Dedication Text
Epigraph Text
# TABLE OF CONTENTS

LIST OF FIGURES .................................................. vii

LIST OF TABLES .................................................... ix

ABSTRACT .......................................................... x

1 INTRODUCTION .................................................... 1

2 MOTIVATION ....................................................... 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 PBSE: A ROBUST PATH-BASED SPECULATIVE EXECUTION FOR DEGRADED-NETWORK TAIL TOLERANCE IN DATA-PARALLEL FRAMEWORKS .......... 18
   3.1 SE Loopholes .................................................. 18
      3.1.1 No Straggler Detected ....................................... 19
      3.1.2 Straggling Backup Tasks .................................... 20
      3.1.3 The Cascading Impact ..................................... 21
      3.1.4 The Flaws ................................................ 22
   3.2 PBSE .......................................................... 23
      3.2.1 Paths ..................................................... 23
      3.2.2 Path Diversity .............................................. 24
      3.2.3 Detection and Speculation .................................. 27
   3.3 Evaluation ..................................................... 29
      3.3.1 PBSE vs. Hadoop (Base) SE .................................. 30
      3.3.2 Detailed Analysis .......................................... 32
      3.3.3 PBSE vs. Other Strategies ................................ 34
      3.3.4 PBSE with Multiple Failures ............................... 36
      3.3.5 PBSE on Heterogeneous Resources .......................... 36
      3.3.6 Limitations ................................................ 37
   3.4 Beyond Hadoop and HDFS ........................................ 37
   3.5 Related Work ................................................ 39
   3.6 Conclusion ................................................... 41
## 4 COBE: CASCADING OUTAGE BUG ELIMINATION

4.1 Design Principles

4.2 COBE

4.3 Implementation
  4.3.1 Program Facts Extraction
  4.3.2 Datalog-based Analysis
  4.3.3 Race in Master Analysis
  4.3.4 Transient Network Error Analysis

4.4 Evaluation
  4.4.1 Methodology
  4.4.2 Bug detection result
  4.4.3 Performance result

4.5 Related Work

4.6 Conclusion

## 5 OTHER WORKS

5.1 Why Do the Clouds Stop? Lessons from Hundreds of High-Profile Outages

5.2 MittOS: Operating System Supports for Millisecond Tail Tolerance in Data-Parallel Storage

5.3 Rivulet: Fault-Tolerant Platform for Smart-Home Applications

5.4 Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems

5.5 ScaleCheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems

5.6 FlyMC: Highly Scalable Testing for Complex Interleavings in Cloud Systems

## 6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusion
  6.1.1 Cascading Performance Failure
  6.1.2 Cascading Outage Bugs

6.2 Future Work
  6.2.1 PBSE
  6.2.2 COBE

REFERENCES
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>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.</td>
</tr>
<tr>
<td>2.2</td>
<td>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 [65], showing that after several hours, the problem cascades to entire cluster, making cluster throughput drops to 1 job/hour.</td>
</tr>
<tr>
<td>2.3</td>
<td>Transient network error in hd8995.</td>
</tr>
<tr>
<td>3.1</td>
<td>Tail-SPOF and “No” Stragglers. Figures (a)-(c) are described in Sections 3.1.1a-c, respectively. $I_1/I_2$ in Figure (a), $M_2$ in (b), and $O_1/O_2$ in (c) are a tail-SPOF that makes all affected tasks slow at the same time, hence “no” straggler. Please see Figure 2.1 for legend description.</td>
</tr>
<tr>
<td>3.2</td>
<td>SE Algorithm “Bug”. The two figures above are explained in Section 3.1.1d.</td>
</tr>
<tr>
<td>3.3</td>
<td>Tail-SPOF and Straggling Backups. Figures (a)-(c) are discussed in Sections 3.1.2a-c, respectively. $I_2$ in Figure (a), $I_2/M_2'$ in (b), and $M_2$ in (c) are a tail-SPOF; the slow NIC is coincidentally involved again in the backup task. Please see Figure 2.1 for legend description.</td>
</tr>
<tr>
<td>3.4</td>
<td>Distribution of job sizes and inter-arrival times. The left figure shows CDF of the number of (map) tasks per job within the chosen 150 jobs from each of the production traces. The number of reduce tasks is mostly 1 in all the jobs. The right figure shows the CDF of job inter-arrival times.</td>
</tr>
<tr>
<td>3.5</td>
<td>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).</td>
</tr>
<tr>
<td>3.6</td>
<td>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 3.5 (e.g., “P80” denotes 80th percentile). For example, the bold (1Mbps) line in Figure 3.6a plots the PBSE speedup at every percentile from Figure 3.5 (i.e., the horizontal difference between the PBSE-1Slow and Base-1Slow lines). As an example point, in Figure 3.5, at 80th percentile (y=0.8), our speed-up is 1.7× ($T_{Base}/T_{PBSE} = 54sec/32sec$) but in Figure 3.6a, the axis is reversed for readability (at x=P80, PBSE speedup is y=1.7).</td>
</tr>
<tr>
<td>3.7</td>
<td>Residual sources of tail latencies (§3.3.2).</td>
</tr>
<tr>
<td>3.8</td>
<td>PBSE vs. other strategies (§3.3.3).</td>
</tr>
<tr>
<td>3.9</td>
<td>PBSE speed-ups with multiple failures and heterogeneous network (§3.3.4 and §3.3.5).</td>
</tr>
</tbody>
</table>
3.10 **Beyond Hadoop/HDFS (§3.4).** The figure shows latencies of microbenchmarks running on four different systems (Hadoop/QFS, Spark, Flume, and S4) with three different setups: baseline without degraded NIC (Base-0Slow), baseline with one 1Mbps degraded NIC (Base-1Slow), and with initial PBSE integration (PBSE-1Slow). Baseline (Base) implies the vanilla versions. The Hadoop/QFS microbenchmark and topology is shown in Figure 3.11c. The Spark microbenchmark is a 2-stage, 4-task, all-to-all communication as similarly depicted in Figure 3.1b. The Flume and S4 microbenchmarks have the same topology. We did not integrate PBSE to S4 as its development is discontinued.

3.11 **Rack slowdown and Erasure-Coded (EC) storage (§3.4).** For simplicity, the figure shows RS(2,1), a Reed Solomon where an input file \(I\) is striped across two chunks \((I_a, I_b)\) with one parity \((I_p)\) with 64KB stripe size (see Figure 2 in [105] for more detail).

4.1 **Cobe Analysis Stack.** The figure above show the high level idea of COBE analysis framework.

4.2 **Cobe Architecture.** Architecture of COBE framework.

4.3 **Example of Harness Code.** The listing above show example of COBE harness used to analyze hb16367.

4.4 **Dependency graph among COBE’s Datalog rules.** The figure above show activated rules and their dependencies across COBE analyses stack. The dashed box represent a Datalog program file that contain specific rules or program facts. The arrows coming out from the box means one or more rules in the Datalog program depends on rule pointed by the arrow head.

4.5 **High level rule for race in master CO pattern.** The bug report \((s,t)\) pair maps into \(((t1,m1,b1),(t2,m2,b2))\) from relation raceInMaster.

