MITIGATING CASCADING PERFORMANCE FAILURES AND OUTAGES IN CLOUD SYSTEMS

A PROPOSAL SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
RIZA O. SUMINTO

CHICAGO, ILLINOIS
2018
# TABLE OF CONTENTS

LIST OF FIGURES ................................. iii  
LIST OF TABLES ................................. iv  
ABSTRACT ................................... v  
1 INTRODUCTION ............................... 1  
2 BACKGROUND ................................. 3  
  2.1 Cascading Performance Failure .......... 3  
  2.1.1 Speculative Execution in Hadoop .... 3  
  2.1.2 Degraded Network Devices .......... 5  
  2.1.3 Fault Model ......................... 6  
  2.1.4 Impacts ............................. 6  
  2.2 Cascading Outage Bugs .................. 7  
  2.2.1 Sample Bugs ......................... 8  
3 RESEARCH DETAIL ............................ 9  
  3.1 Path Based Speculative Execution ...... 9  
  3.1.1 SE Loopholes and Flaws ............. 9  
  3.1.2 PBSE Techniques .................... 10  
  3.1.3 Evaluation ......................... 15  
  3.2 Cascading Outage Bug Elimination ...... 18  
  3.2.1 Project Proposal .................... 18  
  3.2.2 Preliminary Work .................... 20  
4 RESEARCH PROGRESS AND PLAN ............. 22  
  4.1 Research Progress ...................... 22  
  4.2 Research Plan ......................... 22  
5 SUMMARY .................................. 23  
REFERENCES .................................. 24
LIST OF FIGURES

2.1 **A Hadoop job and a successful SE.** Figures (a) and (b) are explained in the “Symbols” and “Successful SE” discussions in Section 2.1.1, respectively. .......................... 5

2.2 **Impact of a degraded NIC.** Figure (a) shows the CDF of job duration times of a Facebook workload on a 15-node Hadoop cluster without and with a slow node (With-0-Slow vs. With-1-Slow lines). The slow node has a 1-Mbps degraded NIC. Figure (b) is a replica of Figure 2 in our prior work [42], showing that after several hours, the problem cascades to entire cluster, making cluster throughput drops to 1 job/hour. ........................................ 7

3.1 **PBSE vs. Hadoop (Base) SE.** The figure above shows CDF of latencies of 150 FB2010 jobs running on 15 nodes with one 1-Mbps degraded NIC (1Slow), no degraded NIC (0Slow), and one dead node (1Dead). ................................. 16

3.2 **PBSE speed-ups (vs. Hadoop Base SE) with varying (a) degradation, (b) workload, and (c) cluster size.** The x-axis above represents the percentiles (y-axis) in Figure 3.1 (e.g., “P80” denotes 80th-percentile). For example, the bold (1Mbps) line in Figure 3.2a plots the PBSE speedup at every percentile from Figure 3.1 (i.e., the horizontal difference between the PBSE-1Slow and Base-1Slow lines). As an example point, in Figure 3.1, at 80th percentile (y=0.8), our speed-up is 1.7× (T_{Base}/T_{PBSE} = 54sec/32sec) but in Figure 3.2a, the axis is reversed for readability (at x=P80, PBSE speedup is y=1.7). ................................. 17
LIST OF TABLES

2.1 Symbols. The table above describes the symbols that we use to represent a job topology, as discussed in Section 2.1.1 and illustrated in Figures 2.1.

3.1 COBE patterns. COBE bug patterns found from initial bug study.
ABSTRACT

Modern distributed systems ("cloud systems") have emerged as a dominant backbone for many today’s applications. As these systems collectively become the “cloud operating system”, users expect high dependability including performance stability and availability. Small jitter in system performance or minutes of service downtimes can have huge impact on company and users satisfactory. In this proposal, we are tackling this challenges. We try to improve cloud system dependability by mitigating disruptive cascading effect in the aspect of performance stability and availability.

For performance reliability aspect, we focus on mitigating cascading performance failure by improving tail tolerance of data-parallel framework. One popular solution to reduce tail latency problem in Speculative execution (SE). Existing SE implementations such as in Hadoop and Spark are considered quite robust. However, we found an important source of tail latencies that current SE implementations cannot handle graciously: node-level network throughput degradation. We reveal the loopholes of current SE implementations under this unique fault model, and how the problem can cascade to entire cluster. We then address the problem using PBSE, a robust, path-based speculative execution that employs three key ingredients: path progress, path diversity, and path-straggler detection and speculation.

For availability aspect, we try to improve cloud system availability by detecting and eliminating cascading outage bugs (CO bugs). CO bug is bug that can cause simultaneous or cascades of failures to each of the individual nodes in the system, which eventually leads to a major outage. While hardware arguably is no longer a single point of failure, our large-scale studies of cloud bugs and outages reveal that CO bugs has emerged as new class of outage-causing bugs and single point of failure in the software.

We are proposing Cascading Outage Bugs Elimination (COBE) project. In this project, we will: (1) study the anatomy of CO bugs, (2) develop CO-bug detection tools to unearth CO bugs, and (3) build CO-bug containment solutions to prevent CO bugs from causing an outage in deployment.
CHAPTER 1
INTRODUCTION

Modern distributed systems ("cloud systems") have emerged as a dominant backbone for many today’s applications. They come in different forms such as scale-out systems [43, 55], key-value stores [37, 41], computing frameworks [40, 53], synchronization [36, 48] and cluster management services [47, 61]. As these systems collectively become the “cloud operating system”, users expect high dependability including reliability and availability. They have to provision fast and stable response time, which means they need stable performance; and must be accessible anytime and anywhere, an ideal 24/7 service uptime if possible.

