Understanding the Effects and Implications of Compute Node Related Failures in Hadoop

Florin Dinu   T. S. Eugene Ng
Computer Science Department;Rice University

ABSTRACT

Hadoop has become a critical component in today's cloud environment. Ensuring good performance for Hadoop is paramount for the wide-range of applications built on top of it. In this paper we analyze Hadoop's behavior under failures involving compute nodes. We find that even a single failure can result in inflated, variable and unpredictable job running times, all undesirable properties in a distributed system. We systematically track the causes underlying this distressing behavior. First, we find that Hadoop makes unrealistic assumptions about task progress rates. These assumptions can be easily invalidated by the cloud environment and, more surprisingly, by Hadoop's own design decisions. The result are significant inefficiencies in Hadoop's speculative execution algorithm. Second, failures are re-discovered individually by each task at the cost of great degradation in job running time. The reason is that Hadoop focuses on extreme scalability and thus trades off possible improvements resulting from sharing failure information between tasks. Third, Hadoop does not consider the causes of connection failures between its tasks. We show that the resulting overloading of connection failure semantics unnecessarily causes an otherwise localized failure to propagate to healthy tasks. We also discuss the implications of our findings and draw attention to new ways of improving Hadoop-like frameworks.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Measurement, Performance, Reliability

Keywords

Failures, Hadoop, Speculative Execution

1. INTRODUCTION

Hadoop has become a cloud workhorse [7]. Major Internet companies rely on Hadoop for their everyday needs involving extremely large data sets [43, 44, 9]: management tasks involving log processing [16, 8], business intelligence applications or the deployment of new products and platforms [18, 44]. Cloud service providers have added Hadoop to their list of offerings [10, 3]. Scientists use Hadoop for a wide range of purposes. To name only a few examples, Hadoop facilitated the implementation of scalable solutions and algorithms for data-intensive text processing [2], assembly of large genomes [4], graph mining [12], machine learning and data mining [1] and large scale social network analysis [13]. Hadoop has also received much attention from the research community. Several studies propose performance improvements [28, 34] or extensions to Hadoop: running Hadoop on a wider range of job types [19, 21], in more challenging environments [50], as a back-end in other large scale systems [44, 38] or as part of a hybrid architecture [14]. As a result of its widespread use and of the critical nature of the applications running on top of it, ensuring good performance for Hadoop jobs is essential.

In this paper, we focus on Hadoop's behavior under compute node failures. The same cloud environments that host Hadoop applications are typically prone to compute node failures failures. Studies show [22, 45, 5] tens of compute node related failures per day and multiple failures per average compute job. Moreover, various environmental conditions such as network components failing or bandwidth quotas being exceeded can be indistinguishable at application level from compute node failures. Rightfully so, compute node failures are becoming a driving force behind the design of large-scale cloud applications [24]. Despite the pervasiveness of failures in the cloud, little research work has been done on analyzing Hadoop's performance under failures and understanding the efficiency of its design decisions in the context of failures.

Given Hadoop's popularity and the fact that failures are the norm rather than the exception in the cloud environment this research direction has immediate practical relevance. Our paper is the first to provide a thorough analysis of Hadoop under failures. The problem of dealing with failures is complex and our goal is to provide deep and insightful technical analysis. We view our paper as a necessary first step for solving what has become a chronically over-looked aspect: designing and building more robust failure detection and recovery algorithms for Hadoop. To this end, a collaborative effort from the community is needed: failure characteristics, job characteristics and cloud resource occupancy in real deployments need to be analyzed. Thus, in addition to the practical relevance this research direction is rich in avenues for impactful future work.

Specifically, in this paper we analyze Hadoop's behavior under fail-stop failures of entire compute nodes and under fail-stop failures of the Hadoop components running on compute nodes (TaskTracker and DataNode). DataNode failures are important because they affect the availability of job input and output data and also delay read and write data operations which are central to Hadoop's performance. TaskTracker failures are equally important because they affect running tasks as well as the availability of intermediate data (i.e. map outputs). Unlike failures affecting logically centralized Hadoop components (JobTracker and NameNode) which can be addressed by distributed coordination mechanisms [39], compute node failures require Hadoop to take explicit measures for detection and recovery and are more likely to cause subtle interactions with the environment or between Hadoop's components.

Our measurements point to a real need for improvement. Surprisingly, we discover that a single failure can lead to large, variable and unpredictable job running times. For example, the running time of a job that takes 220s without failures can vary from 220s to as much as 1000s under TaskTracker failures and to 700s under DataNode failures. This is especially important for short jobs which are frequently encountered in the cloud and have been shown to be a major use case for Hadoop [50, 23, 20]. Such large performance variations are detrimental. They cause unpredictable user costs and prolonged waiting for results, decrease overall cloud utilization and complicate scheduling. In our experiments on a predictable cloud environment, the primary causes for the performance variations are internal to Hadoop's design, namely the inefficiencies in Hadoop's failure detection and recovery algorithms.

We expose three important inefficiencies in Hadoop's design which manifest themselves under compute node failures. First, Hadoop makes unrealistic assumptions about task progress rates. Hadoop seems to think that, with the exception of a few under-performing outliers, tasks progress at comparable rates. For Hadoop this warrants the use of a statistical speculative aggregation algorithm centered around average progress rates. Unfortunately, Hadoop's assumption can be easily invalidated in practice. Both the cloud environment as well as other Hadoop design decisions can result in very fast progressing tasks. For example, a number of recent proposals for improved cloud network design [35, 46] advocate accelerating specific network paths. Alternatively, imbalanced computations can lead to reducer tasks which are very fast because they process little data. Also, in a Hadoop job with multiple reducer waves, reducer tasks not belonging to the first wave can progress at very high rates because they do not have to wait for their input. We show that when Hadoop's assumption is invalidated, a negative effect which we call *delayed speculative execution* can appear. This consists in one speculative execution decision severely delaying or even precluding subsequent speculative executions at great overall costs for the job running time.

Second, Hadoop trades off possible improvements resulting from communication between tasks for extreme scalability. Therefore, each compute task performs failure detection and recovery on its own. The unfortunate effect of this lack of sharing failure information is that multiple tasks could be left wasting time re-discovering a failure that has already been identified by another task. Moreover, a speculated task may have to re-discover the same failure that hindered the progress of the original task in the first place. We find that both Hadoop's speculative execution algorithm as well as the LATE algorithm [50] can be significantly impacted by failures. Importantly, these findings suggest that even state-of-the-art approaches to cause-aware speculative execution [15] may be insufficient. This is because a good speculative execution decisions can be invalidated at runtime when the speculated task is affected by a failure. To ensure that speculated tasks help improve job running time, failure information needs to be effectively shared between tasks at job runtime.

Third, Hadoop uses connection failures between its tasks as a heuristic for detecting node failures. In part, this is warranted by the limited visibility a cloud application can obtain about the cloud environment. Unfortunately, several factors can cause connection failures without implying node failures. Temporary overload conditions such as network congestion or excessive end-host load can cause connection failures. All these conditions are common in data centers [17, 22]. As a result, from only the news of a connection failure Hadoop cannot reliably distinguish an underlying cause. We show that this limitation unnecessarily introduces additional failures into the system. Specifically, otherwise localized failures involving a compute node can propagate to tasks running on healthy nodes. We call this the *induced reducer death* problem.

Our findings are not obvious. The points we highlight about real-life system building decisions and real-life subtle interactions are crucial. In practice, these decisions and interactions often invalidate benefits obtainable through smart solutions built on top. See for example the unexpected but serious ways in which failures affect the LATE and Hadoop speculative execution algorithms.