4.6 **High level rule for transient network error CO pattern.**

4.7 **Minimal static HB graph to reveal hb4539.** Both yellow and cyan node is a memory access to the same location. The yellow nodes can reach failure instructions while the cyan node is not. The two yellow nodes with bold red border is the true positive case reported by the original issue.
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. 4

2.2 CO bugs patterns. CO bugs patterns found from bug study. 8

3.1 Activated PBSE features (§3.3.2). The table shows how many times each feature is activated, by task and job counts, in the PBSE-1Slow experiment in Figure 3.5. 33

4.1 Program facts domain. This table lists domain of individual attribute in program facts relations that extracted by COBE parser. Domains can have type number (integer) or symbol (string). For clarity, we define domain Num and Sym to represent generic number and symbol data. 46

4.2 Examples of extracted program facts. Each attribute has their own domain (see Table 4.1). Subscript number signify attributes that correlated with each other. For example, (Cm1,Cb1) together represent a basic block number Cb within method Cm. 47

4.3 Benchmark bugs. 58

4.4 Evaluation result. The table show result statistics for each bug benchmark. All true positive cases were found by COBE. For.hd8995 query time, the Datalog program was run in compiled mode, while the others was run in interpreter mode. 59

4.5 Related Work (COBE). The table categorizes works that relate to failure analysis, race analysis, and Datalog in the space of program analysis grouped by either targeting specific system or distributed system. 64
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 dissertation, 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 address CO bugs problem with Cascading Outage Bugs Elimination (COBE) project. In this project, we: (1) study the anatomy of CO bugs, (2) develop CO-bug detection tools to unearth CO bugs.
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 [68, 104], key-value stores [59, 64], computing frameworks [63, 100], synchronization [56, 81] and cluster management services [76, 119]. 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 [108]. On the other hand, cloud outages keep happening every year [116, 117, 117], and can easily cripple down a large number of other services [2, 38, 39]. 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 [11, 12, 29]. Company’s stock can plummet after an outage [36]. Sometimes, refunds must be given to customers as a form of apology [39]. As rivals always seek to capitalize an outage [1], millions of users can switch to another competitor, a company’s worst nightmare [17].

In this dissertation, 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 [45, 48, 62, 98, 120, 127, 133], 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[113], 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 [69, 70] 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*. 

2
CHAPTER 2
MOTIVATION

In this dissertation, we aim to improve the dependability of the systems in two aspects, performance reliability and availability. Our work focuses 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,
Symbols

<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>

A sample of a job topology (2 maps, 2 reduces):

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map phase</td>
<td>I₁→M₁, I₂→M₂</td>
</tr>
<tr>
<td>Shuffle phase</td>
<td>M₁→R₁, M₁→R₂, M₂→R₁, M₂→R₂</td>
</tr>
<tr>
<td>Reduce phase</td>
<td>R₁→O₁→O₁', O₂→O₂→O₂'</td>
</tr>
<tr>
<td>Speculated M₁</td>
<td>I₁'→M₁'</td>
</tr>
<tr>
<td>Speculated R₂</td>
<td>M₁→R₂', M₂→R₂'</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.

...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, I₂', I₂''). 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→O₂'→O₂'').

**Successful SE:** The Hadoop SE algorithm (or “base SE” for short), which is based on LATE [133], 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 (M₂') on a new node that coincidentally reads from another fast input...
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 [19–26, 65, 69, 70, 85].

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” [51]. 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 [52]). Some companies log low-level performance metrics but aggregate the results (e.g., hourly disk average latency [74]), 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 [102, 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 [37, 129], 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 [126]. Such problem of “uneven congestion” across datanodes is relatively common [14].

### 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 [65], 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 the literature. However, our large-scale studies of cloud bugs and outages [69, 70] reveal a new class of outage-causing bugs. Specifically, there are bugs that can cause simultaneous or cascades of

![Figure 2.2: Impact of a degraded NIC.](image-url)
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Count</th>
<th>Bug ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Race in Master</td>
<td>23</td>
<td>hb19218, hb16367, hb14536, hb12958, hb9773, hb8519, hb4729, hb4539,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hd7725, hd7707, hd7225, hd6908, hd6289, hd5474, hd5428, hd5425,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hd5283, tb260, tb258, tb246, tb106, tb89, tb58</td>
</tr>
<tr>
<td>Hanging recovery</td>
<td>11</td>
<td>ca13918, hb21344, hb16138, hb14621, hb13802, hb9721, hb5918, hb3664,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hd4816, tb254, tb29</td>
</tr>
<tr>
<td>Repeated buggy recovery</td>
<td>7</td>
<td>hb14598, hb11776, hb9737, hb7515, hd9178, tb259, tb247</td>
</tr>
<tr>
<td>External library exception</td>
<td>7</td>
<td>hb17522, hb15322, hb14247, hd10609, tb301, tb291, tb275</td>
</tr>
<tr>
<td>Silent heartbeat</td>
<td>4</td>
<td>hd9293, hd9107, hd8676, hd6179</td>
</tr>
<tr>
<td>Transient network error</td>
<td>3</td>
<td>hb10272, hd8995, tb181</td>
</tr>
<tr>
<td>Race in Worker</td>
<td>3</td>
<td>hb20403, tb298, tb52</td>
</tr>
<tr>
<td>Authentication bug</td>
<td>3</td>
<td>kd2264, tb245, tb131</td>
</tr>
<tr>
<td>Abnormal message</td>
<td>3</td>
<td>hb10312, hd5483, tb287</td>
</tr>
<tr>
<td>Topology specific bug</td>
<td>2</td>
<td>hb7709, hd10320</td>
</tr>
<tr>
<td>Cascading Worker lockup</td>
<td>2</td>
<td>hb11813, hd7489</td>
</tr>
<tr>
<td>Total</td>
<td>68</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: **CO bugs patterns.** CO bugs patterns found from bug study.

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 study the anatomy of CO bugs and develop tools to detect CO-bug patterns.

### 2.2.1 Sample Bugs

We start COBE project by collecting samples of CO bugs from publicly accessible issue repositories of open-source distributed systems. We initially focused our search in HDFS[13] and HBASE[6] system. Later on, we also add some CO bugs from Yarn[119], Cassandra[3], Kudu[7], and some non-public issues reported by our company partner. Figure 2.2 shows the list of CO bugs that we found in our study, grouped by their CO pattern. In the following subsections, we will explain each of CO pattern and some issues that fall into that CO patterns category.
Race in Master

The most frequent CO bugs pattern that we found is a race condition that is happening in the master node. We refer to this pattern as *race in master*. This pattern is especially prevalent in systems with Master-Worker architecture. Although it shares similarities with traditional *local concurrency (LC) bugs*, *race in master* pattern differs in the subject of the race condition, that it the message delivery timing[85]. Incoming messages to a master node can incur an *order violation* or *atomicity violation*.

*hb4539* is an example of *race in master* pattern that caused by *order violation* of message. HBASE is using ZooKeeper as a synchronization service between *HBASE Master (HMaster)* and *RegionServers*, worker nodes in HBASE. When an HBASE region R1 is transitioning from OPENING to OPENED state, HMaster will spawn an OpenedRegionHandler thread to delete the ZooKeeper node representing R1. But before this thread is executed, the RegionServer RS1 hosting R1 is down, triggering HMaster to reassign R1 to different RegionServer RS2. When this reassignment is finishing, a second OpenedRegionHandler thread will then spawned and compete with the first OpenedRegionHandler thread to delete the ZooKeeper node of R1. The losing thread will catch an exception for trying to delete an already deleted ZooKeeper node, and in turn, crash HMaster. This bug will not happen if the first handler thread is done executing before reassignment begin.

Another example of *race in master* pattern is *hb4729*, caused by *atomicity violation* of events. In the event of a region splitting, HMaster will first unassign the region by creating an ephemeral ZooKeeper node for that region. In the middle of this splitting process, there is an incoming admin command to alter the region, that is also triggering region unassignment for that alter purpose. The second unassignment then try to re-create an already existing ZooKeeper node and caught a NodeExistsException, which in turn trigger HMaster crash.

Master node crash often leads to a cluster-wide outage. This is why many Master-Worker architecture systems are equipped with high availability features to prevent master node becoming
a single point of failures, such as having a backup master node to failover or do a new master election among active worker nodes. But since they share the same master node code, they are prone to hit the same bug again. Worse case, the failover might not happen at all because the original master is hanging instead of failing gracefully (hb12958, hb14536), the backup master also failed because of inconsistent state (hd6289, hb8519), or the race silently cause corruption that leads to outage in future (hd5425, hd5428, hd6908).