Unfortunately, the complexity of the software and environment in which they must run has outpaced existing testing and debugging tools. As cloud systems must run at scale with different topologies, execute complex distributed protocols, face load fluctuations and a wide range of hardware faults, and serve users with diverse job characteristics, maintaining performance stability has been more challenging than ever. Small jitter in system performance can have huge impact on company and users satisfactory [56]. On the other hand, cloud outages keep happening every year [59, 60, 60], and can easily cripple down a large number of other services [11, 30, 31]. Not only do outages hurt customers, they also cause financial and reputation damages. Minutes of service downtimes can create hundreds of thousands of dollar, if not multi-million, of loss in revenue [18, 19, 22]. Company’s stock can plummet after an outage [28]. Sometimes, refunds must be given to customers as a form of apology [31]. As rivals always seek to capitalize an outage [9], millions of users can switch to another competitor, a company’s worst nightmare [21].

In this proposal, we attempt to improve dependability of cloud-scale distributed systems. We are tackling this challenge by mitigating disruptive cascading effect in the aspect of performance stability and availability.

For performance reliability aspect, we focus on mitigating cascading performance failure by improving tail tolerance of data-parallel framework. One popular solution to reduce tail latency
problem in data-parallel frameworks is *speculative execution (SE)*; with SE, if a task runs slower than other tasks in the same job (a “straggler”), the straggling task will be speculated (via a “backup task”). With a rich literature of SE algorithms [32, 33, 39, 52, 62, 64, 67], existing SE implementations such as in Hadoop and Spark are considered quite robust. However, we found an important source of tail latencies that current SE implementations cannot handle gracefully: *node-level network throughput degradation*. We reveal the *loopholes* of current SE implementations under this unique fault model, and how the problem can cascade to entire cluster. We then address the problem using PBSE, a robust, path-based speculative execution that employs three key ingredients: path progress, path diversity, and path-straggler detection and speculation.

And for availability aspect, we focus on preventing downtimes of datacenter and mobile systems caused by *cascading outage bugs*. “No single point of failure” is the mantra for high availability. Hardware arguably is no longer a single point of failure as the philosophy of redundancies has permeated systems design. On the other hand, software redundancy such as N-version programming is deemed expensive and only adopted in mission-critical software such as in avionics. Thus, in many important systems today, software bugs are single points of failure. Some software bugs are “benign”; they might fail some subcomponents but the whole system can tolerate the partial failure. Some other bugs however can lead to outages such as configuration bugs and state-corrupting concurrency bugs, which have been analyzed extensively in literature. However, our large-scale studies of cloud bugs and outages [44, 45] reveal a new class of outage-causing bugs. In particular, *there are bugs that can cause simultaneous or cascades of failures to each of the individual nodes in the system, which eventually leads to a major outage*. We name them *cascading outage (CO) bugs*.

The proposal is a fusion of our previous work and our on-going work. It includes path-based speculative execution [58] and cascading outage bug elimination.
CHAPTER 2
BACKGROUND

In this proposal, we aim to improve the dependability of the systems in two aspects, performance reliability and availability. Our work focus on unearthing bugs that are related to these two issues. For performance reliability, we focus on eliminating the cascading performance failure caused by node-level network throughput degradation, and for availability, we focus on cascading outage (CO) bugs. This chapter discusses the background of these node-level network throughput degradation and CO bugs, and related work to combat them.

2.1 Cascading Performance Failure

We are focusing on the case of cascading performance failure in data-parallel framework caused by presence of a network-degraded node.

We first describe some background materials (§2.1.1) and present real cases of degraded network devices (§2.1.2) which motivates our unique fault model (§2.1.3). We then highlight the impact of this fault model to Hadoop cluster performance (§2.1.4).

Overall, we found that a network-degraded node is worse than a dead node, as the node can create a cascading performance problem. One network-degraded node can make the performance of the entire cluster collapse (e.g., after several hours the whole-cluster job throughput can drop from hundreds of jobs per hour to 1 job/hour). This cascading effect can happen as unspeculated slow tasks lock up the task slots for a long period of time.

2.1.1 Speculative Execution in Hadoop

In Hadoop 2.0 (Yarn), a node contains task containers/slots. When a job is scheduled, Hadoop creates an Application Manager (AM) and deploys the job’s parallel tasks on allocated containers. Each task sends a periodic progress score to AM (via heartbeat). When a task reads/writes a file,
it asks HDFS namenode to retrieve the file’s datanode locations. Hadoop and HDFS nodes are colocated, thus a task can access data remotely (via NIC) or locally (via disk; aka. “data locality”). A file is composed of 64MB blocks. Each is 3-way replicated.

**Symbols:** Table 2.1 describes the symbols we use to represent a job topology. For example, Figure 2.1a illustrates a Hadoop job reading two input blocks (I₁ and I₂); each input block can have 3 replicas (e.g., I₂, I'₂, I''₂). The job runs 2 map tasks (M₁, M₂); reduce tasks (R₁, R₂) are not shown yet. The first map achieves data locality (I₁→M₁ is local) while the second map reads data remotely (I₂→M₂ is via NICs). A complete job will have three stages: Input→Map (e.g., I₁→M₁), Map→Reduce shuffle (e.g., M₁→R₁, M₁→R₂), and Reduce→Output 3-node write pipeline (e.g., R₂→O₂→O'₂→O''₂).