The paper is organized as follows. In §2 we review relevant Hadoop material. In §3 we present how Hadoop deals with failures today. In §4 we give detailed experimental evidence of Hadoop's design inefficiencies. We discuss implications, lessons learned and new ways of improving Hadoop-like frameworks in §5. Finally, we review related work in §6 and conclude in §7.

2. BACKGROUND AND NOTATION

2.1 Notation

For brevity we use the term speculation to refer to *speculative execution*. We say that a task was *speculated* when a new instance of the task was speculatively executed. We distinguish between the *initial* instance of a task and subsequent speculative instances of the same task. We use WTO, RTO and CTO to signify write, read and connect timeouts.

2.2 Hadoop Background

Hadoop [6, 49] separates a job computation into two types of tasks: mappers and reducers. First, mappers read the job input data from Hadoop's distributed file system (HDFS) and produce as their output key-value pairs. These map outputs are stored locally on compute nodes, they are not written to HDFS. The map outputs comprise the input for the reducer tasks. Each reducer processes a separate key range. For this, reducers copy the part of the map outputs which contains values within that key range. This copy phase is called shuffling. Oftentimes, reducers needs to copy part of the output of every single map. Finally, a reducer writes the job output data to HDFS.

Each task has a progress score which attempts to capture how close the task is to completion. The score is 0 at the task's start and 1 at completion. For a reducer, a score of 0.33 signifies the end of the copy (shuffle) phase. At 0.33 all map outputs have been read. A score of 0.66 signifies the end of the sort phase. Between 0.66 and 1 a reduce function is applied and the output data is written to HDFS. The progress rate of a task is the ratio of the progress score over the current task running time. For example, it can take a task 15s to reach a score of 0.45, for a progress rate of 0.03/s.

HDFS is composed of a centralized NameNode and of distributed DataNodes running on compute nodes. DataNodes handle the read and write operations to the HDFS. The NameNode manages the file system metadata and decides which DataNodes data should be read from or written to. HDFS write operations are pipelined. In a pipelined write an HDFS block is replicated at the same time on a number of nodes dictated by a configured replication factor. For example, if data is stored on node A and needs to be replicated on B and C, then, in a pipelined write, data flows from node A to B and from B to C. A WTO/RTO occurs when a HDFS write/read operation is interrupted by a DataNode failure. WTOs occur for reducers while RTOs occur for mappers. CTOs can occur for both mappers and reducers, when they cannot connect to a DataNode.

A TaskTracker is a distributed Hadoop component running on compute nodes which is responsible for starting and managing tasks locally. TaskTrackers are configured with a number of mapper and reducer slots, the same number for every TaskTracker. If a TaskTracker has two reduce slots then a maximum of two reducers can concurrently run on it. If a job requires more reducers (or mappers) than the number of reducer (mapper) slots in the system then the reducers (mapper) are said to run in multiple waves. A TaskTracker communicates regularly with a Job Tracker, a centralized Hadoop component that manages jobs and decides when and where to start new tasks.

2.3 Speculative Execution Background

The JobTracker runs a speculative execution algorithm which attempts to improve job running time by duplicating under-performing tasks. The algorithm in Hadoop 0.21.0 (the version we use in this paper) is a variant of the LATE algorithm [50]. Both algorithms rely on progress rates. Both select a set of candidate tasks for speculation and then execute the candidate task that is estimated to finish farthest in the future. The difference lies in the method used to select the candidates. Hadoop takes a statistical approach. A candidate for speculation is a task whose progress rate is slower by at least one standard deviation than the average progress rate of all started tasks of the same kind (i.e. map or reduce) that belong to one job. Let $Z(T_i)$ be the progress rate of a task T_i and T_{set} the set of all running or completed tasks of the same kind. A task T_{cur} can be speculatively executed if:

$$avg(Z(T_i)_{T_i \in T_{set}}) - std(Z(T_i)_{T_i \in T_{set}}) > Z(T_{cur}) \quad (1)$$

Intuitively Hadoop speculates an under-performing task only when large variations in progress rates occur. In contrast, LATE attempts to speculate tasks as early as possible. For LATE, the candidates are the tasks with the progress rate below a SlowTaskThreshold, which is a percentile of the progress rates for a specific task type. Both algorithms speculate a task only after it has ran for at least 60s. To minimize the impact on available resources both algorithms cap the number of active speculative task instances at 1.

3. FAILURES IN HADOOP

In this section we describe the mechanisms that Hadoop uses to guard against failures. Alongside the speculative execution algorithm described in (§2.3) these mechanisms cause the serious inefficiencies that we uncover in this paper.

We identified these mechanisms by performing source code analysis on Hadoop version 0.21.0. The experiments in the rest of the paper are also based on 0.21.0. At the beginning of November 2011 version 0.21.0 was still the highest Hadoop version available. Recently, Hadoop has moved from the 0.2x versions to the 1.0.x versions. While we have not tested these latest version we have performed a code-level comparison between versions 0.21.0 and 1.0.0. We find that the mechanisms described in this section have remained the same, thus showing that the mechanisms are not short-lived but rather are deeply rooted in Hadoop's design philosophy. The one change we have found concerns the speculative execution algorithm. In 1.0.0, Hadoop has reverted to an older algorithm found in versions 0.20.x. Unfortunately, that particular algorithm has already been shown to have serious inefficiencies [50], a conclusion which lead to the development of the improved algorithm that we analyze in this paper.

3.1 How Hadoop Deals with TaskTracker Failures

Hadoop infers TaskTracker failures by comparing task state variables against tunable threshold values. Table 1 lists the variables used by Hadoop. These variables are constantly updated by Hadoop during the course of a job. For clarity, we omit the names of the thresholds and instead use their default numerical values.

As we examine in detail the decisions related to TaskTracker failures, it shall become apparent that tolerating network congestion and compute node overload is a key driver of many aspects of Hadoop's design. It also seems that Hadoop attributes non-responsiveness primarily to congestion or overload rather than to failure, and has no effective way of differentiating the two cases. To highlight some findings:

- Hadoop is willing to wait for non-responsive nodes for a long time (on the order of 10 minutes). This conservative design allows Hadoop to tolerate non-responsiveness caused by network congestion or compute node overload.
- A *completed* map task whose output data is inaccessible is re-executed very conservatively. This makes sense if the inaccessibility of the data is rooted in congestion or overload. This design decision is in stark contrast to the much more aggressive speculative re-execution of straggler tasks that are *still running* [50].
- The health of a reducer is a function of the progress of the shuffle phase (i.e. the number of successfully copied map outputs). However, Hadoop ignores the underlying cause of unsuccessful shuffles.

3.1.1 Declaring a TaskTracker Dead

TaskTrackers send heartbeats to the JobTracker every 3s. The JobTracker detects TaskTracker failures by checking every 200s if any TaskTrackers have not sent heartbeats for at least 600s. If a TaskTracker is declared dead, the tasks running on it at failure time are restarted on other nodes. Map tasks that completed on the dead TaskTracker are also restarted if the job is still in progress and contains any reducers.

