the Global-MPQ and reduce them. Meanwhile, all the participating nodes provide (or shuffle) their merged output to the ReduceTask, filling up the segments in the Global-MPQ. Throughout the FCM process, we ensure that all data be maintained in memory thereby overlapping the shuffling, merging, and reducing from all participating nodes to the recovering ReduceTask.

1) Failure Handling during Recovery: Failures can also occur during the recovery process. When a recovery ReduceTask fails again, we launch another attempt on a healthy node. FCM does not maintain any local intermediate data. It does not impose any constraint on the choice of the node for recovering a ReduceTask. However, we do need to tear down the structures organized for the previous recovery attempt. When the participant nodes in FCM receive no request from a recovering ReduceTask after a timeout period, they then dismantle their Local-MPQs.

B. Speculative Recovery Scheduling

To take advantage of FCM, we have enhanced the recovery scheduling policy inside MRAppMaster. We aim to achieve two objectives. First, we need to prevent a recovering ReduceTask from becoming a straggler and causing serious job delays. Second, we need to relieve a recovering ReduceTask of the burden of locating its dependent MOFs so that there are no stalls for such ReduceTask in waiting for MOFs, thus no cascading events and no failure amplification, as shown in Section II-C.

Our speculative recovery schedule algorithm is detailed in Algorithm 1. Upon detecting a ReduceTask failure, it determines the recovery mode based on the liveness of the node that hosted the failed ReduceTask (Lines 9 – 13). If the node is still operating, the analytics logs are then available on that node. Our scheduler then re-launches the same ReduceTask on the original node to resume from the logs and recover from the transient ReduceTask failure.

However, the cause of the ReduceTask failure may vary. It could be due to a faulty node or a failed node. In the former case, the node may still be responsive but very slow in I/O or computation. Task re-execution may end up with a straggler because of the slow recovery of ReduceTask. Thus it may not be effective to launch a ReduceTask on the original node.

Algorithm 1 Enhanced Failure Recovery Scheduling Policy

1: $R \leftarrow \{$A failure report$\}$
2: $T_{reduces} \leftarrow \{$Failed ReduceTasks in $R\}$
3: $T_{maps} \leftarrow \{$Failed MapTasks in R and lost MOFs involved in $R\}$
4: $N \leftarrow \{$The source node of $R\}$
5: **for all** $m \in T_{maps}$ **do**
6: schedule another attempt of m on a healthy node with higher priority.
7: **end for**
8: **for all** $r \in T_{reduces}$ **do**
9: **if** N is still alive **then**
10: **if** no. of attempts on N of r is $< limit_{local}$ **then**
11: schedule another attempt of r on N.
12: **end if**
13: **end if**
14: **if** no. of running attempts of r is ≤ 2 **then**
15: $t \leftarrow \{$a speculative task of $r\}$
16: **if** no. of FCM tasks in the job is $\leq FCM_{cap}$ **then**
17: schedule t with **FCM** mode.
18: **else**
19: schedule t with regular mode.
20: **end if**
21: **end if**
22: **end for**

Therefore, regardless whether a node fails or not, we spawn a *speculative ReduceTask* [24] on a healthy node (Lines 14 – 21) when there is a ReduceTask failure. This speculative task needs to be executed in the FCM mode (Lines 15 – 17) for fast recovery. To limit the resource demand and reduce the synchronization frequency caused by FCM, we cap the number of ReduceTasks in the FCM mode in each job (Line 16, the threshold is set as 10 by default). When a node has actually failed, the checking condition at Line 9 fails. We migrate the failed ReduceTasks to a healthy node (Lines 14 – 20).

Our speculative scheduler proactively re-launches MapTasks from a failed node such that these MapTasks can regenerate their MOFs (Lines 5 – 7). This avoids long stalls to other running ReduceTasks so that they are not detected as failed because of the lack of progresses. In so doing, spatial amplification of ReduceTask failures is prevented. By the same token, with the MOFs regenerated proactively, a recovering ReduceTask does not have to locate missing MOFs, thus no risk of being stalled again and marked as another failure. Therefore the temporal amplification is eliminated. Note that, proactively re-executing MapTasks may lead to unnecessary regeneration of MOFs. For example, all ReduceTasks may have progressed beyond the shuffle/merge stage or the failed ReduceTask can leverage the analytics logs stored on HDFS. MapTasks are generally short-living and their re-execution is typically light weight, as shown in Fig. 1. Therefore, because of the criticality in cracking down failure amplification, it is a small cost well paid to speculative re-execution of MapTasks on the failed node to ensure the availability of MOFs.