**Hanging recovery**

In this CO pattern, the system is aware of ongoing failure and attempt to do the recovery. However, the recovery procedure is buggy and causes the cluster to hang, unable to service requests. These recovery bugs may happen due to the flaw in the recovery logic itself, or due to another interleaving events that happen concurrently with the recovery process.

In hb21344, an HBASE cluster is hanging after RegionServer hosting the META region crashed. HMaster tries to recover by reassigning the META region to other RegionServer. The recovery steps involve marking the META region’s ZooKeeper node as OPENING and sending openRegion RPC to newly assigned RegionServer. Unfortunately, the openRegion RPC is also failing. After 10 times retry, the recovery procedure is rolled back, but the ZooKeeper node state is not reverted to OFFLINE. HBASE cluster was hanging with an assumption that META region is still in the OPENING state, even when HMaster is restarted.

Similarly, hd9721 also involves a META region recovery procedure. In the middle of recovery, the target RegionServer for META reassignment is restarting, causing it to change its name (RegionServer name components contains its start time). Because of this mismatch, the RegionServer unable to update the ZooKeeper node state from OPENING to OPENED, and the cluster is hanging waiting for META.
Repeated buggy recovery

The recovery procedure in a distributed system typically works by moving the workloads of the failing node to another active node. However, if the workload is the trigger for the node to fail in the first place, the failover node will most likely hit the same failure as well. It may be because of the workload itself that is corrupted, or the same buggy code that is being run. As the same failover logic repeats, all of the nodes then becomes unavailable. We refer to this kind of CO pattern as *repeated buggy recovery*.

An example of this CO pattern is hb9737. A corrupt HFile makes its way into a region and the region becomes offline. HMaster notice the region becomes offline and try to assign it to different RegionServer. Upon reading the corrupt HFile, an exception is thrown and that RegionServer drops the region again. When this exception occurs, there is a bug that causes the RegionServer not to close the filesystem stream used to read the corrupt HFile. As the failover logic repeats, the region keeps bouncing between multiple RegionServer, accumulates orphaned filesystem stream, and one-by-one crashing with OutOfMemoryError.

In hb14598, a table scan operation against region containing particularly wide rows will cause RegionServer to crash with OutOfMemoryError. This lead to *cascading region server death*, as the RegionServer hosting the region died, opened on a new server, the client retried the scan, and the new RegionServer died as well.

External library exception

Distributed systems often build in layers of subsystems. For example, HBASE was built on top of HDFS as the storage layer, and incorporate ZooKeeper for internode synchronization. One system needs to interact with other systems in the different layers through the client API library. Consequently, an error or failure that is happening in the subsystem may propagate to the main system and lead to an outage. We refer to this CO pattern as *external library exception* pattern.

One of CO bug that falls into this pattern is hb14247. Each of HBASE RegionServers is writing
their Write Ahead Log (WAL) file into HDFS. After some time, HBASE archived them into a single HDFS directory. In big clusters, because of long time-to-live of WAL or disabled replications, the number of files under WALs archive directory reaches the max-directory-items limit of HDFS (1048576 items), HDFS client library throws an exception and crash the HBASE cluster. A simple solution for this bug is to separate the old WALs into different directories according to the server name of the WAL.

In tb301, a Hadoop cluster with Azure Data Lake Storage (ADLS) backend is experiencing service outages due to an outdated ADLS SDK library being used. The problem only solved after upgrading to a newer version containing updated ADLS SDK library.

Silent heartbeat

In many distributed systems, nodes availability is determined by periodic signaling between nodes, referred to as heartbeat. If a heartbeat signal from an endpoint is not heard after a certain period, that endpoint is deemed as unavailable by the system. Silent heartbeat is a CO pattern where an outage happens because of a missing or delayed heartbeat signal in either sender or receiver, causing the system to falsely interpret the sender node as unavailable.

An example of silent heartbeat caused by the sender node is hd8676. A rolling upgrade of HDFS DataNodes involves cleaning up data blocks in trash directories. This cleanup is done synchronously by the same thread that is doing heartbeat signaling. In a big busy cluster where the deletion rate is also high, a lot of data blocks can pile up in the DataNode trash directories. Hence, this cleaning process blocks the heartbeat and causes heartbeat expiration. HDFS NameNode losing hundreds of DataNodes after delayed upgrade finalization. The fix for this bug is to make the deletion of trash directories as asynchronous.

In hd9293, silent heartbeat is happening due to delay on the receiver side. In HDFS cluster, DataNodes send heartbeat signals to both Active NameNode and Standby NameNode in a serial manner. An edit log processing by Standby NameNode holds its FSNamesystem lock for too
Figure 2.3: **Transient network error in hd8995.**

long, causing a delay in processing incoming heartbeat. Active NameNode starts removing stale DataNodes which can not send a heartbeat to Active NameNode because they are stuck waiting for a response from Standby NameNode. A similar CO bug caused by slow Standby NameNode is also happening in hd6179.

Transient network error

Distributed systems that highly available often come with a protocol to recover from network partition. When the network partition is resolved, nodes that previously unreachable is expected to automatically sync up with the rest of the cluster. However, we found some cases where the system fails to recover from a network partition, especially if the network partition is happening intermittently. We refer to this kind of CO pattern as **transient network error** pattern.

Figure 2.3 illustrates a **transient network error** pattern in hd8995. When DataNode gets partitioned from NameNode for more than the heartbeat expiration time, DataNode is expected to re-register again with NameNode. Datanodes keep retrying the last RPC call and when it finally gets through, the NameNode will tell it to re-register. The DataNode is supposed to create a registration object which contains address 0.0.0.0, pass it to the NameNode which updates the address and returns it, then the DataNode saves the updated registration object for future calls. The problem is the DataNode saves off the initial registration object containing 0.0.0.0 before it receives the
NameNode response. Intermittent network error happens right in this registration process triggers an exception and left DataNode with invalid registration object containing 0.0.0.0. When the network connectivity restored, the next call to NameNode using this invalid registration object will raise UnregisteredException that in turn will signal DataNode to terminate.

Another example of *transient network error* pattern is hb10272. HBase client caches a connection failure to a server and any subsequent attempt to connect to the server throws a FailedServerException. If a node which hosted both of the active HMaster and ROOT/META table goes offline, the newly anointed HMaster’s initial attempt to connect to the dead RegionServer will fail with NoRouteToHostException which it handles. But on the second attempt, it crashes with FailedServerException. Each of the backup masters will crash with the same error and restarting them will have the same effect. Once this happens, the cluster will remain non-operational until the node with region server is brought online, or the Zookeeper node containing the root region server and/or META entry from the ROOT table is deleted.

**Race in Worker**

In contrast with *race in master, race in worker* pattern is outage that is caused by race condition happening in the worker nodes. While we believe that message delivery timings can also contribute to *race in worker*, all bugs that we found so far usually stem from the use of a non-thread-safe library.

HBASE had this CO pattern in hb20403. HBASE RegionServer sometimes prefetches HFile to improve performance. The prefetching is done by multiple concurrent prefetch threads over a single input stream. Most of the time, the underlying input stream (such as DFSInputStream) is thread-safe, or has a reliable fall back mechanism in case race condition is happening. However, if the file is encrypted, CryptoInputStream will be used instead, and it is not meant to be thread-safe.

In tb52, HBase inside an HDFS Encryption Zone causes Cluster Failure under Load. HBase cannot run safely within HDFS encryption zones because of different concurrency assumptions in
the HBase write-ahead log and HDFS encrypting output streams.

Authentication bug

Authentication bug pattern is a CO pattern where cluster nodes fail to communicate with each other due to authentication issues between them. In high-security setup, an additional authentication layer usually added into the distributed system. Nodes need to authenticate with each other before start communicating by exchanging identity ticket/certificate, often with the help of a trusted third-party service such as Kerberos [16]. These authentication certificates need to be updated periodically to ensure security. Failure to keep node certificates up to date will lead to an authentication error, causing the cluster nodes unable to communicate with each other.