**Successful SE:** The Hadoop SE algorithm (or “base SE” for short), which is based on LATE [67], runs in the AM of every job. Figure 2.1b depicts a successful SE: I₂’s node has a degraded NIC (bold circle), thus M₂ runs slower than M₁ and is marked as a straggler, then the AM spawns a new speculative/backup task (M'₂) on a new node that coincidentally reads from another fast input

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>Application/Job Manager</td>
</tr>
<tr>
<td>Iᵢ</td>
<td>Node for an HDFS input block to task i</td>
</tr>
<tr>
<td>Iᵢ', Iᵢ''</td>
<td>2nd and 3rd replica nodes of an input block</td>
</tr>
<tr>
<td>Mᵢ</td>
<td>Node for map task i</td>
</tr>
<tr>
<td>Mᵢ'</td>
<td>Node for speculated (') map i</td>
</tr>
<tr>
<td>Rᵢ</td>
<td>Node for reduce task i</td>
</tr>
<tr>
<td>Oᵢ</td>
<td>Node for output block</td>
</tr>
<tr>
<td>Oᵢ', Oᵢ''</td>
<td>2nd and 3rd replica nodes of an output block</td>
</tr>
</tbody>
</table>

Table 2.1: **Symbols.** The table above describes the symbols that we use to represent a job topology, as discussed in Section 2.1.1 and illustrated in Figures 2.1.
replica ($I'_2 \rightarrow M'_2$). For every task, the AM by default limits to only one backup task.

### 2.1.2 Degraded Network Devices

Beyond fail-stop, network devices can exhibit “unexpected” forms of failures. Below, we re-tell the real cases of limping network devices in the field [1–8, 42, 44, 45, 49].

In many cases, NIC cards exhibit a high-degree of packet loss (from 10% up to 40%), which then causes spikes of TCP retries, dropping throughput by orders of magnitude. An unexpected auto-negotiation between a NIC and a TOR switch reduced the bandwidth between them (an auto-configuration issue). A clogged air filter in a switch fan caused overheating, and subsequently heavy re-transmission (e.g., 10% packet loss). Some optical transceivers collapsed from Gbps to Kbps rate (but only in one direction). A non-deterministic Linux driver bug degraded a Gbps NIC’s performance to Kbps rate. Worn-out cables reportedly can also drop network performance. A worn-out Fibre Channel Pass-through module in a high-end server blade added 200-3000 ms delay.

As an additional note, we also attempted to find (or perform) large-scale statistical studies on this problem but to no avail. As alluded elsewhere, stories of “unexpected” failures are unfortunately “only passed by operators over beers” [34]. For performance-degraded devices, one of the issues is that, most hardware vendors do not log performance faults at such a low level (unlike hard errors [35]). Some companies log low-level performance metrics but aggregate the results (e.g., hourly disk average latency [46]), preventing a detailed study. Thus, the problem of performance-
degraded devices is still under studied and requires further investigation.

### 2.1.3 Fault Model

Given the cases above, our fault model is a severe network bandwidth degradation experienced by one or more machines. For example, the bandwidth of a NIC can drop to low Mbps or Kbps level, which can be caused by many hardware and software faults such as bit errors, extreme packet loss, overheating, clogged air filters, defects, buggy auto-negotiations, and buggy firmware and drivers, as discussed above.

A severe node-level bandwidth degradation can also happen in public multi-tenant clouds where extreme outliers are occasionally observed [54, Fig. 1]. For instance, if all the tenants of a 32-core machine run network intensive processes, each process might only observe $\sim 30$ Mbps, given a 1GBps NIC. With a higher-bandwidth 10-100GBps NIC and future 1000-core processors [29, 65], the same problem will apply. Furthermore, over-allocation of VMs more than the available CPUs can reduce the obtained bandwidth by each VM due to heavy context switching [63]. Such problem of “uneven congestion” across datanodes is relatively common [20].

### 2.1.4 Impacts

**Slow tasks are not speculated:** Under the fault model above, Hadoop SE fails to speculate slow tasks. Figure 2.2a shows the CDF of job duration times of a Facebook workload running on a 15-node Hadoop cluster without and with one 1-Mbps slow node (With-0-Slow vs. With-1-Slow nodes). The 1-Mbps slow node represents a degraded NIC. As shown, in a healthy cluster, all jobs finish in less than 3 minutes. But with a slow-NIC node, many tasks are not speculated and cannot escape the degraded NIC, resulting in long job tail latencies, with 10% of the jobs ($y=0.9$) finishing more than 1 hour.

**“One degraded device to slow them all:”** Figure 2.2b shows the impact of a slow NIC to the entire cluster over time. Without a slow NIC, the cluster’s throughput (#jobs finished) increases
Figure 2.2: **Impact of a degraded NIC.** Figure (a) shows the CDF of job duration times of a Facebook workload on a 15-node Hadoop cluster without and with a slow node (With-0-Slow vs. With-1-Slow lines). The slow node has a 1-Mbps degraded NIC. Figure (b) is a replica of Figure 2 in our prior work [42], showing that after several hours, the problem cascades to entire cluster, making cluster throughput drops to 1 job/hour.

steadily (around 172 jobs/hour). But with a slow NIC, after about 4 hours \((x=250\text{min})\) the cluster throughput collapses to 1 job/hour.

The two figures show that existing speculative execution fails to cut tail latencies induced by our fault model.

### 2.2 Cascading Outage Bugs

“No single point of failure” is the mantra for high availability. Hardware arguably is no longer a single point of failure as the philosophy of redundancies has permeated systems design. On the other hand, software redundancy such as N-version programming is deemed expensive and only adopted in mission-critical software such as in avionics. Thus, in many important systems today, software bugs are single points of failure.

Some software bugs are “benign”; they might fail some subcomponents but the whole system might tolerate the partial failure. Some other bugs however can lead to outages, bugs such as state-corrupting configuration and concurrency bugs, which have been analyzed extensively in literature. However, our large-scale studies of cloud bugs and outages [44, 45] reveal a new class of outage-causing bugs. Specifically, **there are bugs that can cause simultaneous or cascades of failures to**
each of the individual nodes in the system, which eventually leads to a major outage. We name them cascading outage (CO) bugs.