3.1.2 Declaring Map Outputs Lost

The loss of a TaskTracker makes all map outputs it stores inaccessible to reducers. Hadoop recomputes a map output early (i.e. does not wait for the TaskTracker to be declared dead) if the JobTracker receives enough notifications that reducers are unable to obtain the map output. The output of map M is recomputed if:

$$N_j(M) > 0.5 * R_j \quad \text{and} \quad N_j(M) \geq 3.$$

Let L be the list of map outputs that a reducer R wants to copy from TaskTracker H. A notification is sent immediately if a read error occurs while R is copying the output of some map M1 in L. $F_j^R(M)$ is incremented only for M1 in this case. If on the other hand R cannot connect to H, $F_j^R(M)$ is increased by 1 for every map M in L. If, after several unsuccessful connection attempts $F_j^R(M) \ mod \ 10 = 0$ for some M, then the TaskTracker responsible for R sends a notification to the JobTracker that R cannot copy M's output. A back-off mechanism is used to dictate how soon after a connection error a node can be contacted again for map outputs. After every failure, for every map M for which $F_j^R(M)$ is incremented, a penalty is computed for the node running M:

Var.	Description	Var.	Description	Var.	Description
P_j^R	Time from reducer R's start until it last made progress	R_j	Nr. of reducers currently running	T_j^R	Time since reducer R last made progress
M_j	Nr. of maps (input splits) for a job	D_j^R	Nr. of map outputs copied by reducer R	S_j^R	Nr. of maps reducer R failed to shuffle from
$F_j^R(M)$	Nr. of times reducer R failed to copy map M's output	A_j^R	Total nr. of shuffles attempted by reducer R	Q_j	Maximum running time among completed maps
$N_j(M)$	Nr. of notifications that map M's output is unavailable.			K_j^R	Nr. of failed shuffle attempts by reducer R

Table 1: Variables for failure handling in Hadoop. The format is $X_j^R(M)$. A subscript denotes the variable is per job. A superscript denotes the variable is per reducer. The parenthesis denotes that the variable applies to a map.

$$penalty = 10 * (1.3)^{F_j^R(M)}.$$

A new timer is set to *penalty* seconds in the future. Whenever a timer fires another connection is attempted.

3.1.3 *Declaring a Reducer Faulty*

A TaskTracker considers a reducer running on it to be faulty if the reducer failed too many times to copy map outputs. Three conditions need to be simultaneously true for a reducer to be considered faulty. First,

$$K_j^R \geq 0.5 * A_j^R.$$

In other words at least 50% of all shuffles attempted by reducer R need to fail. Second, either

$$S_j^R \geq 5 \quad \text{or} \quad S_j^R = M_j - D_j^R.$$

Third, either the reducer has not progressed enough or it has been stalled for much of its expected lifetime.

$$D_j^R < 0.5 * M_j \quad \text{or} \quad T_j^R \geq 0.5 * max(P_j^R, Q_j).$$

Note that for Hadoop only the existence of a connection failure is important but not the cause of the failure.

3.2 How Hadoop Deals with DataNode Failures

Hadoop detects DataNode failures using connection errors and timeouts. If a timeout expires or an existing connection is broken, the read or write operation is restarted with new source or destination nodes obtained from the NameNode.

The timeouts used by HDFS requests to recover from DataNode failures are conservatively chosen, likely in order to accommodate transient congestion episodes which are known to be common to data centers [17]. Both an initial task and a speculative task can suffer from these timeouts. RTOs and CTOs are on the order of 60s while the WTOs are on the order of 480s. Differences of 5s-15s in absolute timeout values exist and depend on the position of a DataNode in the HDFS write pipeline. For this paper's argument these minute differences are inconsequential.

4. EXPERIMENTS EXPOSING THE INEFFICIENCIES OF FAILURE DETECTION AND RECOVERY IN HADOOP

4.1 Methodology

For our experiments we used 15 machines from 4 racks in the OpenCirrus testbed [11]. One node is reserved for the JobTracker and NameNode, the rest of the nodes run DataNode and TaskTracker processes. Each node has 2 quad-core Intel Xeon E5420 2.50 Ghz CPUs. The network is 10 to 1 oversubscribed. We run Hadoop 0.21.0 with the default configuration. Importantly, the compute nodes as well as the network were not shared with other users. The resources were solely used by our Hadoop jobs. Even more, the compute nodes were not virtualized. As a result the performance of our testbed was predictable. This allows us to clearly identify the performance variations caused by Hadoop's design.

We independently study DataNode and TaskTracker failures because in many Hadoop deployments DataNodes and TaskTrackers are collocated and therefore, under compute node failures it would be hard to single-out the underlying cause. We first analyze TaskTracker failures. After the TaskTracker failure experiments one of the compute nodes became permanently disabled because of a hardware issue and this left us with one less compute node for the DataNode failure analysis. Fortunately, the two sets of results are independent, therefore this failure does not affect out findings.

The job we use for this paper sorts 10GB of random data using 2 map slots per node and 2 reduce slots per node. In the experiments we vary the number of reducers and the number of reducer waves. 200 runs are performed for each experiment. Without failures the job takes on average 220s to complete. We chose this relatively short job because current studies show the significant popularity of short jobs in cloud workloads [50, 23, 20]. Our goal is not to exhaustively and quantitatively analyze Hadoop performance over many job and failure types. Instead, our aim is to expose the inefficiencies, the subtle interactions and the underlying design decisions in Hadoop. Nevertheless, we argue that the thorough understanding obtained from our paper is also insightful for longer jobs and for jobs running on larger deployments. Hadoop's failure detection and recovery algorithms (§3) remain the same regardless of scale because they use non-adaptive timeouts and the proportion of TCP connection failures. Even for multiple failures (which are more probable in larger deployments or longer jobs), the same algorithms apply. Also, oftentimes the multiple failures are independent and they have a cumulative effect. This effect can be estimated as the sum of the effect of single failures.

We consider TaskTracker and DataNode processes failures as well as the failure of the entire compute node running these processes. The difference between the two failure types lies in the existence of TCP reset (RST) packets that are sent by the host OS when a process is killed. RST packets may serve as an early failure signal. We induce the single DataNode (or TaskTracker) fail-stop failures by randomly killing one of the DataNodes (or TaskTrackers) at a random time after the job is started and before the 220s mark. At the end of each run we restart Hadoop. We simulate a fail-stop failure of the compute node running a DataNode (or TaskTracker) by filtering all RST packets sent after the failure if the

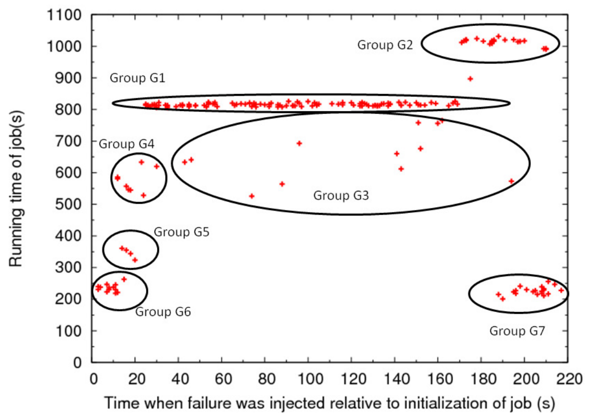

Figure 1: Clusters of job running times under TaskTracker failure. Without any failures the average job running time is 220s

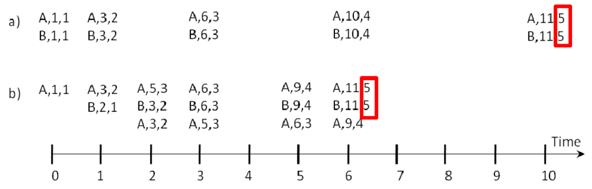

Figure 2: Illustration of early notifications. The tuple format is (map name, time the penalty expires, $F_j^R(M)$). For example, (A,3,2) means at time 3 a new connection should be attempted to get map output A, and there have been 2 failed attempts so far. The tuple values are taken immediately after the corresponding timestamp. This example considers that notifications are sent when $F_j^R(M) = 5$. Note that this occurs at different moments, shown by rectangles.

source port of the RST packet corresponds to the ports used by the failed DataNode (or TaskTracker). In our experiments we look at how failures impact the job running time and the job startup time. We consider the job startup time to be the time between the job submission and the job start assuming no waiting for task slots and no job queueing delays. We consider the job running time to be the time between the job start and job end.