One example of service outage due to an authentication bug is kd2264. A bug in Apache KUDU client causes them to never re-read an updated ticket, even if their underlying ticket cache on disk has been updated with a new credential. Other services that query data from KUDU become unable to query after 30 days since the last ticket read.

In tb245, Apache Impala with TLS enabled may fail to start after upgrade. Patch for a security issue included in the new version caused a mismatch in the domain name of the certificate, which expects a fully qualified domain name (FQDN), versus the hostname used to connect.

Topology specific bug

Topology specific bug pattern is a CO pattern where an outage only happens in a specific network topology. hb7709 and hd10320 both fall into this CO pattern.

In hb7709, two HBASE clusters A and B are set with Master-Master replication. In this mode, replication is sent across in both the directions, for different or same tables, i.e., both of the clusters are acting both as master and slave. A third cluster C is misconfigured to replicate to cluster A. This cause all edits originating from C will be bouncing between A and B forever. In the long run, this infinite ping-pong saturates cluster-wide network traffic.
In 

hd10320, a CO bug can surface if there are rack failures that end up leaving only one rack available. HDFS default block placement policy seeks to put a block replica on a different rack. But if there is only one rack available, the replication thread can get an InvalidTopologyException, which then propagated up and terminate the NameNode.

Abnormal message

Cascading outage can also happen to distributed systems due to the handling of a message that is corrupt or out of ordinary. We refer to this CO pattern as abnormal message pattern.

An example of abnormal message pattern is hd5483. NameNode expects a DataNode to only hold a single copy of any particular data node. However, there was a case where a single DataNode reporting two replicas of the same data block on two different storages. The DataNode has both storages mounted, one storage is mounted as read-write and the other storage is mounted as read-only. Because one DataNode reporting more than one replica of the same block, NameNode failed an assertion and crashed.

In tb287, Apache Sentry may crash Hive Metastore (HMS) due to abnormal notification messages. Sentry expect notifications from HMS to: have no gaps; be monotonically increasing; not have duplicates. If these assumptions are broken Sentry would be very conservative and request several full snapshots around 5 or more times per day. Full Snapshots are resource intensive and can take 10 or more minutes. Sentry also blocks HMS threads when it performs sync operation. If a full snapshot is being triggered, many of these sync operations timeout and leads to HMS crashing.

Cascading Worker lockup

Cascading worker lockup pattern is a CO pattern where an error or blocking event happening in a single worker node is cascading to another worker nodes, making the cluster to hang indefinitely. Unlike race in master or hanging recovery pattern that usually involve master node, cascading
worker lockup happens entirely between worker nodes.

In **hd7489**, many DataNodes hanging their heartbeat and requests from clients. An exception from one DataNode propagates to other DataNodes, which in turn trigger those DataNodes to run disk error checking. The disk error checking holds big locks that block them from heart beating or serving client requests. Other DataNodes attempting to a hung DataNode can hang as well, causing a cascading failure.

In **hb11813**, an unchecked recursion-terminating condition cause a RegionServer to lose all of its RPC handler threads due to stack overflow. This RegionServer becomes unable to respond to any request. Other RegionServers that try to communicate with this hanging RegionServer will also hang indefinitely, which in turn brings down the entire cluster. Stopping the first hanging RegionServer unblocks the cluster and everything comes back to normal.
CHAPTER 3

PBSE: A ROBUST PATH-BASED SPECULATIVE EXECUTION FOR DEGRADED-NETWORK TAIL TOLERANCE IN DATA-PARALLEL FRAMEWORKS

In Section 2.1, we present our case of cascading performance failure in data-parallel framework caused by presence of a network-degraded node. Overall, we found that a network-degraded node is worse than a dead node, as the node can create a cascading performance problem.

In this chapter, we will discuss about PBSE, a robust, path-based speculative execution for degraded-network tail tolerance in data-parallel frameworks. We will discuss about Hadoop SE loopholes (§3.1), PBSE design and evaluation (§3.2-3.3), further integrations (§3.4), related work and conclusion (§3.5-3.6).

3.1 SE Loopholes

This section presents the many cases of failed speculative execution (i.e., “SE loopholes”). For simplicity of discussion, we only inject one degraded NIC.

**Benchmarking:** To test Hadoop SE robustness to the fault model above, we ran real-world production traces on 15-60 nodes with one slow NIC (more details in §3.3). To uncover SE loopholes, we collect all task topologies where the job latencies are significantly longer than the expected latency (as if the jobs run on a healthy cluster without any slow NIC). We ran long experiments, more than 850 hours, because every run leads to non-deterministic task placements that could reveal new loopholes.

**SE Loopholes:** An SE loophole is a unique topology where a job cannot escape from the slow NIC. That is, the job’s latency follows the rate of the degraded bandwidth. In such a topology, the slow NIC becomes a single point of tail-latency failure (a “tail-SPOF”). By showing tail-SPOF, we can reveal not just the straggling tasks, but also the straggling paths. Below we describe some
of the representative loopholes we found (all have been confirmed by Hadoop developers). For each, we use a minimum topology for simplicity of illustration. The loopholes are categorized into no-straggler-detected (§3.1.1) and straggling-backup (§3.1.2) problems.

We note that our prior work only reported three “limplock” topologies [65, §5.1.2]) from only four simple microbenchmarks. In this subsequent work, hundreds of hours of deployment allow us to debug more job topologies and uncover more loopholes.

3.1.1 No Straggler Detected

Hadoop SE is only triggered when at least one task is straggling. We discovered several topologies where all tasks (of a job) are slow, hence “no” straggler.

(a) Same slow input source: Figure 3.1a shows two map tasks (M₁, M₂) reading data remotely. Coincidentally (due to HDFS’s selection randomness when locality is not met), both tasks retrieve their input blocks (I₁, I₂) from the same slow-NIC node. Because all the tasks are slow, there is “no” straggler. Ideally, a notion of “path diversity” should be enforced to ensure that the tail-SPOF (I₁/I₂’s node) is detected. For example, M₂ should choose another input source (I₂’ or I₂’’).

(b) Same slow map source: Figure 3.1b shows a similar problem, but now during shuffle. Here, the map tasks (M₁, M₂) already complete normally; note that M₂ is also fast due to data locality (I₂→M₂ does not use the slow NIC). However, when shuffle starts, all the reducers (R₁, R₂) fetch M₂’s intermediate data through the slow NIC, hence “no” straggling reducers. Ideally,
if we monitor path progresses (the four arrows), the two straggling paths (M₂→R₁, M₂→R₂) and the culprit node (M₂’s node) can be easily detected.

While case (a) above might only happen in small-parallel jobs, case (b) can easily happen in large-parallel jobs, as it only needs one slow map task with data locality to hit the slow NIC.

(c) Same slow output intersection: Figure 3.1c shows another case in write phase. Here, the reducers’ write pipelines (R₁→O₁→O₁’₁, R₂→O₂→O₂’₂) intersect the same slow NIC (O₁/O₂’s node), hence “no” straggling reducer.

(d) SE algorithm “bug”: Figure 3.2a shows progress scores of three reducers, all observe a fast shuffle (the quick increase of progress scores to 0.8), but one reducer (R₁) gets a slow-NIC in its output pipeline (flat progress score). Ideally, R₁ (which is much slower than R₂&R₃) should be speculated, but SE is never triggered. Figure 3.2b reveals the root cause of why R₁ never get speculated. With a fast shuffle but a slow output, the estimated R₁ replacement time (EstReplace) is always slightly higher (0.1-0.3 minutes) than the estimated R₁ completion time (EstComplete). Hence, speculation is not deemed beneficial (incorrectly).

3.1.2 Straggling Backup Tasks

Let’s suppose a straggler is detected, then the backup task is expected to finish faster. By default, Hadoop limits one backup per speculated task (e.g., M₁’₁ for M₁, but no M₂’’). We found loopholes where this “one-shot” task speculation is also “unlucky” (involves the slow NIC again).
Figure 3.3: **Tail-SPOF and Straggling Backups.** Figures (a)-(c) are discussed in Sections 3.1.2a-c, respectively. I₂ in Figure (a), I₂/M₂ in (b), and M₂ in (c) are a tail-SPOF; the slow NIC is coincidentally involved again in the backup task. Please see Figure 2.1 for legend description.