To tackle this new class of bugs, we are proposing Cascading Outage Bug Elimination (COBE) project. In this COBE project, we will: (1) study the anatomy of CO bugs, (2) develop CO-bug detection tools, and (3) build CO-bug containment strategies.

2.2.1 Sample Bugs

We now present three real cases of CO bugs that motivate COBE project.

1. **HBase**: HBase can run tens to thousands of region servers, each is responsible for a set of table regions. A case of CO bug, HB-9737 [16], surfaced when a region server crashed due to a bad handling of corrupt region files. In HBase, if a server dies, another live server will be picked to handle the orphan region files. This means that the picked server will execute the same buggy code (the bad handling of corrupt files), thus crashing the new server too. As the same failover logic repeats, all the nodes become unavailable.

2. **DynamoDB**: A similar CO bug caused an outage of Amazon DynamoDB in September 2015 [10, 25]. Here, many storage servers attempted to refresh their membership data to a busy metadata service. As the storage servers observe unexpected timeouts, they disqualify themselves from accepting requests, eventually causing major unavailability.

3. **Safari**: CO bugs not only affect datacenter systems, but also client/mobile systems. Early in 2016, “Apple’s [iPhone/iPad] Safari browser crashed worldwide” [15, 23], caused by a communication bug between the browsers auto-search-suggest component and the backend server (similar to the DynamoDB communication bug between the storage servers and the metadata service).

The cases above reveal single points of failure in hidden dependencies; there is a single root failure (e.g., file corruption, unavailable service) that eventually affects the entire system.
CHAPTER 3
RESEARCH DETAIL

3.1 Path Based Speculative Execution

To understand the impact of node-level network throughput degradation, we tested Hadoop [13] as well as other systems including Spark [66], Flume [12], and S4 [14], on a cluster of machines with one slow-NIC node.

In the following section, we will discuss current SE loopholes and flaws that we discover (§3.1.1), our techniques to solve the problem (§3.1.2), and we evaluate our performance compared to current SE (§3.1.3).

3.1.1 SE Loopholes and Flaws

From our test run, we discovered that many tasks transfer data through the slow-NIC node but cannot escape from it, resulting in long tail latencies. Our analysis uncovered many surprising loopholes in existing SE implementations, which we bucket into two categories: the “no” straggler problem, where all tasks of a job involve the slow-NIC node (since all tasks are slow, there is “no” straggler detected) and the straggling backup problem, where the backup task involves the slow-NIC node again (hence, both of the original and the backup tasks are straggling at the same time).

Hadoop is a decade-old mature software and many SE algorithms are derived from Hadoop / MapReduce [40, 67]. Thus, we believe there are some fundamental flaws that lead to the existence of SE loopholes. We believe there are two flaws:

(1) Node-level network degradation is not incorporated as a fault model. Yet, such failures occur in production. This fault model is different than “node contentions” where CPU and local storage are also contended (in which cases, the base SE is sufficient). In our model, only the NIC is degraded, not CPU nor storage.
(2) Task ≠ Path. When the fault model above is not incorporated, the concept of path is not considered. Fatally, path progresses of a task are lumped into one progress score, yet a task can observe differing path progresses. Due to lack of path information, slow paths are hidden. Worse, Hadoop can blame the straggling task even though the culprit is another node.

### 3.1.2 PBSE Techniques

To address this problem, we present solution to the problem, *path based speculative execution* (PBSE), which contain three important ingredients: path progress, path diversity, and path-straggler detection and speculation.

We now present the three important elements of PBSE: path progress, path diversity, and path-straggler detection and speculation. The next sections details our design in the context of Hadoop/HDFS stack with 3-way replication and 1 slow NIC ($F=1$)\(^1\)

**Path Progress**

The heart of PBSE is the exposure of path progresses to the SE algorithm. A path progress $P$ is a tuple of \{Src, Dst, Bytes, $T$, BW\}, sent by tasks to the job’s manager (AM); $Bytes$ denotes the amount of bytes transferred within the elapsed time $T$ and $BW$ denotes the path bandwidth (derived from $Bytes/T$) between the source-destination (Src, Dst) pair. Path progresses are piggybacked along with existing task heartbeats to the AM. In PBSE, tasks expose to AM the following paths:

- **Input→Map (I→m):** In Hadoop/ HDFS stack, this is typically a one-to-one path (e.g., I$_2$→M$_2$) as a map task usually reads one 64/128-MB block. Inputs of multiple blocks are usually split to multiple map tasks.

- **Map→Reduce (m→R):** This is typically an all-to-all shuffling communication between a set of

\(^1\) $F$ denotes the tolerable number of failures.
map and reduce tasks; many-to-many or one-to-one communication is possible, depending on the user-defined jobs and data content. The AM now can compare the path progress of every $m\to R$ path in the shuffle stage.

- **Reduce→Output (R→O):** Unlike earlier paths above, an output path is a pipeline of sub-paths (e.g., $R_1\to O_1\to O'_1\to O''_1$). A single slow node in the pipeline will become a downstream bottleneck. To allow fine-grained detection, we expose the individual sub-path progresses. For example, if $R_1\to O_1$ is fast, but $O_1\to O'_1$ and $O'_1\to O''_1$ are slow, $O'_1$ can be the culprit.

The key to our implementation is a more information exposure from the storage (HDFS) to compute (Hadoop) layers. Without more transparency, important information about paths is hidden. Fortunately, the concept of transparency in Hadoop/HDFS already exists (e.g., data locality exposure), hence the feasibility of our extension. The core responsibility of HDFS does not change (i.e., read/write files); it now simply exports more information to support more SE intelligence in the Hadoop layer.