Here is a quick roadmap of the experiments that follow. Details and arguments for our experiment choices can be found alongside the experiments. In the first experiment we analyze in great detail TaskTracker process failures and find significant performance variations. The subsequent three experiments confirm and expand on our findings for an increased number of reducers, for running a second concurrent job and for simulating TaskTracker node failures. For DataNode failure we start by explaining delayed speculative execution - an important inefficiency - using one sample run. We then analyze this inefficiency over three experiments each with a different number of reducers and reducer waves. We then change the speculative execution algorithm. We use LATE [50] and show the downside of Hadoop's philosophy to not share failure information. Our last experiment shows that DataNode failure can even significantly affect the job preparation stage.

4.2 TaskTracker Failure Analysis

Our first experiment details Hadoop's behavior under TaskTracker process failures. For this first experiment we chose process failures because the presence of RST packets enabled us to perform a more thorough analysis. In the absence of RST packets, CTOs would slow down Hadoop's reaction considerably and would mask the effect of important subtle interactions.

4.2.1 Detailed Analysis

Figure 1 plots job running time against TaskTracker failure injection time. 14 reducers in 1 wave are used in this experiment. Out of 200 runs, 193 are plotted and 7 failed. Note the large variation in job running time. This is due to a large variation in the efficiency of Hadoop's failure detection and recovery mechanisms. To explain, we cluster the results into 8 groups based on the underlying causes. The first 7 groups are depicted in the figure. The 7 failed runs form group G8. The highlights that the reader may want to keep in mind are:

- When the failure affects few reducers, failure detection and recovery is exacerbated.
- Detection and recovery time in Hadoop is unpredictable – an undesirable property in a distributed system. The time it takes reducers to send notifications is variable and so is the time necessary to detect TaskTracker failures.
- The mechanisms used to detect lost map outputs and faulty reducers interact badly. As a result otherwise localized failure propagate in the system. Many reducers die unnecessarily as a result of attempting connections to a failed TaskTracker. This leads to unnecessary re-executions of reducers, thus exacerbating recovery.

Group G1. In G1, at least one map output on the failed TaskTracker was copied by all reducers before the failure. After the failure, the reducer on the failed TaskTracker is speculated on another node and it will be unable to obtain the map outputs located on the failed TaskTracker. According to the penalty computation (§3.1.2) the speculative reducer needs 10 failed connections attempts to the failed TaskTracker (416s in total) before a notification about the lost map outputs can be sent. For this one reducer to send 3 notifications and trigger the re-computation of a map, more than 1200s (i.e. 3 notifications each necessitating 416s) would typically be necessary. The other reducers, even though still running, do not help send notifications because they already copied the lost map outputs. Thus, the TaskTracker timeout (§3.1.1) expires first. Only then are the maps on the failed TaskTracker restarted. This explains the large job running times in G1 and their constancy. G1 shows that the efficiency of failure detection and recovery in Hadoop is impacted when few reducers are affected and map outputs are lost.

Group G2. This group differs from G1 only in that the job running time is further increased by roughly 200s. This is caused by the mechanism Hadoop uses to check for failed TaskTrackers (§3.1.1). To explain, let D be the interval between checks, T_f the time of the failure, T_d the time the failure is detected, T_c the time the last check would be performed if no failures occurred. Also let $n * D$ be the time after which a TaskTracker is declared dead for not sending any heartbeats. For G1, $T_f < T_c$ and therefore $T_d = T_c + n * D$. However, for G2, $T_f > T_c$ and as a result $T_d = T_c + D + n * D$. In Hadoop, by default, $D = 200s$ and $n = 3$. The difference between T_d for the two groups is exactly the 200s that distinguish G2 from G1. In conclusion, the timing of the TaskTracker failure with respect to the JobTracker checks can further increase job running time.

Group G3. In G3, the reducer on the failed TaskTracker is also speculated but sends notifications considerably earlier than the usual 416s. We call these *early notifications*. 3 early notifications are sent and this causes the map outputs to be recomputed before the TaskTracker timeout expires (3.1.2). To explain early notifications consider the simplified example in Figure 2 where the penalty (3.1.2) is linear ($penalty = F_j^R(M)$) and the threshold for sending notifications is 5. A more detailed example is available in [25]. Reducer R needs to copy the output of two maps A and B located on the same node. Case a) shows regular notifications and occurs when connections to the node cannot be established.

Case b) shows early notifications and can be caused by a read error during the copy of A's output. Due to the read error, only $F_j^R(A)$ is initially incremented. This de-synchronization between $F_j^R(A)$ and $F_j^R(B)$ causes the connections to the node to be attempted more frequently. As a result, failure counts increase faster and notifications are sent earlier.

Because the real function for calculating penalties in Hadoop is exponential (§3.1.2), a faster increase in the failure counts translates into large savings in time. As a result of early notifications, runs in G3 finish by as much as 300s faster than the runs in group G1.

Group G4. For G4, the failure occurs after the first map wave but before any of the map outputs from the first map wave is copied by all reducers. With multiple reducers still requiring the lost outputs, the JobTracker receives enough notifications to start the map output re-computation §(3.1.2) before the TaskTracker timeout expires. The trait of the runs in G4 is that not enough early notifications are sent to trigger the re-computation of map outputs early.

Group G5. As opposed to G4, in G5 enough early notifications are sent to trigger map output re-computation earlier.

Group G6. The failure occurs during the first map wave, so no map outputs are lost. The maps on the failed TaskTracker are speculated and this overlaps with subsequent maps waves. As a result, there is no noticeable impact on the job running time.

Group G7. This group contains runs where the TaskTracker was failed after all its tasks finished running correctly. As a result, the job running time is not affected.

Group G8. This group contains the failed jobs. The failed jobs are caused by Hadoop's default behavior to abort a job if one of the job's tasks fails 4 times. A reduce task can fail 4 times because of the induced death problem described next.

4.2.2 *Induced Reducer Death*

In several groups we encounter the problem of induced reducer death. Otherwise localized failures propagate to healthy tasks in the system. This is the case for the reducers which although run on healthy nodes, their death is caused by the repeated failure to connect to the failed TaskTracker. Such a reducer dies (possibly after sending notifications) because a large percent of its shuffles failed, it is stalled for too long and it copied all map output but the failed ones §(3.1.3). We also see reducers die within seconds of their start (without having sent notifications) because the conditions in §(3.1.3) become temporarily true when the failed node is chosen among the first nodes to connect to. In this case most of the shuffles fail and there is little progress made. Induced reducer death wastes time by causing task re-execution and wastes resources since shuffles need to be repeated.

4.2.3 *Effect of Alternative Configurations*

Subsection (§3.1) suggests failure detection is sensitive to the number of reducers. We increase the number of reducers to 56 and the number of reduce slots to 6 per node. Figure 3 shows the results. Considerably fewer runs rely on the expiration of the TaskTracker timeout compared to the 14 reducer case because more reducers means more chances to send enough notifications to trigger map output re-computation before the TaskTracker timeout expires. However, Hadoop still behaves unpredictably. The variation in job running time is more pronounced for 56 reducers because each reducer can behave differently: it can suffer from induced death or send notifications early. With a larger number of reducers these different behaviors yield many different outcomes.