(a) *Same slow input source:* Figure 3.3a shows a map (M₂) reading from a slow remote node (I₂) and is correctly marked as a straggler (slower than M₁). However, when the backup task (M₂') started, HDFS gave the *same* input source (I₂). As a result, *both* the original and backup tasks (M₂, M₂') are slow.

(b) *Same slow node in original and backup paths:* Figure 3.3b reveals a similar case but with a slightly different topology. Here, the backup (M₂') chooses a different source (I₂'), but it is put in the slow node. As a result, both the original and backup paths (I₂→M₂ and I₂'→M₂') involve a tail-SPOF (M₂'/I₂’s node).

(c) *Same slow map source:* Figure 3.3c depicts a similar shuffle topology as Figure 3.1b, but now the shuffle pattern is not all-to-all (job/data specific). Since M₂’s NIC is slow, R₂ becomes a straggler. The backup R₂' however *cannot* choose another map other than reading through the slow M₂’s NIC again. If the initial paths (the first three arrows) are exposed, a proper recovery can be done (*e.g.*, pinpoint M₂ as the culprit and run M₂').

### 3.1.3 The Cascading Impact

In Section 2.1.4 and Figure 2.2b, we show that the performance of the entire can eventually collapse. As explained in our previous work [65, §5.1.4], the reason for this is that the slow and unspeculated tasks are occupying the task containers/slots in healthy nodes for a long time. For example, in Figures 3.1 and 3.3, although only one node is degraded (bold edge), other nodes are
affected (striped nodes). Newer jobs that arrive are possible to interact with the degraded node. Eventually, all tasks in all nodes are “locked up” by the degraded NIC and there are not enough free containers/slots for new jobs. In summary, failing to speculate slow tasks from a degraded network can be fatal.

3.1.4 The Flaws

Hadoop is a decade-old mature software and many SE algorithms are derived from Hadoop/MapReduce \[63, 133\]. 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 $\neq$ 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 (e.g., in Figure 3.3c, $R_2$ is blamed even though $M_2$ is the culprit).

In other works, task sub-progresses are also lumped into one progress score (e.g., in Late [133, §4.4], Grass [47, §5.1], Mantri [48, §5.5], Wrangler [127, §4.2.1], Dolly [45, §2.1.1], ParaTimer [98, §2.5], Parallax [99, §3]). While these novel methods are superb in optimizing SE for other root causes (e.g., node contentions, heterogeneous resources), they do not specifically tackle network-only degradation (at individual nodes). Some of these works also tries to monitor data transfer progresses [48], but they are still lumped into a single progress score.
3.2 PBSE

We now present the three important elements of PBSE: path progress (§3.2.1), path diversity (§3.2.2), and path-straggler detection and speculation (§3.2.3), and at the end conclude the advantages of PBSE (§3.6). This section details our design in the context of Hadoop/HDFS stack with 3-way replication and 1 slow NIC ($F = 1$).

3.2.1 Paths

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., $I_2 \rightarrow M_2$) 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 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 $m \rightarrow R$ path in the shuffle stage.

- **Reduce→Output (R→O):** Unlike earlier paths above, an output path is a pipeline of sub-paths (e.g., $R_1 \rightarrow O_1 \rightarrow O'_1 \rightarrow O''_1$). 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 $R_1 \rightarrow O_1$ is fast, but $O_1 \rightarrow O'_1$ and $O'_1 \rightarrow O''_1$ are slow, $O'_1$ can be the culprit.

1. $F$ denotes the tolerable number of failures.
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.

3.2.2 Path Diversity

Straggler detection is only effective if independent progresses are comparable. However, patterns such as \( X \rightarrow M_1 \) and \( X \rightarrow M_2 \) 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 in every MapReduce stage:

(a) No input-SPOF in \( I \rightarrow m \) paths: It is possible that map tasks on different nodes read inputs from the same node (\( I_1 \rightarrow M_1 \), \( I_2 \rightarrow M_2 \), 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.

There are two possible designs, proactive and reactive. Proactive enforces all tasks of a job to synchronize with each other to verify the receipt of at least two input nodes. This early synchronization is not practical because tasks do not always start at the same time (depends on container availability). Furthermore, considering that in common cases not all jobs receive an input-SPOF, this approach imposes an unnecessary overhead.

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

\(^2\) A=B implies A and B are in the same node.
to re-pick another input node (e.g., I’₂→M₂ and I’₂≠I₁).³ After the switch (I₂ to I’₂), the task continues reading from the last read offset (no restart overhead).

(b) No map-SPOF in I→m and m→R paths: It is possible that map tasks are assigned to the same node (I₁→M₁, I₂→M₂, M₁=M₂, and M₁/M₂’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., M₁≠M’₁, M₂≠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. One caveat is when a job deploys only one map (reads only one input block), in which case we split it to two map tasks, each reading half of the input. This case however is very rare.

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→R and R→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 [45].

(d) No output-SPOF in R→O paths: Output pipelines of all reducers can intersect the same node (e.g., R₁→O₁→O’₁, R₂→O₂→O’₂, and O₁=O₂). Handling this output-SPOF is similar to Rule (a). However, since write is different than read, the pipeline re-picking overhead can be significant if not designed carefully.

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).

³ A≠B implies A and B are not in the same node.
Contrary, in our design, when shuffling finishes, the no-SPOF write pipelines are ready to use.

We now explain why pre-allocating pipelines removes a significant overhead. Unlike read switch in Rule (a), switching nodes in the middle of writes is not possible. In our strawman design, after pipeline creation (e.g., $R_2 \rightarrow X \rightarrow O'_2 \rightarrow ...$), reducers report paths to AM and begin writing, similar to Rule (a). Imagine when an output-SPOF $X$ is found but $R_2$ already wrote 5 MB. A simple switch (e.g., $R_2 \rightarrow Y \rightarrow ...$) is impossible because $R_2$ no longer has the data (because $R_2$’s HDFS client layer only buffers 4 MB of output). Filling $Y$ with the already-transferred data from $O'_2$ will require complex changes in the storage layer (HDFS) and alter its append-only nature. Another way around is to create a backup reducer ($R'_2$) with a new no-SPOF pipeline (e.g., $R'_2 \rightarrow Y \rightarrow ...$), which unfortunately incurs a high overhead as $R'_2$ must repeat the shuffle phase. For these reasons, we employ a background pre-allocation, which obviates pipeline switching in the middle of writes.

Another intricacy of output pipelines is that an output intersection does not always imply an output-SPOF. Let us consider $R_1 \rightarrow A \rightarrow X$ and $R_2 \rightarrow B \rightarrow X$. Although $X$ is an output intersection, there is enough sub-path diversity needed to detect a tail-SPOF. Specifically, we can still compare the upper-stream $R_1 \rightarrow A$ and the lower-stream $A \rightarrow X$ to detect whether $X$ is slow. Thus, as long as the intersection node is not the first node in all the write pipelines, pipeline re-picking is unnecessary. As an additional note, we collapse local-transfer edges; for example, if $R_1 \rightarrow A$ and $R_2 \rightarrow B$ are local disk writes, $A$ and $B$ are removed, resulting in $R_1 \rightarrow X$ and $R_2 \rightarrow X$, which will activate path diversity as $X$ is a a potential tail-SPOF.

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
M₂ must be placed somewhere else and will read its input \( (I_2/I'_2/I''_2) \) remotely.

### 3.2.3 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 \( BW \) (§3.2.1) 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., \( A \rightarrow B \)) 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 [134]. 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 \( A \rightarrow B \) 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.