**Path Diversity**

Straggler detection is only effective if independent progresses are comparable. However, patterns such as $X\to M_1$ and $X\to M_2$ with X as the tail-SPOF is possible, in which case potential stragglers are undetectable. To address this, *path diversity* prevents a potential tail-SPOF by enforcing independent, comparable paths. While the idea is simple, the challenge lies in efficiently removing potential input-SPOF, map-SPOF, reduce-SPOF, and output-SPOF in every MapReduce stage:

(a) **No input-SPOF in I→m paths:** It is possible that map tasks on different nodes read inputs from the same node ($I_1\to M_1$, $I_2\to M_2$, and $I_1=I_2$).\(^2\) To enforce path diversity, map tasks must ask HDFS to diversify input nodes, at least to two ($F+1$) source nodes.

Here, we take the reactive approach. We let map tasks run independently in parallel, but when

\(^2\) A=B implies A and B are in the same node.
map tasks send their first heartbeats to the AM, they report their input nodes. If the AM detects a potential input-SPOF, it will reactively inform one (as $F=1$) of the tasks to ask HDFS namenode to re-pick another input node (e.g., $I_2' \rightarrow M_2$ and $I_2' \neq I_1$). After the switch ($I_2$ to $I_2'$), the task continues reading from the last read offset (no restart overhead).

(b) No map-SPOF in $I \rightarrow m$ and $m \rightarrow R$ paths: It is possible that map tasks are assigned to the same node ($I_1 \rightarrow M_1$, $I_2 \rightarrow M_2$, $M_1 = M_2$, and $M_1/M_2$’s node is a potential tail-SPOF); note that Hadoop only disallows a backup and the original tasks to run in the same node (e.g., $M_1 \neq M_1'$, $M_2 \neq M_2'$). Thus, to prevent one map-SPOF ($F=1$), we enforce at least two nodes ($F+1$) chosen for all the map tasks of a job.

As of the implementation, when a job manager (AM) requests $C$ containers from the resource manager (RM), the AM also supplies the rule. RM will then return $C-1$ containers to the AM first, which is important so that most tasks can start. For the last container, if the rule is not satisfied and no other node is currently available, RM must wait. To prevent starvation, if other tasks already finish half way, RM can break the rule.

(c) No reduce-SPOF in $m \rightarrow R$ and $R \rightarrow O$ paths: In a similar way, we enforce each job to have reducers at least in two different nodes. Since the number of reducers is defined by the user, not the runtime, the only way to prevent a potential reduce-SPOF is by cloning the single reducer. This is reasonable as a single reducer implies a small job and cloning small tasks is not costly [32].

Through a few design iterations, we modify the reduce stage to pre-allocate write pipelines during shuffling and keep re-picking until all the pipelines are free from an output-SPOF. In vanilla Hadoop, write pipelines are created after shuffling (after reducers are ready to write the output). Contrary, in our design, when shuffling finishes, the no-SPOF write pipelines are ready to use.

Finally, we would like to note that by default PBSE will follow the original task placement (including data locality) from the Hadoop scheduler and the original input source selection from HDFS. Only in rare conditions will PBSE break data locality. For example, let us suppose $I_1 \rightarrow M_1$...
and I_2 \rightarrow M_2 achieve data locality and both data transfers happen in the same node. PBSE will try to move M_2 to another node (the “no map-SPOF” rule) ideally to one of the two other nodes that contain I_2’s replicas (I’_2 or I”_2). But if the nodes of I’_2 and I”_2 do not have a free container, then M_2 must be placed somewhere else and will read its input (I_2/I’_2/I”_2) remotely.

Path-Based Detection and Speculation

As path diversity ensures no potential tail-SPOF, we then can compare paths, detect path-stragglers, and pinpoint the faulty node/NIC. Similar to base SE, PBSE detection algorithm is per-job (in AM) and runs for every MapReduce stage (input, shuffle, output). As an important note, PBSE runs side by side with the base SE; the latter handles task stragglers, while PBSE handles path stragglers. PBSE detection algorithm runs in three phases:

(1) **Detecting path stragglers:** In every MapReduce stage, AM collects a set of paths (S_P) and labels P as a potential straggling path if its BW is less than $\beta \times$ the average bandwidth, where $0 \leq \beta \leq 1.0$ (configurable). The straggling path will be speculated only if its estimated end time is longer than the estimated path replacement time (plus the standard deviation). If a straggling path (e.g., A→B) is to be speculated, we execute the following phases below, in order to pinpoint which node (A or B) is the culprit.

(2) **Detecting the slow-NIC node with failure groups:** We categorize every P in S_P into failure/risk groups [68]. A failure group $G_N$ is created for every source/destination node N in S_P. If a path P involves a node N, P is put in $G_N$. For example, path A→B will be in $G_A$ and $G_B$ groups. In every group G, we take the total bandwidth. If there is a group whose bandwidth is smaller than $\beta \times$ the average of all group bandwidths, then a slow-NIC node is detected.

(3) **Detecting the slow-NIC node with heuristics:** Failure groups work effectively in cases with many paths (e.g., many-to-many communications). In some cases, not enough paths exist to pinpoint the culprit. For example, given only a fast A→B and a straggling C→D, we cannot pinpoint
the faulty node (C or D). Fortunately, given three factors (path diversity rules, the nature of MapReduce stages, and existing rules in Hadoop SE), we can employ the following effective heuristics:

(a) **Greedy approach:** Let’s consider a fast $I_1 \rightarrow M_1$ and a straggling $I_2 \rightarrow M_2$; the latter must be speculated, but the fault could be in $I_2$ or $M_2$. Fortunately, Hadoop SE by default prohibits $M_2'$ to run on the same node as $M_2$. Thus, we could speculate with $I_2 \rightarrow M_2'$. However, we take a greedy approach where we speculate a completely new pair $I_2' \rightarrow M_2'$ (avoiding both $I_2$ and $M_2$ nodes). To implement this, when Hadoop spawns a task ($M_2'$), it can provide a blacklisted input source ($I_2$) to HDFS.