Next, we run two concurrent instances of the 14 reducer job and analyze the effect the second scheduled job has on the running time of the first. Figure 4 shows the results for the first scheduled job compared to the case when it runs alone. Without failures, the first scheduled job finishes after a baseline time of roughly 400s. The increase from 220s to 400s is caused by the contention with the second job. The large variation in running times is still present. The second job does not directly help detect the failure faster because the counters in (§3.1) are defined per job. However, the presence of the second job indirectly influences the first job. Contention causes longer running time and in Hadoop this leads to increased speculation of reducers. A larger percentage of jobs finish around the baseline time because sometimes the reducer on the failed TaskTracker is speculated before the failure and copies the map outputs that will become lost. This increased speculation also leads to more notifications so fewer jobs rely on the TaskTracker timeout expiration. Note also the running times around 850s. These jobs rely on the TaskTracker timeout expiration but suffer from the contention with the second job.

The next experiment mimics the failure of an entire node running a TaskTracker. Results are shown in Figure 5 for the 56 reducer job. The lack of RST packets means every connection attempt is subject to a 180s timeout. There is not enough time for reducers to send notifications so all jobs impacted by failure rely on the TaskTracker timeout expiration in order to continue. Moreover, reducers finish only after all their pending connections finish. If a pending connection is stuck waiting for the 180s timeout to expire, this stalls the whole reducer. This delay can also cause speculation and therefore increased network contention. These factors are responsible for the variation in running time starting with 850s.

4.3 DataNode Failure Analysis - Delayed Speculative Execution

The DataNode experiments simulate the failure of entire compute nodes running DataNodes. Thus, RST packets do not appear. We do not present the effect of DataNode process failures since their impact is low. While a DataNode failure is expected to cause some job running time variation and performance degradation, the speculative execution algorithm should eliminate significant negative effects. Our results show the opposite. As a quick example consider Figure 8. In this experiment, the speculative execution algorithm is largely ineffective after the map phase finishes (80s). The complete results for different number of reducers and reducer waves are plotted in Figures 8, 9, 10 and 11.

4.3.1 *Understanding Delayed Speculative Execution*

To understand the DataNode failure results, we first take a deeper look at the interactions between DataNode failures and the speculative execution algorithm. We show that these interactions can cause a detrimental effect which we deem delayed speculative execution. This consists in one speculation substantially delaying future speculations, or in the extreme case precluding any future speculations. The reason lies with the statistical nature of Hadoop's speculative execution algorithm (§2.3).

To explain delayed speculative execution, consider the sample run in Figure 6 which plots the progress rates of two reducers alongside the value of the left side of equation (1) from (§2.3). We call this left side the *limit*. For this run, 13 reducers are started in total,

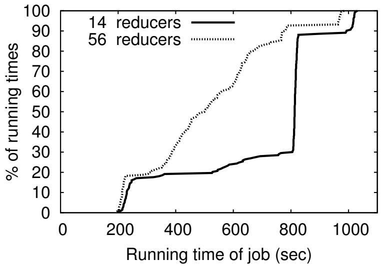

Figure 3: Increased nr of reducers

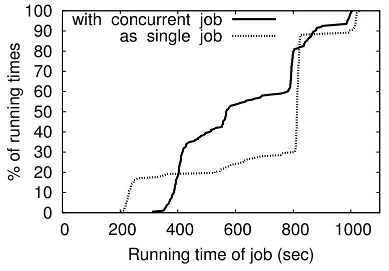

Figure 4: Single vs two concurrent jobs

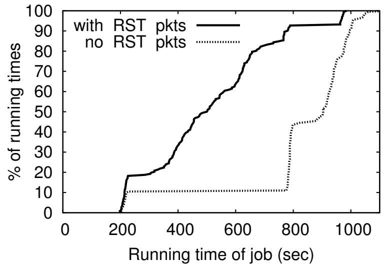

Figure 5: Effect of RST packets

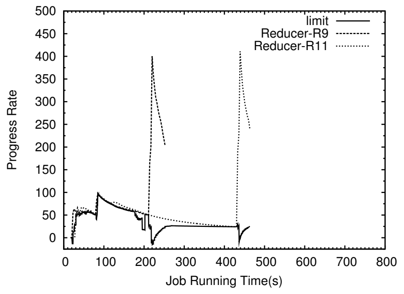

Figure 6: Delayed speculative exec.

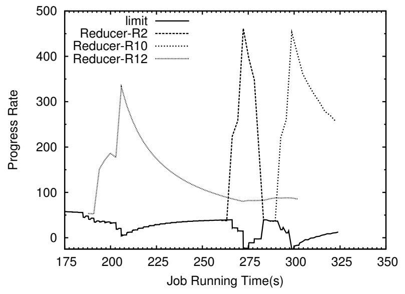

Figure 7: Types of progress rate spikes

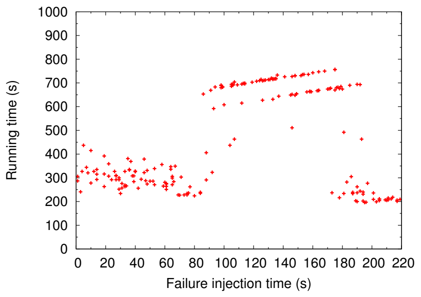

Figure 8: 52 reducers - 1 wave

all in 1 wave. In Hadoop, the progress rate for one task is the maximum rate over the progress rates of the initial task instance and the speculative instance. When an instance of the task completes, the final progress rate for the task is that of the completed instance. Initially, the reducers need to wait for the map phase to end. Their progress rates are close to the average rate and the standard deviation is small. Hence, the limit is relatively high and close to the average rate. A DataNode failure occurs at time 176s and affects reducers R9 and R11. The failure interrupted the write phase of these reducers and therefore R9 and R11 are stuck in a WTO. At time 200s, R9 is speculated. The progress rate for the speculative R9 is very high because it does not need to wait for map outputs to be computed. The outputs are readily available for copying. The sort phase is also fast and this helps further increase the progress rate of speculative R9. In Figure 6 this high rate is visible as a sudden large spike. As a result of the first spike, the average rate increases but the standard deviation increases more. Consequently, the limit decreases. Around 200s, the progress rate for R9 decreases because the speculative R9 needs time to finish the write phase. Because of this progress rate decrease the limit increases but not to the point where it would allow R11 to be speculated. R11 is speculated only around 450s when its progress rate becomes lower than the limit due to the prolong stall in the WTO. In the extreme case, if the limit is lowered too much (can even become negative) then no further speculation may be possible. To continue, all reducers stuck in an WTO would need to wait for the WTO to expire, because they cannot be speculated. In the general case, spikes need not be isolated as in our example. Several reducers can have progress rate spikes at the same time.

The influence that a speculation has on the limit and consequently on the start of subsequent speculations depends on the shape of the spike it creates. We plot these shapes in Figure 7. The ascending part of the spike decreases the limit. The severity of this ascending part depends on the amount of data the reducer needs to shuffle and on the speed of the network transfers. If little data is necessary or network transfers are very fast, then the reducer quickly finishes the shuffle and sort phases with a very high progress rate. The decreasing part of the spike influences how much the limit increases. In our runs we see three distinct decreasing shapes each of which influences the limit differently. A short decrease signals that the write phase proceeded normally (reducer R10). A longer decrease signals that the speculative task also encountered a CTO because of the DataNode failure (reducer R12). A sharp decrease signals that the initial reducer finished shortly after the speculative reducer finished the shuffle and sort phases (reducer R2).

4.3.2 *Effects of Delayed Speculative Execution on the Reduce Phase*

Next, we explain in detail the results for DataNode failures injected during the reduce phase but after the map phase ends at roughly 80s.