V. Experimental Evaluation

We have thoroughly evaluated the effectiveness of our ALM framework for failure resiliency and its performance overhead.

A. Experimental Environment

TABLE I: List of key YARN configuration parameters.

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

Cluster Setup: all the experiments are conducted on a cluster featuring 21 machines that are connected through 10 Gigabit Ethernet. Each machine is equipped with four 2.67 GHz hex-core Intel Xeon X5650 CPUs, 24 GB memory and 1 SATA-based SSD.

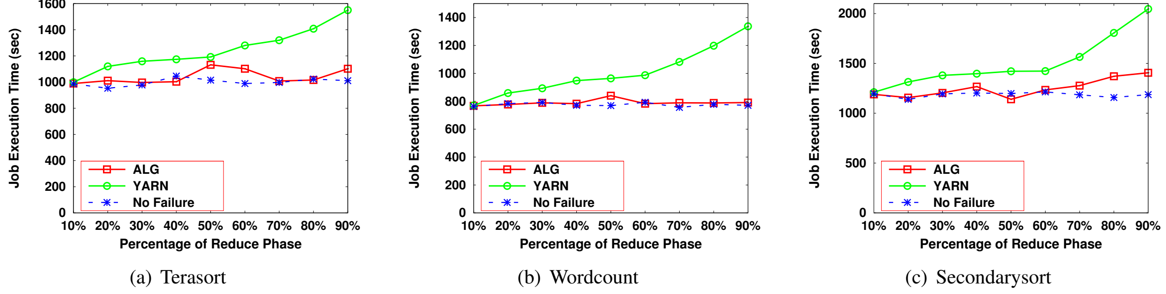
Configuration: we use YARN-v2.2.0 as the code base with JDK 1.7. One node is dedicated as the ResourceManager and the NameNode of YARN and HDFS. Table I lists the key parameters that are relevant to the performance of YARN programs, along with their tuned values.

Benchmarks: we have selected three representative MapReduce programs, including Terasort, Wordcount, and Secondarysort. Detailed characteristics of above three benchmarks can be found in [17], [3], [21], [19].

B. Throttling the Job Execution Delay

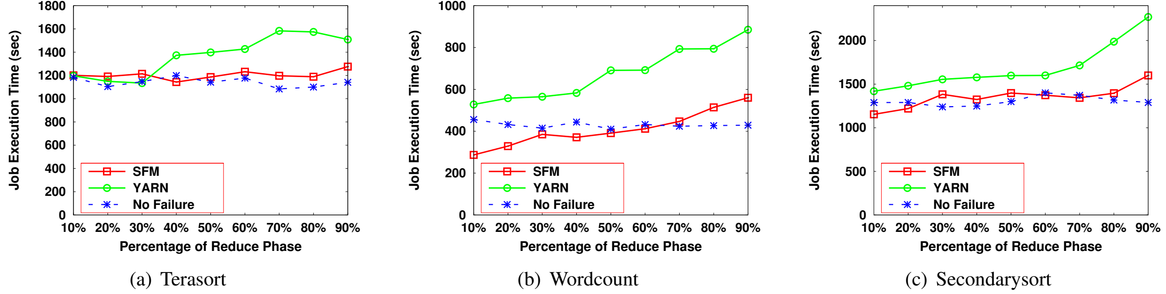
We start by analyzing the effectiveness of our ALM framework on throttling the execution delay caused by task and node failures as described in Section II-B. In this test, we have used all three benchmarks. The input sizes for Terasort, Wordcount, and Secondarysort are 100 GB, 10 GB, and 10 GB, respectively. We inject out-of-memory exceptions to crash a task to emulate the transient task failures and stop the network services on a node for node failures. Each of the results is the average of three test runs.