   Let’s consider the shuffling topology in Figure 3.1b with 1000Mbps normal links and a slow M₂’s NIC at 5 Mbps. AM receives four paths (\( M_1 \rightarrow R_1, M_1 \rightarrow R_2, M_2 \rightarrow R_1, \) and \( M_2 \rightarrow R_2 \)) along with their bandwidths (e.g., 490, 450, 3, and 2 Mbps respectively). Four groups are created and the path bandwidths are grouped (\( M_1:\{490, 450\}, M_2:\{3, 2\}, R_1:\{490, 3\}, \) and \( R_2:\{450, 2\} \)). After the sums are computed, M₂’s node (5 Mbps total) will be marked as the culprit, implying

---

4. We omit our algorithm details because it is similar to task-level time-estimation and speculation [32]. The difference is that we run the speculation algorithm on path progresses, not just task-level progresses.
M₂ must be speculated, even though it is in the reduce stage (in PBSE, the stage does not define the straggler).

(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 I₁→M₁ and a stragglng I₂→M₂; the latter must be speculated, but the fault could be in I₂ or M₂. Fortunately, Hadoop SE by default prohibits M₂′ to run on the same node as M₂. Thus, we could speculate with I₂→M₂′. However, we take a greedy approach where we speculate a completely new pair I₂′→M₂′ (avoiding both I₂ and M₂ nodes). To implement this, when Hadoop spawns a task (M₂′), it can provide a blacklisted input source (I₂) to HDFS.

Arguably, node of I₂′ could be busier than I₂’s node, and hence our greedy algorithm is sub-optimal. However, we find that HDFS does not employ a fine-grained load balancing policy; it only tries to achieve rack/node data locality (else, it uses a random selection). This simplicity is reasonable because Hadoop tasks are evenly spread out, hence a balanced cluster-wide read.

(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 M₁→R₁ and a slow M₂→R₂). By deduction, since M₂ already “passes the check” in the I→M₂ stage (it was not detected as a slow-NIC node), then the culprit is likely to be R₂. Thus, M₂→R₂′ backup path will start. Compared to deduction approach, employing a greedy approach in shuffling stage is more expensive (e.g., speculating M₂′→R₂′ requires spawning M₂′).

(c) Dynamic retries: Using the same example above (slow M₂→R₂), caution must be taken if M₂ reads locally. That is, if I→M₂ only involves a local transfer, M₂ is not yet proven to be fault-free. In this case, blaming M₂ or R₂ 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.

### 3.3 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 [10], 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. We set $\beta = 0.1$ (§3.2.3), 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) [61]. In each, we pick a sequence of 150 jobs with the lowest

---

5. 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.

6. 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 3.4: Distribution of job sizes and inter-arrival times. The left figure shows CDF of the number of (map) tasks per job within the chosen 150 jobs from each of the production traces. The number of reduce tasks is mostly 1 in all the jobs. The right figure shows the CDF of job inter-arrival times.

inter-arrival time (i.e., a busy cluster). We use SWIM to replay and rescale the traces properly to our cluster sizes as instructed [33]. Figure 3.4 shows the distribution of job sizes and inter-arrival times.

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

### 3.3.1 PBSE vs. Hadoop (Base) SE

Figure 3.5 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 3.5. First, as alluded in §3.1, Hadoop SE cannot escape tail-SPOF caused by the degraded NIC, resulting in long job tail latencies with 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); we dissect this more later (§3.3.2). 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 net-
Figure 3.5: **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).

work. 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 3.6a), workload (3.6b), and cluster size (3.6c). 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 3.6a 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
Figure 3.6: **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 3.5 (e.g., “P80” denotes 80<sup>th</sup>-percentile). For example, the bold (1Mbps) line in Figure 3.6a plots the PBSE speedup at every percentile from Figure 3.5 (i.e., the horizontal difference between the PBSE-1Slow and Base-1Slow lines). As an example point, in Figure 3.5, at 80<sup>th</sup>-percentile (y=0.8), our speed-up is 1.7× (T<sub>Base</sub>/T<sub>PBSE</sub> = 54sec/32sec) but in Figure 3.6a, the axis is reversed for readability (at x=P80, PBSE speedup is y=1.7).

3.5). 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 3.6b 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 3.6c 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).

### 3.3.2 Detailed Analysis

We now dissect our first experiment’s results (Figure 3.5).

**Activated features:** Table 3.1 shows how many times PBSE features (§3.2.2-3.2.3) are activated in the PBSE-1Slow experiment in Figure 3.5. First, in terms of path diversity, 66 tasks (59 jobs) require I→m diversity. In the FB2010 workload, 125 jobs only have one reducer, thus requiring m→R diversity (reducer cloning). R→O diversity is rarely needed (0), mainly because of
Table 3.1: **Activated PBSE features** (§3.3.2). *The table shows how many times each feature is activated, by task and job counts, in the PBSE-1Slow experiment in Figure 3.5.*

<table>
<thead>
<tr>
<th>Features</th>
<th>In stage:</th>
<th>#Tasks</th>
<th>#Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path</td>
<td>I→M</td>
<td>66</td>
<td>59</td>
</tr>
<tr>
<td>Diversity</td>
<td>M→R</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>(<em>§3.2.2</em>)</td>
<td>R→O</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| Speculation  | I→M       | 62     | 54    |
| Speculation  | M→R       | 26     | 6     |
| Speculation  | R→O       | 28     | 12    |

**Figure 3.7:** **Residual sources of tail latencies** (§3.3.2).

<table>
<thead>
<tr>
<th>Start-up overhead</th>
<th>(a)</th>
<th>Fast task</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/M:</td>
<td></td>
<td>Good node</td>
</tr>
<tr>
<td>I/M:</td>
<td></td>
<td>Slow task</td>
</tr>
</tbody>
</table>

| Residual Tail     | (b) | Slow task |
| Residual Tail     | (b) | Slow task |
| Residual Tail     | (b) | Slow task |

| Residual Tail     | (b) | Slow task |
| Residual Tail     | (b) | Slow task |
| Residual Tail     | (b) | Slow task |

| Residual Tail     | (b) | Slow task |
| Residual Tail     | (b) | Slow task |
| Residual Tail     | (b) | Slow task |

<table>
<thead>
<tr>
<th>Start-up overhead</th>
<th>(c)</th>
<th>Slow task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual Tail</td>
<td>(c)</td>
<td>Slow task</td>
</tr>
<tr>
<td>Residual Tail</td>
<td>(c)</td>
<td>Slow task</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start-up overhead</th>
<th>(c)</th>
<th>Slow task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual Tail</td>
<td>(c)</td>
<td>Slow task</td>
</tr>
<tr>
<td>Residual Tail</td>
<td>(c)</td>
<td>Slow task</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start-up overhead</th>
<th>(c)</th>
<th>Slow task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual Tail</td>
<td>(c)</td>
<td>Slow task</td>
</tr>
<tr>
<td>Residual Tail</td>
<td>(c)</td>
<td>Slow task</td>
</tr>
</tbody>
</table>

**Residual overhead:** We next discuss interesting findings on why PBSE cannot reach the “perfect” case (Base-0Slow vs. PBSE-1Slow lines shown in Figure 3.5). Below are the sources of residual overhead that we discovered:

**Start-up overhead:** Figure 3.7a shows that even when a straggling path (I₂→M₂) is detected early, running the backup task (M₂') will require 1-3 seconds of start-up overhead, which includes JVM warm-up (class loading, interpretation) and “localization” [28] (including transferring application’s .jar files from HDFS to the task’s node). At this point, this overhead is not removable and still become an ongoing problem [88].
“Straggling” start-up/localization: Figure 3.7b shows two maps with one of them (M₂) running on a slow-NIC node. This causes the transferring of application’s .jar files from HDFS to M₂’s node to be slower than the one in M₁. In our experience, this “straggling” start-up can consume 1-9 seconds (depending on the job size). What we find interesting is that start-up time is not accounted in task progress and SE decision making. In other words, start-up durations of M₁ and M₂ are not compared, and hence no straggling start-up is detected, delaying the task-straggler detection.