For the 52-reducer, 1-wave case in Figure 8 the Hadoop speculative execution algorithm is ineffective after the map phase ends (∼80s). Notice the two parallel clusters of increasing job running time greater than 600s. The high job running times are caused by delayed speculative execution. Due to delayed speculative execution there is usually at least one reducer that cannot be speculated and therefore has to wait for the WTO to expire before continuing. The reason why two clusters exist lies in a Hadoop code-level design choice where a reducer does not remember a failed DataNode if it caused a WTO. Thus, the same failed DataNode can cause the reducer to get stuck in a CTO after the WTO. On the other hand, after a CTO, the reducer remembers the failed DataNode and no further CTOs are caused by that failure. If the WTO occurs at the last block that the reducer needs to write, no CTOs can follow. Therefore, one cluster is formed by reducers suffering only from a WTO while the other cluster is comprised of reducers suffering from both a WTO and a CTO. The steady increase in job running time for each of the clusters is a function of how close to the end of the job the failure was injected.

For the 13-reducer, 1-wave case in Figure 9 the speculative execution algorithm is more effective after the map phase end (∼80s). Large job running times caused by delayed speculative execution

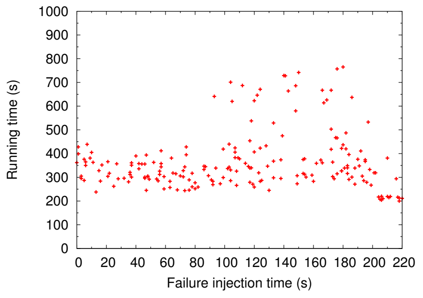

Figure 9: 13 reducers - 1 wave

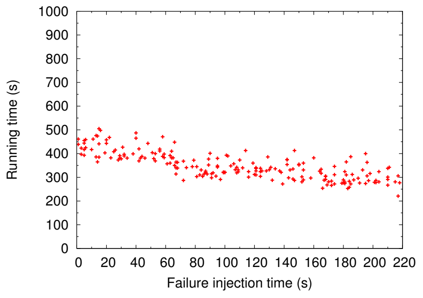

Figure 10: 52 reducers - 4 waves

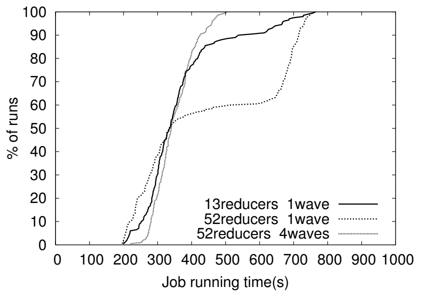

Figure 11: CDF for Figures 8, 9, 10

are still common but faster running times also exist. Compared to the 52-reducer case, each of the 13 reducers is responsible for writing 4 times more blocks and this considerably increases the chance that a CTO affects the write phase of a speculative reducer. As a result of these CTOs the limit is increased more and speculation becomes possible again thus resulting in some faster running times. Moreover, with only 13 reducers, less speculations are necessary overall since fewer reducers are impacted by the failure. Sometimes only 1 or 2 speculations are necessary overall and if both are started at the same time (in the first spike) there is no other speculation to be delayed.

For the 52-reducer, 4-wave case presented in Figure 10 the speculative execution algorithm performs well. The reason is that the reducers in the last 3 waves all have very high progress rates initially since the map outputs are already available and the sort phase is fast. As a result, the limit becomes high and less influenced by subsequent spikes. Consequently, further speculation is not impaired.

4.3.3 *Effects of Speculative Execution on the Map Phase*

We next explain the results for the experiments in Figures 8, 9, 10, and 11 when the failure is injected before 80s. During the first 80s, failures overlap with the map phase and reducers are not yet in the write phase. We did not see cases of delayed speculative execution for the map phase because mappers, unlike reducers, did not have to wait for their input data to be available and the map progress rates were similar. In theory, delayed speculative execution is also possible for the map phase when there is a large variation in progress rates among maps. This can happen in a topology with variable bandwidth. In these cases fast maps could skew the statistics.

Nevertheless, for the map phase we also identified speculative execution inefficiencies under DataNode failures. We encounter needless speculative execution caused by not including in the decision process information about why a task is slow . For example, a map task can stall on a 60s CTO but the speculative execution algorithm speculates a task only after the task has run for at least 60s. The speculation can be needless here because it oftentimes occurs exactly when the CTO expires and the initial map task can continue and quickly finish.

When the failure occurs during the map phase, the job running times are smaller than when reducers are affected. However, job running time variation still exists and is caused by several factors most of which are common to all 3 experiments from Figures 8, 9, 10, and 11. For example, sometimes the NameNode encounters a CTO at the end of a job, when it writes to HDFS a file with details about the run. This delays the delivery of the job results to the user even though all tasks, and therefore the computation are finished. Also, if one of the maps from the last map waves suffers from a CTO this impacts job running time more since the CTO cannot be overlapped with other map waves. The reducers are delayed until the map stuck in the CTO finishes. Specific to the 52-reducer, 4-wave case is the fact that timeouts are possibly encountered by reducers in every of the 4 waves. As a result, job running times are slightly larger for this scenario.

4.4 DataNode Failure Analysis - Not Sharing Failure Information

In this section, we show the effect of Hadoop's philosophy to trade-off sharing of potentially useful information for extreme scalability. The effect is a significant increase in job-running time caused by failures being re-discovered by each task separately.

4.4.1 *Using LATE as an Alternative Algorithm*

We chose LATE as the speculative execution algorithm in this experiment because its goal is to react to under-performing tasks as early as possible. We first look at the 52-reducer, 1-wave case. As suggested in [50] we set the LATE SlowTaskThreshold to the 25th percentile. The results are plotted in Figures 12 and 13.

Overall, LATE performs better than Hadoop's speculative execution algorithm but running times larger than 600s are still present. Because of its more aggressive nature, LATE oftentimes speculates a task before the failure and therefore tasks having both the initial and the speculative instance running before the failure are present. The large job running times in this experiment are the runs in which both the initial task instance and its speculative instance are stuck in a WTO because they are affected by the same DataNode failure. Hadoop does not allow sharing of failure information and therefore the failure is re-discovered individually by each task. The task can continue and finish only after the WTOs expires for one of its instances.

For the 52-reducer 4-wave and the 13-reducer 1-wave cases LATE did not produce large job running times. In these cases, in our experiments the problem described above is still possible but it is less probable since fewer reducers are active at the same time. Note that it is enough for just one task to be affected in the manner described and the whole job's performance is significantly affected.

4.4.2 *Delayed Job Start-up*

We now analyze the effect of DataNode failures on the job start-up time. On each run, we fail one random DataNode at a random time starting 5s before the job submission time and ending 5s after. Job submission time is at 5s. The results are pictured in Figure 14 and 15. Without DataNode failures, the job start-up time is roughly 1s, thus the JobTracker finishes all write operations soon after 6s. This explains why failures occurring after 6s in Figure 14 do not

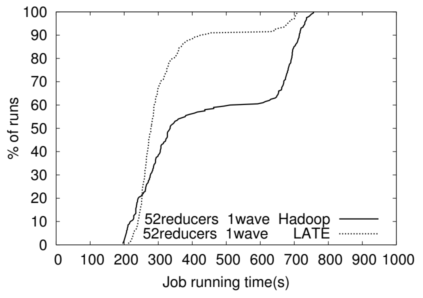

Figure 12: LATE vs Hadoop CDF

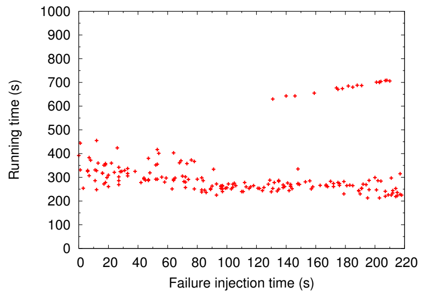

Figure 13: LATE 52 reducers 1-wave

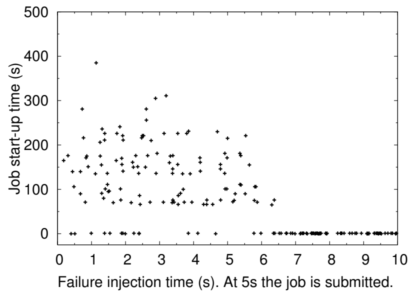

Figure 14: Job start-up times

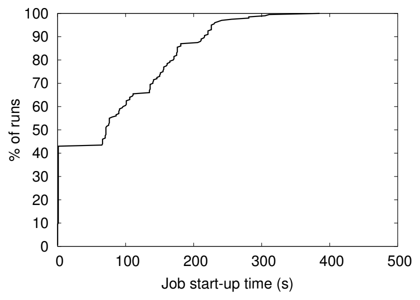

Figure 15: CDF of job startup times

impact job start-up time. However, if the DataNode failure occurs during the first 6s most jobs are impacted.