(b) **Deduction approach:** While the greedy approach works well in the Input→Map stage, other stages need to employ a deduction approach. Let’s consider a one-to-one shuffling phase (a fast $M_1 \rightarrow R_1$ and a slow $M_2 \rightarrow R_2$). By deduction, since $M_2$ already “passes the check” in the $I \rightarrow M_2$ stage (it was not detected as a slow-NIC node), then the culprit is likely to be $R_2$. Thus, $M_2 \rightarrow R_2'$ backup path will start. Compared to deduction approach, employing a greedy approach in shuffling stage is more expensive (e.g., speculating $M_2' \rightarrow R_2'$ requires spawning $M_2'$).

(c) **Dynamic retries:** Using the same example above (slow $M_2 \rightarrow R_2$), caution must be taken if $M_2$ reads locally. That is, if $I \rightarrow M_2$ only involves a local transfer, $M_2$ is not yet proven to be fault-free. In this case, blaming $M_2$ or $R_2$ is only 50/50-chance correct. In such a case, we initially do not blame the map side because speculating $M_2$ with $M_2'$ is more expensive. We instead take the less expensive gamble; we first speculate the reducer (with $R_2'$), but if $M_2 \rightarrow R_2'$ path is also slow, we perform a second retry ($M_2' \rightarrow R_2'$). Put simply, sometimes it can take one or two retries to pinpoint one faulty node. We call this dynamic retries, which is different than the limited-retry in base SE (default of 1).

The above examples only cover $I \rightarrow m$ and $m \rightarrow R$ stages, but the techniques are also adopted for $R \rightarrow O$ stage.
3.1.3 Evaluation

We implemented PBSE in Hadoop/HDFS v2.7.1 in 6003 LOC (3270 in AM, 1351 in Task Management, and 1382 in HDFS). We now evaluate our implementation.

**Setup:** We use Emulab nodes [17], each running a (dual-thread) 2×8-core Intel Xeon CPU E5-2630v3 @ 2.40GHz with 64GB DRAM and 1Gbps NIC. We use 15-60 nodes, 12 task slots per Hadoop node (the other 4 core for HDFS), and 64MB HDFS block size.\(^4\) We set \(\beta=0.1\), a non-aggressive path speculation.

**Slowdown injection:** We use Linux tc to delay one NIC to 60, 30, 10, 1, and 0.1 Mbps; 60-30 Mbps represent a contended NIC with 16-32 other network-intensive tenants and 10-0.1 Mbps represent a realistic degraded NIC; real cases of 10%-40% packet loss were observed in production systems (§2.1.2), which translate to 2 Mbps to 0.1 Mbps NIC (as throughput exhibits exponential-decaying pattern with respect to packet loss rate).

**Workload:** We use real-world production workloads from Facebook (FB2009 and FB2010) and Cloudera (CC-b and CC-e) [38]. In each, we pick a sequence of 150 jobs\(^5\) with the lowest inter-arrival time (i.e., a busy cluster). We use SWIM to replay and rescale the traces properly to our cluster sizes as instructed [24].

**Metrics:** We use two primary metrics: job duration \((T)\) and speed-up \((=T_{Base}/T_{PBSE})\).

Figure 3.1 shows the CDF of latencies of 150 jobs from FB2010 on 15 nodes with five different setups from right (worse) to left (better): Base Hadoop SE with one 1Mbps slow NIC (BaseSE-1Slow), PBSE with the same slow NIC (PBSE-1Slow), PBSE without any bad NIC (PBSE-0Slow), Base SE without any bad NIC (BaseSE-0Slow), and Base SE with one dead node (BaseSE-1Dead).

We make the following observations from Figure 3.1. First, as alluded in §3.1.1, Hadoop SE cannot escape tail-SPOF caused by the degraded NIC, resulting in long job tail latencies with

---

\(^4\) Today, HDFS default block size is 128 MB, which actually will show better PBSE results because of the longer data transfer. We use 64 MB to be consistent with all of our initial experiments.

\(^5\) 150 jobs are chosen so that every normal run takes about 15 minutes; this is because the experiments with severe delay injections (e.g., 1 Mbps) can run for hours for base Hadoop. Longer runs are possible but will prevent us from completing many experiments.
Figure 3.1: **PBSE vs. Hadoop (Base) SE.** The figure above shows CDF of latencies of 150 FB2010 jobs running on 15 nodes with one 1-Mbps degraded NIC (1Slow), no degraded NIC (0Slow), and one dead node (1Dead).

The longest job finishing after 6004 seconds (BaseSE-1Slow line). Second, PBSE is much more effective than Hadoop SE; it successfully cuts tail latencies induced by degraded NIC (PBSE-1Slow vs. BaseSE-1Slow). Third, PBSE cannot reach the “perfect” scenario (BaseSE-0Slow). Fourth, with Hadoop SE, a slow NIC is worse than a dead node (BaseSE-1Slow vs. BaseSE-1Dead); put simply, Hadoop is robust against fail-stop failures but not degraded network. Finally, in the normal case, PBSE does not exhibit any overhead; the resulting job latencies in PBSE and Hadoop SE under no failure are similar (PBSE-0Slow vs. Base-0Slow).