Fig. 8 shows that, in the event of a single ReduceTask failure, our ALG technique can effectively speed up the process of failure recovery. This is because ALG can avoid repeating the accomplished analytics progresses such as the shuffle, merge or reduce work in the ReduceTask. At the job execution level, on average, ALG efficiently outperforms YARN by 15.4%, 20.1% and 15.9% for Terasort, Wordcount, and Secondarysort, respectively, for 9 different failure points. In particular, when failures are induced at the 90% progress of ReduceTasks, ALG provides up to 28.9%, 40.8% and 31.3% improvement. Meanwhile, its performance is comparable to the failure-free case, indicating that ALG can achieve failure recovery at little overhead.



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

Fig. 8: Effective failure recovery with ALG at little overhead.



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

Fig. 9: SFM substantially shortens migration and recovery of ReduceTasks without much performance impact.

As also shown in Fig. 8, ALG mitigates the performance variation that exists in YARN when transient failures occurs at different time points. This is a very desirable feature because it can help MapReduce jobs to achieve predictable turnaround time. Across three benchmarks, compared to the cases that failures occur at 10% of Reduce phase, comparing failures at different progress points, the job execution time does increase noticeably. Failures at 90% progress can increase the execution time by 11.3%, 3.1%, and 18.3% for Terasort, Wordcount, and Secondarysort, respectively, compared to failures at 10% progress. The difference is significantly better with ALG compared to YARN. With the original YARN, failures at 90% progress can increase the job execution time by as much as 55.2%, 73.6%, 68.9%, for Terasort, Wordcount, and Secondarysort, respectively, compared to failures at 10% progress. Such significant delays of job execution time due to failures can lead to less predictability of job completion.

Complementary to ALG, SFM aims to rapidly recover from node failures to prevent migrated ReduceTasks from becoming stragglers. Fig. 9 illustrates the migration and recovery performance of SFM when a node failure occurs at different points of the reduce phase. At the job execution level, compared to YARN, SFM shortens the migration and recovery process by 10.9%, 39.4%, and 18.8% on average for Terasort, Wordcount, and Secondarysort, respectively. In particular, when failures occur at the 90% progress, SFM achieves an improvement by up to 15.4%, 36.7% and 29.5%, respectively, for the three benchmarks, This is because, with SFM, the migration and recovery of ReduceTasks is facilitated by speculative scheduling of MapTasks and ReduceTasks and by the availability of previously logged analytics records and enhanced migration execution mechanism.

Interestingly, for Wordcount that only contains a single ReduceTask, when failures happen at early phase (before 50%), SFM with a failure can even outperform the failure-free case. This is due to acceleration obtained from collective merging and pipelined internal execution, which together eliminate the reliance on disks for intermediate data merging. However, SFM requires the synchronization of all participant nodes for emergency handling and bookkeeping of intermediate data merging progress. This can affect the scalability of the system and consume higher resources compared to the execution of a plain ReduceTask. Thus, we only advocate SFM as a task migration and recovery mechanism upon failures, not an alternative implementation of regular ReduceTasks.

C. Cracking Down Failure Amplification

As discussed in Section II-C, the failure recovery mechanism in YARN can lead to temporal and spatial failure amplification due to the reliance on newly recovered ReduceTasks to detect lost map output files. Our ALM framework provides a speculative scheduling mechanism for the recovery, which removes such dependency and actively regenerates lost intermediate data when node failures are detected.

In this section, we use Wordcount to evaluate the issue of temporal failure amplification because of its simplistic composition of a single ReduceTask. The profiling result of the reduce phase is shown in Fig. 10. We use Terasort to evaluate the impact of ALM on spatial failure amplification. Table II illustrates the profiling results of spatial amplification.