The reason is that even before a job is started, the JobTracker needs to make multiple HDFS write requests to replicate job-specific files. By default, 6 such files are written: job.jar (java classes for the job), job.split and job.splitmetainfo (description of map inputs), job.xml (job parameter values), jobToken (security permissions) and job.info. Any timeout delaying the writing of these files delays the start of the whole job. However, Hadoop does not share failure information between these 6 write operations. Moreover, the job.jar file is replicated by default using a large replication factor of 10 [49]. This makes this operation even more susceptible to DataNode failures. Unfortunately, this large replication factor is not adaptive and can cause inefficiencies when failures occur in small clusters. In our runs, with 13 total DataNodes and 10 DataNodes required by the large replication factor for job.jar, the chance that the write operation was impacted by the randomly induced failure was high. This explains why only few runs in Figure 14 were unaffected by a failure injected before 6s.

4.4.3 *The Effect on the Map Phase*

Having understood that Hadoop does not share failure information between tasks we can also apply this to the experiments in Figures 8, 9, 10, and 11, for the case when the failure is injected before 80s. Several mappers are influenced by the failure. This is because each map task performs 3 or 5 HDFS block read operations for processing one single input split. The first access is for the job.split file which identifies the input split for the job. The second access reads the input data while the third access reads the start of the subsequent block because a map input split can span HDFS block boundaries. Two more accesses can appear in case the map input split is not at the beginning of the HDFS file. Generally,the more HDFS accesses a task performs the greater the chance a failure will impact the task. Since, Hadoop does not share failure information between mappers and therefore many mappers can encounter a CTO because of the same failure.

5. DISCUSSIONS AND IMPLICATIONS

Delayed speculative execution is a general problem. Delayed speculative execution is a general concern for statistics-based speculative execution algorithms such as Hadoop's. There are many ways to trigger delayed speculative execution and failures are just one of them. The large HDFS timeouts are not a fundamental cause of delayed speculative execution, although they can add to the overhead. Two common conditions are needed to trigger the negative effects of delayed speculative execution: the existence of slow tasks that would benefit from speculation and conditions for tasks to suddenly speed up and create progress rate spikes. Slow tasks have many common causes including failures, timeouts, slow machines or slow network transfers. Progress rate spikes can be caused by varying input data availability (no more waiting is necessary for map outputs after the map phase ends) or by small reducer input data size (small input size means fast progress). Varying network speeds can also cause progress rate spikes. This especially concerns recent proposals for circuit-augmented network topologies [46, 29] that inherently present large variations in bandwidth over different paths.

With performance variation becoming a fact of life in the cloud it would be useful to develop speculative execution algorithms that forgo statistics altogether and instead are centered on a thorough understanding of the computation performed as well as of the current performance of the infrastructure. For example, static analysis of jobs [32, 34] could be used to generate a performance model that can be subsequently leveraged for scheduling and speculative execution decisions.

Not sharing failure information. We have shown the negative effects of not sharing failure information at job runtime. Regardless of how good a speculative execution decision may be before a failure, all benefits can quickly be invalidated when a speculated task is affected by a failure because it needs to read or write data to the failed node. Therefore, good speculative execution decisions are not sufficient. Care must be taken at runtime to ensure the success of the speculative execution decision. Sharing failure information at runtime is one potential approach to ensure this success. The benefits of sharing failure information are not limited to speculated tasks. Different tasks can significantly benefit from sharing failure information because they would not have to individually rediscover a failure.

We believe that sharing need not be limited to failure information. Performance, scalability or straggler information could also potentially be shared not only inside one application but among similar cloud applications [27]. With more information obtained from sharing, a large scale computing framework such as Hadoop would be more likely to make better provisioning and runtime decisions. Nevertheless, Hadoop's reason for not sharing information is warranted. Given the unprecedented scale of today's cloud environment sharing information between tasks, if not carefully done, can quickly become a serious bottleneck. Moreover, the extreme scalability of Hadoop's design is a cornerstone of Hadoop's success [43, 44]. Future work can analyze what is the minimum amount of information that, if shared, can yield maximum benefits. Also, it is useful to analyze the trade-off between the shared information's freshness and the overall gain on the system's performance.

Decoupling failure recovery from overload recovery. TCP connection failures are not only an indication of task failures but also of congestion. However, the two factors require different actions. It makes sense to restart a reducer placed disadvantageously in a network position susceptible to recurring congestion. However, it is inefficient to restart a reducer because it cannot connect to a failed TaskTracker. The data will still be unavailable regardless of whether the reducer is restarted or not. Unfortunately, the news of a connection failure does not by itself help Hadoop distinguish the underlying cause. This overloading of connection failure semantics ultimately leads to a more fragile system as exemplified by the induced reducer death problem.

In the future, it can prove useful to decouple failure recovery from overload recovery entirely. For dealing with compute node load, solutions can leverage the history of a compute node's behavior which has been shown to be a good predictor of transient compute node load over short time scales [15]. For dealing with network congestion, the use of network protocols such as AQM/ECN [30, 40, 26] that expose more information to the applications can be considered.

The need for adaptivity. Timeouts and thresholds in Hadoop are static. The disadvantage of static timeouts is that they cannot correctly handle all situations. Conservative timeouts are useful to cause a task's progress rate to slow down in order to be noticed by the speculative execution algorithm. Conservative timeouts are also useful for protection against temporary network or compute node overload. However, short timeouts may allow fail-over to other DataNodes when data can be read from or written to multiple DataNodes. TaskTracker thresholds are also static and we have shown that this leads to poor performance when few reducers are impacted by a TaskTracker failure.

Future work should consider adaptive timeouts that are set using system wide information about congestion, state of a job and availability of data. As a more general solution it would prove useful to complement Hadoop's design with a dependable failure detection and performance measuring mechanism, that would go beyond timeouts and guesswork - approaches that we have shown to be inadequate today. Hadoop needs to be much more aware of its environment and adapt to performance influencing environmental characteristics: reliability, sharing of resources, use of virtualization, performance variability.

The need for analysis work on large scale computing frameworks. Our paper is the first to provide a thorough analysis of Hadoop's performance under failure conditions. We believe such analysis work is fundamental for improving application performance in cloud environments. There is already a large body of work analyzing the performance of representative cloud infrastructures [41, 33, 48, 31]. We think this should be complemented with analysis work on representative cloud applications especially given their large but still increasing popularity. We hope our paper is an insightful first step to this end.

We have shown that Hadoop's internal mechanisms cause significant and unpredictable performance variations under failures. These results suggest that it is challenging to model Hadoop's performance under failure conditions. Comparing our results against recent work on simulating the performance of Hadoop [47] under failures highlights the difficulty in developing accurate models of Hadoop's behavior based mainly on Hadoop's high-level design specifications. The subtle interactions that lead to performance variations do not appear in the model. Nevertheless such modeling work is very important in the cloud since users need to be able to estimate application performance in order to choose suitable large scale computing frameworks or cloud environments. We believe analysis work such as ours can be leveraged in the development of more advanced models of Hadoop's behavior.