We now perform further experiments by varying the degraded NIC bandwidth (Figure 3.2a), workload (3.2b), and cluster size (3.2c). To compress the resulting figures, we will only show the speedup of PBSE over Hadoop SE (a more readable metric), as explained in the figure caption.

**Varying NIC degradation:** Figure 3.2a shows PBSE speed-ups when we vary the NIC bandwidth of the slow node to 60, 30, 10, 1, and 0.1 Mbps (the FB2010 and 15-node setups are kept the same). We make two observations from this figure. First, PBSE has higher speed-ups at higher percentiles. In Hadoop SE, if a large job is “locked” by a tail-SPOF, the job’s duration becomes extremely long. PBSE on the other hand can quickly detect and failover from the straggling paths. With a 60Mbps congested NIC, PBSE delivers some speed-ups (1.5-1.7×) above P98. With a more congested NIC (30 Mbps), PBSE benefits start to become apparent, showing 1.5-2× speed-
ups above P90. Second, PBSE speed-up increases (2-70×) when the NIC degradation is more severe (e.g., the speedups under 1 Mbps are relatively higher than 10 Mbps). However, under a very severe NIC degradation (0.1 Mbps), our speed-up is still positive but slightly reduced. The reason is that at 0.1 Mbps, the degraded node becomes highly congested, causing timeouts and triggering fail-stop failover. Again, in Hadoop SE, a dead node is better than a slow NIC (Figure 3.1). The dangerous point is when a degraded NIC slows down at a rate that does not trigger any timeout.

**Varying workload and cluster size:** Figure 3.2b shows PBSE speed-ups when we vary the workloads: FB2009, FB2010, CC-b, CC-e (1Mbps injection and 15-node setups are kept the same). As shown, PBSE works well in many different workloads. Finally, Figure 3.2c shows PBSE speed-ups when we vary the cluster size: 15 to 60 nodes (1Mbps injection and FB2010 setups are kept the same). The figure shows that regardless of the cluster size, a degraded NIC can affect many jobs. The larger the cluster size, tail-SPOF probability is reduced but still appear at a significant rate (> P90).
3.2 Cascading Outage Bug Elimination

Next we move on to cascading outage bugs (CO bugs). Our view of the CO bugs in §2.2.1 is that in all of the cases, the complete outage can be alleviated if we can (a) understand the bug pattern, (b) detect the potential cascading/simultaneous impacts, and (c) allow the system to continue in degraded mode. For example, in the HBase case, some regions can be made unavailable while most of other regions (potentially hundreds of thousands of them) are still servable. In the DynamoDB case, the storage nodes can perhaps continue in read-only mode temporarily as opposed to rejecting all requests. In the Safari case, the browser can continue to run with auto-search-suggest disabled.

3.2.1 Project Proposal

We are proposing COBE project, which comprise of these following three phases.

Phase 1: Study of CO bugs anatomy

Establishing the anatomy of special bugs is a crucial step in eliminating them, which have done successfully in the past for concurrency [51], distributed [49], performance [57], and scalability bugs [50]. Thus, our first step is to understand the patterns of CO bugs.

To give a flavor of our initial findings, below we describe some CO bug patterns we have observed.

1. Repeated buggy logic after failover: Here, a failover repeats the same buggy logic in other nodes (e.g., the HBase case above). We also observed the same pattern in another HBase bug but the root failure is logcleaning exception.

2. Positive feedback loop: This is the case where some nodes declared dead/busy due to request overload, but the recovery introduces more load (e.g., a re-mirroring storm), which causes more nodes to be marked dead/busy, which then causes more recovery. This is similar to the first pattern but is more about busy nodes as opposed to crashing.
3. Simultaneous, deterministic crash paths on non-critical service errors: Here, there is a crash-leading path executed in all nodes but the root error was caused by non-critical services/components (e.g., the DynamoDB and Safari cases).

4. Distributed deadlock: Here, each node is waiting for other nodes to progress (similar to the classical deadlock problem in multi-threaded process). For example, a Cassandra coordination bug caused all nodes to keep gossiping during bootstrapping but never entering a normal state.

Phase 2: CO Detection (Offline)

Based on the patterns we establish, our next step is to create CO bug detection tools. For example, to detect pattern (1), a program analysis needs to connect crash path and recovery path and checks whether the recovery path may lead to the same crash path. To detect pattern (3), we need to perform path backward-slicing from crash points to the root causes and categorizes whether they are supposedly tolerable or can be replaced with a degraded mode. To reduce the number of crash points to analyze, we will devise some heuristics or leverage post-mortem failure diagnosis. Overall, we will explore and build various CO detection approaches.

This phase only unearths the possibility of CO bugs but is not enough to isolate them in practice, which leads us to the next phase, CO containment.

Phase 3: CO Containment (Online)

In this phase, we will add to software systems the capability of containing the cascading nature of CO bugs in live deployment. Below are some of the containment principles we will develop and integrate to our target systems.

First, software systems must distinguish hardware and software failures. When a “node” is dead, it is typically caused by one of them (not both). CO bugs tend to kill some machines grad-
ually and leave some “trails” \textit{(e.g.,} exception logs, core dumps). If the same trail appears in all nodes, the system should be suspicious of the existence of CO bugs. Second, upon detection of CO bug, the system can go in degraded mode (rather than continue and eventually shut down the entire system). For example, the three sample cases above are containable. The challenge is to develop domain specific containment strategies \textit{(e.g.,} stop the failover, move to read-only mode, skip auto-suggest feature). Third, as we introduce degrade mode, other components must be degrade-tolerant. Today’s systems typically only accept two modes: work or fail. However, a degraded mode can cause unintended side effects to other layers \textit{(e.g.,} skipping a buggy load balancer may cause unintended backlogs in some nodes), which must be taken care properly. Overall, we will formulate and address the challenges of building CO containment.