“Straggling” clean-up: Figure 3.7c shows that after a job finishes (but before returning to user), the AM must write a JobHistory file to HDFS (part of the clean-up operation). It is possible that one of the output nodes is slow (e.g., AM→O→O′→O″), in which case AM will be stuck in this “straggling” clean-up, especially with a large JobHistory file from a large job (M₁..Mₙ, R₁..Rₙ, n≫1). This case is also outside the scope of SE.

We believe the last two findings reveal more flaws of existing tail-tolerance strategies: they only focus on “task progress,” but do not include “operational progress” (start-up/clean-up) as part of SE decision making, which results in irremovable tail latencies. Again, these flaws surface when our unique fault model (§2.1.3) is considered. Fortunately, all the problems above are solvable; start-up overhead is being solved elsewhere [88] and straggling localization/clean-up can be unearthed by incorporating start-up/clean-up paths and their latencies as part of SE decision making.

3.3.3 PBSE vs. Other Strategies

In this section, we will compare PBSE against other scheduling and tail-tolerant approaches.

Figure 3.8a shows the same setup as in Figure 3.5, but now we vary the scheduling configurations: capacity (default), FIFO, and fair scheduling. The figure essentially confirms that the SE loopholes (§3.1) are not about scheduling problems; changing the scheduler does not eliminate tail latencies induced by the degraded NIC.

Figure 3.8b shows the same setup, but now we vary the tail-tolerant strategies: hedged read
(0.5s), hedged read (0s), aggressive SE, and task cloning. The first one, hedged read (HRead-0.5), is a new HDFS feature [34] that enables a map task (e.g., I2→M2) to automatically read from another replica (I′2→M2) if the first 64KB packet is not received after 0.5 sec. The map will use the data from the fastest read. The second one (HRead-0.0) does not wait at all. Hedged read can unnecessarily consume network resources. As shown, HRead-0.0 does not eliminate all the tail-SPOF latencies (with a job latency maximum of 7708 seconds). This is because hedged read only solves the tail-SPOF scenarios in the input stage, but not across all the MapReduce stages.

The third one (Aggr-SE) is the base SE but with the most aggressive SE configuration\(^7\) and the last one (Cloning) represents task-level cloning\(^8\) [45] (a.k.a. hedged requests [62]). Aggressive SE speculates more intensively in all the MapReduce stages, but long tail latencies still appear (with a maximum of 6801 seconds). Even cloning also exhibits a long tail (3663 seconds at the end of the tail) as it still inherits the flaws of base SE. In this scenario, PBSE is the most effective (a maximum of only 251 seconds), as it solves the fundamental limitations of base SE.

In summary, all other approaches above only reduce but do not eliminate the possibility of

---

7. Aggressive SE configurations:  
   speculative-cap-running-tasks=1.0 (default=0.1);  
   speculative-cap-total-tasks=1.0 (default=0.01);  
   minimum-allowed-tasks=100 (default=10);  
   retry-after-speculate=1000ms (default=15000ms); and  
   slowtaskthreshold=0.1 (default=1.0).

8. We slightly modified Hadoop to always speculate every task at the moment the task starts (Hadoop does not support cloning by default).
We did not empirically evaluate PBSE with other strategies in the literature [45, 46, 48, 98, 99, 127] because either they are either proprietary or integrated to non-Hadoop frameworks (e.g., Dryad [82], SCOPE [58], or Spark [9]), but they were discussed earlier (§3.1.4).

### 3.3.4 PBSE with Multiple Failures

We also evaluate PBSE against two NIC failures (both 1 Mbps). This is done by changing the value of $F$ (§3.2). To handle two slow-NIC nodes ($F=2$), at least three nodes ($F+1$) are required for path diversity. $F=3$ is currently not possible as the replication factor is only $3 \times$. Figure 3.9a shows that PBSE performs effectively as well under the two-failure scenario.

### 3.3.5 PBSE on Heterogeneous Resources

To show that PBSE does not break the performance of Hadoop SE under heterogeneous resources [133], we run PBSE on a stable-state network throughput distribution in Amazon EC2 that is popularly cited (ranging from 200 to 920 Mbps; please see Figure 3 in [121] for the detailed distribution). Figure 3.9b shows that PBSE speed-up is constantly around one with small fluctuations at high percentiles from large jobs. The figure also shows the different path-straggler thresholds we use ($\beta$ in §3.2.3). With a higher threshold, sensitivity is higher and more paths are speculated. However, because the heterogeneity is not severe (>100 Mbps), the original tasks always complete faster than the backup tasks.
3.3.6 Limitations

PBSE can fail in extreme corner cases: for example, if a file currently only has one surviving replica (path diversity is impossible); if a large batch of devices degrade simultaneously beyond the tolerable number of failures; or if there is not enough node availability. Note that the base Hadoop SE also fails in such cases. When these cases happen, PBSE can log warning messages to allow operators to query the log and correlate the warnings with slow jobs (if any).

Another limitation of PBSE is that in a virtualized environment (e.g., EC2), if nodes (as VMs) are packed to the same machine, PBSE’s path diversity will not work. PBSE works if the VMs are deployed across many machines and they expose the machine#.

3.4 Beyond Hadoop and HDFS

To show PBSE generality for many other data-parallel frameworks beyond Hadoop/HDFS, we analyzed Apache Spark [9, 132], Flume [4], S4 [8], and Quantcast File System (QFS) [105]. We found that all of them suffer from the tail-SPOF problem, as shown in Figure 3.10 (Base-0Slow vs. Base-1Slow bars). We have performed an initial integration of PBSE to Spark, Flume, and Hadoop/QFS stack and showed that it speculates effectively, avoids the degraded network, and cuts tail latencies (PBSE-1Slow bars). We did not integrate further to S4 as its development is discontinued.

We briefly describe the tail-tolerance flaws we found in these other systems. Spark (Figure 3.10b) has a built-in SE similar to Hadoop SE, hence it is prone to the same tail-SPOF issues (§3.1). Flume (Figure 3.10c) uses a static timeout to detect slow channels (no path comparisons). If a channel’s throughput falls below 1000 events/sec, a failover is triggered. If a NIC degrades to slightly above the threshold, the timeout is not triggered. S4 (Figure 3.10d) has a similar static timeout to Flume. Worse, it has a shared queue design in the fan-out protocol. Thus, a slow recipient will cripple the sender in transferring data to the other recipients (a head-of-line blocking problem).
Figure 3.10: Beyond Hadoop/HDFS (§3.4). The figure shows latencies of microbenchmarks running on four different systems (Hadoop/QFS, Spark, Flume, and S4) with three different setups: baseline without degraded NIC (Base-0Slow), baseline with one 1Mbps degraded NIC (Base-1Slow), and with initial PBSE integration (PBSE-1Slow). Baseline (Base) implies the vanilla versions. The Hadoop/QFS microbenchmark and topology is shown in Figure 3.11c. The Spark microbenchmark is a 2-stage, 4-task, all-to-all communication as similarly depicted in Figure 3.1b. The Flume and S4 microbenchmarks have the same topology. We did not integrate PBSE to S4 as its development is discontinued.

We also analyzed the Hadoop/QFS stack due to its differences from the Hadoop/HDFS (3-way replication) stack. Hadoop/QFS represents computation on erasure-coded (EC) storage. Many EC storage systems [80, 105, 124] embed tail-tolerant mechanisms in their client layer. EC-level SE with \( m \) parities can tolerate up to \( m \) slow NICs. In addition to tolerating slow NICs, EC-level SE can also tolerate rack-level slowdown (which can be caused by a degraded TOR switch or a malfunctioning power in the rack). For example in Figure 3.11a, \( M_1 \) reads chunks of a file (\( I_a, I_b \)). As reading from \( I_b \) is slower than from \( I_a \) (due to the slow Rack-2), the EC client layer triggers its own EC-level SE, creating a backup speculative read from \( I_p \) to construct the late \( I_b \).