As shown in Fig. 10, upon detecting the failure at around 116 seconds, instead of immediately relaunching a ReduceTask, SFM firstly prioritizes the regeneration of lost map output files by launching MapTasks. Though such decision

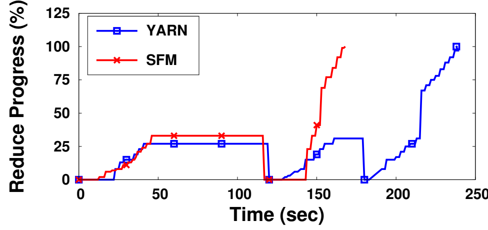


Fig. 10: SFM eliminates temporal amplification and efficiently improves the failure recovery.

can delay the launch time of a recovery task by 18 seconds, the newly-launched ReduceTask does not suffer from repeated timeouts when trying to retrieve the map output files from the failed node, thereby avoiding any preemption by the YARN scheduler. With such speculative migration and recovery of tasks and data, SFM eliminates temporal failure amplification.

TABLE II: Speculative Recovery Scheduling curbs the infectious impact of node failures.

Type	Point of First Failure	Number of Additional Failures	Execution Time
YARN	10%	2	429 seconds
SFM	10%	0	435 seconds
YARN	20%	5	533 seconds
SFM	20%	0	449 seconds
YARN	30%	3	516 seconds
SFM	30%	0	445 seconds

Furthermore, SFM prevents task failures on one failed node from inflicting failures of other healthy ReduceTasks as shown in Table II. In the original YARN, even if a failed node does not contain any running ReduceTask, the lost map output files can cause more ReduceTasks to fail as shown in Table II, resulting in spatial failure amplification and delayed job execution. In contrast, SFM cracks down such amplification and prevents additional ReduceTasks from being affected. By the time a healthy ReduceTask reports connection failures, SFM is aware of the cause and requests ReduceTask to wait until the lost map output files are regenerated.

Taken together, our detailed profiling adequately corroborates that SFM can crack down both temporal and spatial failure amplifications and exhibit better failure resilience.

D. Performance Characteristics of ALG

Performance Penalty on Normal Job Execution: Fig. 11 compares the performance between YARN and ALG in the failure-free environments using Terasort with different input sizes, ranging from 10 GB to 320 GB. The results show that ALG incurs negligible penalty to ReduceTasks, with little degradation on the job execution under failure-free scenario.

Performance at Different Logging Frequencies: ALG is insensitive to the logging frequency. Fig. 12 shows the performance of Terasort under different frequencies. As shown in the figure, ALG exhibits fairly stable performance. More importantly, we have observed that the more frequently ALG is invoked, the lower interference it causes to the task execution

Fig. 11: Negligible overhead of ALG in failure-free scenarios.

due to the smaller amount of analytics progress in each period. Such characteristic allows us to frequently log the analytics work of MapReduce programs and minimize the time for recovering analytics progress after failures.

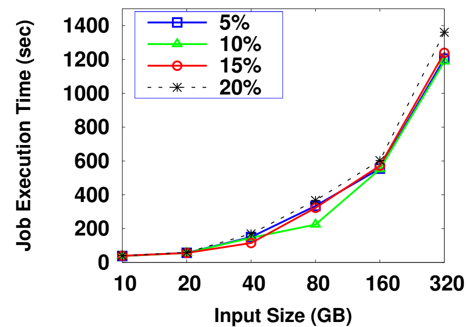


Fig. 12: ALG performance at different logging frequencies.

Performance with Different Replication Levels: Another factor that helps reduce overhead is that ALG constrains the replication level within a single rack rather than replicating across an HDFS cluster. It only replicates entire blocks when committing tasks. As shown in Fig. 13, increasing the replication level to the cluster level incurs significant overhead on the reduce phase. Compared to the node-level replication, constraining the replication within a rack imposes small overhead for small datasets. When the data size increases to 320 GB, replication at the rack level delays the reduce progress by about 18.4%. However, the progress degradation significantly rises to 55.7% for the cluster-level replication.

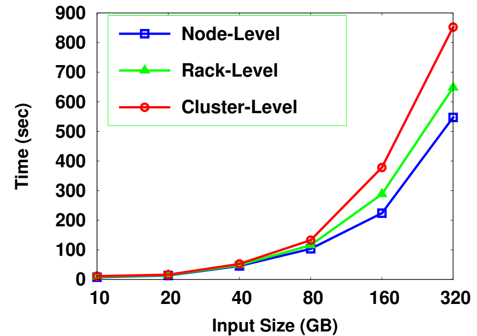


Fig. 13: Impact of different replication levels on reduce stage.