6. RELATED WORK

In the existing literature, smart replication of intermediate data (e.g. map outputs) has been proposed to improve performance under TaskTracker failures [36, 15]. Replication minimizes the need for re-computation of intermediate data and allows for fast failover if one replica cannot be contacted as a result of a failure. Unfortunately, replication may not be always beneficial. It has been shown [36] that replicating intermediate data guards against certain failures at the cost of overhead during periods without failures. Moreover, replication can aggravate the severity of existing hotspots. Therefore, complementing replication with an understanding of failure detection and recovery is equally important. Existing work on leveraging opportunistic environments for large distributed computations [37] can also benefit from this understanding as such environments exhibit behavior that is similar to failures.

A recent study [31] characterizes the effect of network-related failures in the cloud. While the study does not deal with application-level effects, it shows network-related failures have an important effect on the data transfers. Applications built on top of large scale computing frameworks like Hadoop typically rely heavily on data transfers. Our study paints a complementary picture to the effects of network-related failures. We look at compute node failures and specifically target application-level design inefficiencies and interactions.

Other recent studies have found and analyzed significant cloud performance variability [41, 33, 48, 42]. The variability detected by these studies mainly stems from environmental causes such as sharing the data center network, using virtualized environments or leveraging the functionality provided by cloud services. Our work complements these study by identifying and analyzing the significant performance variation caused by the design of a cloud application itself. We showed that this variation can appear even in cloud environments with predictable performance.

Our work is also related to recent efforts for improving the performance of speculative execution algorithms in large scale computing frameworks [50, 15]. There are however important differences. Our work goes beyond speculative execution. This related body of work does not consider failure detection but we do so in detail. These related studies only relate to failure indirectly through outliers which are one possible effect of failures. Instead we analyze the effect of failures directly and exhaustively. We find that failures interact with Hadoop's inner workings in subtle ways (e.g. induced reducer death) and are at odds with Hadoop's design decisions (not sharing any information for scalability reasons). We have even discovered possible improvements to this past body of work. We analyzed the LATE algorithm [50] and showed it can be improved under failures. Moreover, the delayed speculative execution problem we uncovered is a new and important concern for statistical speculative execution algorithms.

7. CONCLUSION

In this paper we exposed and analyzed Hadoop's sluggish, variable and unpredictable performance under compute node failures. We identified several design decisions responsible: delayed speculative execution, the lack of sharing of failure information and the overloading of connection failure semantics. We believe our findings are generally insightful beyond Hadoop and will pave the way for a new class of more advanced large scale computing frameworks that are more predictable and more robust.

8. ACKNOWLEDGEMENTS

This research was sponsored by NSF CAREER Award CNS-0448546, NeTS FIND CNS-0721990, NeTS CNS- 1018807, by an Alfred P. Sloan Research Fellowship, an IBM Faculty Award, and by Microsoft Corp. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the Alfred P. Sloan Foundation, IBM Corp., Microsoft Corp., or the U.S. government.

9. REFERENCES

[1] Apache Mahout. http://mahout.apache.org/.

[2] Cloud9. http://lintool.github.com/Cloud9/.

[3] Cloudera. http://www.cloudera.com/hadoop/.

[4] Contrail. http://sourceforge.net/apps/mediawiki/contrail-bio/index.php?title=Contrail.

[5] Failure Rates in Google Data Centers. http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/.

[6] Hadoop. http://hadoop.apache.org/.

[7] Hadoop Wiki - Powered By. http://wiki.apache.org/hadoop/PoweredBy.

[8] How Rackspace Now Uses MapReduce and Hadoop to Query Terabytes of Data. http://highscalability.com/how-rackspace-now-uses-mapreduce-and-hadoop-query-terabytes-data.

[9] J. Zawodny - Yahoo! Launches World's Largest Hadoop Production Application. http://developer.yahoo.com/blogs/hadoop/posts/2008/02/yahoo-worlds-largest-production-hadoop/.

[10] Microsoft Embraces Elephant of Open Source. http://www.wired.com/wiredenterprise/2011/10/microsoft-and-hadoop/.

[11] Open Cirrus(TM). https://opencirrus.org/.

[12] Pegasus. http://www.cs.cmu.edu/ pegasus/.

[13] X-RIME. http://xrime.sourceforge.net/.

[14] A. Abouzeid, K. Bajda Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. In *VLDB*, 2009.

[15] 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 *OSDI*, 2010.

[16] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, 2010.

[17] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. In *WREN*, 2009.

[18] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. SIGMOD, 2011.

[19] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.

[20] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS*, 2011.

[21] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[22] J. Dean. Experiences with MapReduce, an Abstraction for Large-Scale Computation. In *Keynote I: PACT*, 2006.

[23] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[24] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[25] F. Dinu and T. S. E. Ng. Hadoop's Overload Tolerant Design Exacerbates Failure Detection and Recovery. In *NETDB*, 2011.

[26] F. Dinu and T. S. E. Ng. Inferring a Network Congestion Map with Zero Traffic Overhead. In *ICNP*, 2011.

[27] F. Dinu and T. S. E. Ng. Synergy2Cloud: Introducing Cross-Sharing of Application Experiences Into the Cloud Management Cycle. In *Hot-ICE*, 2012.

[28] J. Dittrich, J-A Q. Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). In *VLDB*, 2010.

[29] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *SIGCOMM*, 2010.

[30] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[31] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*, 2011.

[32] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, and A. Rowstron. Automatic io filtering for optimizing cloud analytics. In *Technical Report no. MSR-TR-2012-3, Microsoft Research*, January 2012. http://research.microsoft.com/apps/pubs/default.aspx?id=157556.

[33] A. Iosup, N. Yigitbasi, and D. Epema. On the Performance Variability of Production Cloud Services. In *CCGrid*, 2011.

[34] E. Jahani, M. J. Cafarella, and C. Re. Automatic optimization for mapreduce programs. In *VLDB*, 2011.

[35] S. Kandula, J. Padhye, and P. Bahl. Flyways to De-Congest Data Center Networks. In *HotNETS*, 2009.

[36] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making Cloud Intermediate Data Fault-Tolerant. In *SOCC*, 2010.

[37] H. Lin, X. Ma, J. Archuleta, W. Feng, M. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic eNvironments. In *HPDC*, 2010.

[38] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[39] P. Hunt and M. Konar and F. P. Junqueira and B. Reeed. Zookeeper: Wait-Free Coordination for Internet-Scale Systems. In *USENIX ATC*, 2010.

[40] K. Ramakrishnan, S. Floyd, and D. Black. *RFC 3168 - The Addition of Explicit Congestion Notification to IP*, 2001.

[41] J. Schad, J. Dittrich, and J-A Q. Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. In *VLDB*, 2010.

[42] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B.Saha. Sharing the Data Center Network. In *NSDI*, 2011.

[43] A. Thusoo, S. Anthony, N. Jain, R. Murthy, Z. Shao, D. Borthakur, J. S. Sarma, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, 2010.

[44] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[45] K. Venkatesh and N. Nagappan. Characterizing Cloud Computing Hardware Reliability. In *SOCC*, 2010.

[46] G. Wang, D. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time Optics in Data Centers. In *SIGCOMM*, 2010.

[47] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A Simulation Approach to Evaluating Design Decisions in MapReduce Setups. In *MASCOTS*, 2009.

[48] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM*, 2010.

[49] T. White. Hadoop: The definitive guide.

[50] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.