3.2.2 Preliminary Work

In the phase of CO bug study, we have begin our classification of CO bugs anatomy. Table 3.1 show some patterns of CO bugs that we find so far, analyzed from issue repositories hosted under Apache Software Foundation Projects [26].

In respect of offline CO detection, we start building two static analysis tools based on WALA [27]. The first tools we build will perform backward-slicing from crash points to the root causes, and will come up with set of variable constraints that should be satisfied for the crash to happen. We are planning to add symbolic execution as the next pipeline to verify if there is a possible combination of variable values that satisfy the constraints.

Our observation on some of the CO bugs shows commonalities on the crash path, such that there are remote exception passed from one node to another nodes through RPC calls \textit{(e.g.,} in master-slave architecture, exception thrown from master node and caught by slave node). Under this observation, we build the second static analysis tool to detect crash path potential caused by remote exception. We define crash path as exception flow between \textit{exception point}, the earliest point in the code that may throw exception, up to \textit{shutdown point}, the exception handler that catch
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Count</th>
<th>Bug ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated buggy recovery</td>
<td>6</td>
<td>HB-9737, HB-3664, HD-9178, HD-8995, HD-8960, HD-6937</td>
</tr>
<tr>
<td>Master node shutdown</td>
<td>5</td>
<td>HD-8258, HD-7725, HD-7414, HD-6908, YR-4459</td>
</tr>
<tr>
<td>Master node deadlock</td>
<td>4</td>
<td>HB-16138, HD-11817, HD-9294, HD-4816</td>
</tr>
<tr>
<td>Timeout</td>
<td>4</td>
<td>HD-9293, HD-9107, HD-8676, HD-6545</td>
</tr>
<tr>
<td>Hanging recovery</td>
<td>3</td>
<td>CA-13918, HD-10609, HD-6696</td>
</tr>
<tr>
<td>Partial thread shutdown</td>
<td>2</td>
<td>HD-9287, SP-22083</td>
</tr>
<tr>
<td>Topology specific bug</td>
<td>2</td>
<td>HB-7709, HD-10320</td>
</tr>
<tr>
<td>Total</td>
<td>26</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: **COBE patterns.** *COBE bug patterns found from initial bug study.*

the exception and initiate system shutdown (*e.g.*, `System.exit()` in Java). Specifically, our tool will trace existence of crash path that the exception point and shutdown point belong to to different node. Our preliminary test over HDFS codes was able to catch at least 3 crash path between NameNode - DataNode, and NameNode - SecondaryNameNode. We expect more crash path will be revealed as we optimize the tool and inspect different target systems.
CHAPTER 4
RESEARCH PROGRESS AND PLAN

4.1 Research Progress

<table>
<thead>
<tr>
<th>Cascading Performance Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Based Speculative Execution (PBSE)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cascading Outage Bugs Elimination (COBE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is our ongoing work. Our target system for this project include <em>distributed systems</em> (e.g., Hadoop, HBase, Cassandra, and Spark) and <em>mobile software systems</em> (e.g., Google Chromium browser).</td>
</tr>
</tbody>
</table>

4.2 Research Plan

To fulfill the dissertation research, we are working on our last piece, COBE project. This is a plan of action we proposed.

1. Study CO bugs anatomy from online issue repositories, 3 months, 3/1/2018 - 6/1/2018

2. Develop static analysis tools for offline CO detection, 2 month, 6/1/2018 - 8/1/2018

3. Design online strategy for CO containment, 1 month, 8/1/2018 - 9/1/2018

4. Implement CO containment protocol on target systems, 2 month, 9/1/2018 - 11/1/2018

5. Evaluate COBE implementation, 1 month, 11/1/2018 - 12/1/2018

6. Write the dissertation, 1 month, 12/1/2018 - 1/1/2019

7. Defend the dissertation
CHAPTER 5
SUMMARY

In this proposal, we aim to mitigating disruptive cascading effect in cloud-scale distributed systems that disturb performance and availability of the systems. For performance aspect, we focus in improving tail tolerance of data-parallel framework. For availability aspect, we focus in eliminating cascading outage bugs.

In the aspect of performance stability, we found an important source of tail latencies that current Speculative execution (SE) implementations cannot handle graciously: node-level network throughput degradation. We reveal the loopholes of current SE implementations under this unique fault model, and how the problem can cascade to entire cluster. We address the problem using PBSE, a robust, path-based speculative execution that employs three key ingredients: path progress, path diversity, and path-straggler detection and speculation.

In the aspect of availability, we are focusing on detecting and eliminating cascading outage bugs (CO bugs). We believe that the system code itself has emerge as a new single point of failure, which lead to occurrence of CO bugs. To solve CO bugs problem, we are proposing Cascading Outage Bug Elimination (COBE) project, which comprise of three phases: (i) Study of CO bugs anatomy, to better understand the patterns of CO bugs. (ii) Offline CO detection, creating analysis tools to detect potential CO bugs in the code. (iii) Online CO containment, adding to software system the capability of containing the cascading nature of CO bugs in live deployment.
REFERENCES

[1] Personal Communication from datacenter operators of University of Chicago IT Services.
[6] Personal Communication from H. Birali Runesha (Director of Research Computing Center, University of Chicago).

[21] Messaging app Telegram added 5m new users the day after WhatsApp outage (link), February 24, 2014.

[22] RIM lost $54 million on four-day global BlackBerry outage (link), March 20, 2012.


[28] Twitter Stock (TWTR) Dropped $1 During Site Outage (link), March 11, 2014.