E. Performance Characteristics of SFM

Performance under Concurrent Failures: We measure the performance of MapReduce programs under multiple concurrent failures. As shown in Fig. 14, SFM achieves substantial performance improvement compared to YARN when multiple concurrent failures occur to the ReduceTasks. Recovery performance between YARN and SFM when each ReduceTask needs to process different data sizes, ranging from 1 GB to 32 GB.

On average, SFM cuts down on the recovery time by up to 40.7%, 44.3%, 49.5% for 1, 5 and 10 concurrent failures cases, respectively. In addition, the improvement ratio increases linearly with the data sizes, rising from 37.2% for recovering 1 GB to 62.1% for recovering 32 GB under 5 concurrent failures case. With more intermediate data, the performance of original ReduceTask is heavily subject to the intermediate data merging, which leads to drastic performance degradation when disks are under heavy workload. On the contrary, SFM eliminates such bottleneck, and its performance is determined by the network bandwidth and aggregated disk bandwidth from servers that supply intermediate data.

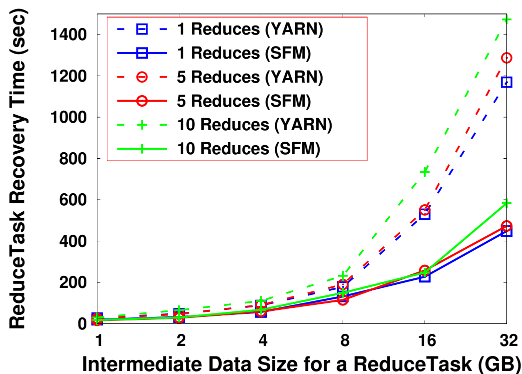


Fig. 14: SFM for the recovery of multiple concurrent failures.

Effectiveness of Leveraging Logged Analytics: SFM is designed to leverage the logged analytics work by ALG to avoid unnecessary data deserialization and reduce computation. As shown in Fig. 15, ALM with integrated SFM and ALG (SFM+ALG) can further accelerate the recovery process of Terasort, Wordcount, and Secondarysort by 11.4%, 16.1% and 25.8%, respectively, compared to the SFM-only case. For Terasort, its reduce function outputs the <k,v> pairs. The improvement comes from the avoidance of deserializing the intermediate data. For Secondarysort, the improvement comes from the resurrection of logged progress on reduce computation. This substantially helps improve the recovery process of Secondarysort. Therefore a more profound improvement ratio is observed for Secondarysort than the other two benchmarks. Note that such improvement requires ALG to replicate the reduce results during the reduce phase, which can slow down the normal progress of ReduceTasks with large datasets as shown in Section V-D. Therefore, our ALM framework can be executed with both ALG and SFM enabled, but users need to be aware of the potential overhead when ALM has to replicate large output of ReduceTasks.

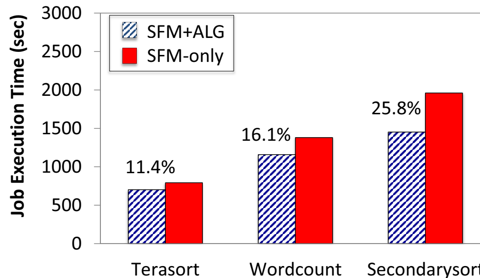


Fig. 15: Benefits of enabling both ALG and SFM.

VI. Related Work

Although there is a large body of research related to improving the performance of MapReduce, little research has been conducted to investigate its failure resiliency issue. In this section, we provide a review of several closely related works on failure characteristics, failure detection, and recovery algorithms.

Ko *et al.* designed an Intermediate Storage System (ISS) [11], an extension of HDFS, to enable highly available intermediate data, so that MapReduce jobs can avoid repeating the same MapTasks after node failures. However, to ensure such availability, ISS needs to replicate intermediate data on HDFS at the cost of imposing heavy overhead on the file system and network. This issue has also been discussed in [8], [14]. Relying on the original ReduceTask to recover from task or node failures, ISS suffered from the same performance collapse as the existing YARN. ISS was an early work on improving the failure recovery of MapReduce frameworks. It failed to recognize the severe issues associated with ReduceTask failures. Our work examines these open issues and provides a set of techniques to address them.

Quiané-Ruiz *et al.* introduced RAFTing MapReduce [14] to preserve the computation status of MapTasks while replicating the intermediate map output to remote servers and avoiding the need of recomputing the MapTasks on the crashed node. It was similar to [11] in spirit. Although such design could efficiently recover failed MapTasks, it did not address the slow recovery issue of ReduceTasks either. In addition, two factors can limit its applicability. Firstly, RAFTing MapReduce requires the pre-assignment of ReduceTasks such that MapTasks can push the intermediate data to the ReduceTask during task execution. However, such pre-assignment is hard, if not infeasible, to achieve in a shared MapReduce cluster, especially when it is heavily loaded as discussed in [6], [16], [18]. Secondly, replicating map output files can incur a significant performance overhead when the intermediate data size is large [11]. In contrast, our work is not constrained by the above factors and applicable to general MapReduce clusters.

Dinu *et.al.* [8] systematically analyzed the impact of node failures on MapReduce programs. In particular, they studied how the failures of TaskTrackers and DataNodes in Hadoop

1.x affect the performance. They revealed several insightful observations: (1) a single failure can have significant performance impact on MapReduce applications; (2) failure information is not shared among running tasks, causing local failures to propagate to healthy tasks, which is similar to the cascading infection introduced in this paper; and (3) existing straggler detection algorithms [24], [4] make unrealistic assumptions on the task progress. As a result, DataNode failures can render those algorithms ineffective, causing an issue of delayed speculative execution which can severely slow down the progress of the reduce phase. Our work is inspired by this analysis but provides a thorough quantification of job execution delays and failure amplifications. Furthermore, we have developed techniques to solve these problems.

VII. Conclusion

We have examined the resiliency of MapReduce to task and node failures using the next-generation Hadoop framework, i.e., YARN. This is an issue that has received little attention, but revealed as serious in our experimental analysis. In particular, we have observed drastic performance degradation due to ReduceTask failures and cascading task failures due to node failures, which we refer to as failure amplification. In this paper, we have designed and implemented a fault-tolerant MapReduce framework that overcomes the performance degradation and failure amplification issues. Specifically, we introduce two techniques called analytics logging and speculative fast migration to alleviate the impacts of task and node failures. We have thoroughly evaluated the benefits of these techniques to a variety of failure scenarios. Our experimental results demonstrate that that our techniques can crack down failure amplification and deliver fast job execution compared to the existing task re-execution mechanism in MapReduce.

Acknowledgment

This work is funded in part by an Alabama Innovation Award, and by National Science Foundation awards 1059376, 1320016, 1340947, and 1432892. Yandong Wang contributed to the research as a graduate student at Auburn. He is currently affiliated with IBM Watson.

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] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 61–74, New York, NY, USA, 2012. ACM.

[4] 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.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, chapter 42. Crash Consistency: FSCK and Journaling. Arpaci-Dusseau Books, 2012.

[6] 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.

[7] 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.

[8] 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.

[9] Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. In *ICPP*, pages 471–478. IEEE Computer Society, 2006.

[10] 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.

[11] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 181–192, New York, NY, USA, 2010. ACM.

[12] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 985–996, New York, NY, USA, 2011. ACM.

[13] X. Li, Y. Wang, Y. Jiao, C. Xu, and W. Yu. Coomr: Cross-task coordination for efficient data management in mapreduce programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 42:1–42:11, New York, NY, USA, 2013. ACM.

[14] 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.

[15] 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.

[16] 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.

[17] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[18] 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.

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

[20] Y. Xu, P. Kostamaa, and L. Gao. Integrating hadoop and parallel dbms. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 969–974. ACM, 2010.

[21] Yandong Wang and Cong Xu and Xiaobing Li and Weikuan Yu. JVM-Bypass shuffling for hadoop acceleration. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[22] Yandong Wang and Robin Goldstone and Weikuan Yu and Teng Wang. Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. In *28th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2014)*, Phoenix, USA, May 2014.

[23] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[24] 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.