Unfortunately, EC-level SE also has loopholes. Figure 3.11b shows a similar case but with slow Rack-1. Here, the EC-level SE is not triggered as all reads (\( I_a \rightarrow M_1, I_b \rightarrow M_1 \)) are slow. Let’s suppose another map (\( M_2 \)) completes fast in Rack-3, as in Figure 3.11c. Here, Hadoop declares \( M_1 \) as a straggler, however it is possible that the backup \( M_1' \) will run in Rack-1, which means it must also read through the slow rack. As a result, both original and backup tasks (\( M_1 \) and \( M_1' \)) are

---

9. Quantcast File System (QFS) [27, 105] is a Reed-Solomon (RS) erasure-coded (EC) distributed file system. Although HDFS-RAID supports RS [30], when we started the project (in 2015/2016), the RS is only executed in the background. In HDFS-RAID, files are still triple-replicated initially. Moreover, because the stripe size is the same as the block size (64 MB), only large files are erasure coded while small files are still triple replicated (e.g., with RS(10,4), only files with \( 10 \times 64 \) MB size are erasure coded). HDFS with foreground EC (HDFS-EC) was still an ongoing non-stable development [15]. In contrast, QFS erasure-code data in the foreground with 64KB stripe size, hence our usage of QFS.
Figure 3.11: **Rack slowdown and Erasure-Coded (EC) storage (§3.4).** For simplicity, the figure shows RS(2,1), a Reed Solomon where an input file I is striped across two chunks (I_a, I_b) with one parity (I_p) with 64KB stripe size (see Figure 2 in [105] for more detail).

To address this, with PBSE, the EC layer (QFS) exposes the individual read paths to Hadoop. In Figure 3.11c, if we expose I_a→M_1 and I_b→M_1 paths, Hadoop can try placing the backup M'_1 in another rack (Rack-2/3) and furthermore informs the EC layer to have M'_1 directly read from I_b and I_p (instead of I_a). Overall, the key principle is the same: when path progresses are exposed, the compute and storage layers can make a more informed decision. Figure 3.10a shows that in a topology like Figure 3.11c, without PBSE, the job follows the slow Rack-1 performance (Base-1Slow bar), but with PBSE, the job can escape from the slow rack (PBSE-1Slow).

In summary, we have performed successful initial integrations of PBSE to multiple systems, which we believe show its generality to any data-parallel frameworks that need robust tail tolerance against node-level network degradation. We leave full testing of these additional integrations as a future work.

### 3.5 Related Work

We now discuss related work.
**Paths:** The concept of “paths” is prevalent in the context of modeling (e.g., Magpie [53]), fine-grained tracing (e.g., XTrace [66], Pivot Tracing [94]), diagnosis (e.g., black-box debugging [41], path-based failure [60]), and availability auditing (e.g., INDaaS [134]), among many others. This set of work is mainly about monitoring and diagnosing paths. In PBSE, we actively “control” paths, for a better online tail tolerance.

**Tail tolerance:** Earlier (§3.1.4), we have discussed a subtle limitation of existing SE implementations that hide path progresses [45, 47, 48, 98, 99, 133]. While SE is considered a reactive tail-tolerance, proactive ones have also been proposed, for example by cloning (e.g., Dolly [45] and hedged requests [62]), launching few extra tasks (e.g., KMN [120]), or placing tasks more intelligently (e.g., Wrangler [127]). This novel set of work also uses the classical definition of progress score (§3.1.4). Probabilistically, cloning or launching extra tasks can reduce tail-SPOF, but fundamentally, as paths are not exposed and controlled, tail-SPOF is still possible (§3.3.3). Tail tolerance is also deployed in the storage layer (e.g., RobuStore [124], CostTLO [123], C3 [115], Azure [80]). PBSE shows that storage and compute layers need to collaborate for a more robust tail tolerance.

**Task placement/scheduling:** There is a large body of work on task placement and scheduling (e.g., Pacman [46], delay scheduling [131], Quincy [83], Retro [93]). These efforts attempt to cut tail latencies in the initial task placements by achieving better data locality, load balancing, and resource utilization. However, they do not modify SE algorithms, and thus SE (and PBSE) is orthogonal to this line of work.

**Tail root causes:** A large body of literature has discovered many root causes of tail latencies including resource contention of shared resources [48, 50], hardware performance variability [87], workload imbalance [67], data and compute skew [44], background processes [87], heterogeneous resources [133], degraded disks or SSDs [74, 128], and buggy machine configuration (e.g., disabled process caches) [63]. For degraded disks and processors, existing (task-based) speculations are sufficient to detect such problems. PBSE highlights that degraded network is an important fault
model to address.

**Iterative graph frameworks:** Other types of data-parallel systems include iterative graph processing frameworks [5, 40, 100]. As reported, they do not employ speculative execution within the frameworks, but rather deal with stragglers within the running algorithms [75]. Our initial evaluation of Apache Giraph [5] shows that a slow NIC can hamper the entire graph computation, mainly because Giraph workers must occasionally checkpoint its states to HDFS (plus the use of barrier synchronization), thus experiencing a tail-SPOF (as in Figure 3.1c). This suggests that part of PBSE may be applicable to graph processing frameworks as well.

**Distributed system bugs:** Distributed systems are hard to get right. A plethora of related work combat a variety of bugs in distributed systems, including concurrency [72, 85], configuration [125], dependency [134], error/crash-handling [84, 130], performance [94], and scalability [86] bugs. In this work, we highlight performance issues caused by speculative execution bugs/loopholes.

### 3.6 Conclusion

Performance-degraded mode is dangerous; software systems tend to continue using the device without explicit failure warnings (hence, no failover). Such intricate problems took hours or days until manually diagnosed, usually after whole-cluster performance is affected (a cascading failure) [21, 24, 25]. In this work, we show that node-level network degradation combined with SE loopholes is dangerous. We believe it is the responsibility of software’s tail-tolerant strategies, not just monitoring tools, to properly handle performance-degraded network devices. To this end, we have presented PBSE as a novel, online solution to the problem.
CHAPTER 4

COBE: CASCADING OUTAGE BUG ELIMINATION

In Section 2.2, we have presented our case about cascading outage (CO) bugs. Our large-scale studies of cloud bugs and outages \[69, 70\] reveal some cases of bugs that can cause simultaneous or cascades of failures to each of the individual nodes in the system. And through our CO bugs studies presented in Section 2.2.1, we have a better understanding of the various patterns of CO bug. The cases from CO bugs study confirm that there are 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.

In this chapter, we discuss COBE, our program analysis framework to help detect the existence of CO patterns in distributed systems. We will discuss the design principle of the program analysis framework that we build in 4.1 and introduce COBE in Section 4.2. We discuss COBE implementation in Section 4.3 and evaluate it in Section 4.4. Lastly, we discuss related work and conclusion in Section 4.5 and 4.6 respectively.

### 4.1 Design Principles

From the sample of CO bugs that we present in Section 2.2.1, we get a glimpse of the characteristics of CO bugs. For a CO bug to surface, it requires complex interleavings and combinations of failures to happen. It may crash the system immediately, or went silent for hours or days from the initial triggering until the outage occurs. These preconditions are complex enough such that CO bugs went uncaught during the development phase, even under rigorous unit testings. Model-checking and other dynamic analysis may help to reveal these CO bugs, but they are often based on actual program runs, hence often by how program execution behaves in ordinary settings.

We believe that some CO bugs share similar patterns in the code, whether it is a race condition, resource leak, or network error pattern. We can capture a potential CO bug by doing careful
program analysis to detect if certain CO bug patterns exist in the system.

However, expressing a pattern of CO bugs is not quite straightforward, let alone expressing it for distributed systems. First, a specific CO pattern may involve multiple analyses to be combined. For example, capturing *transient network error* pattern will require network error analysis plus registration protocol analysis, *race in master* pattern will require race analysis plus analysis of message communication between different nodes, and so on. The second problem is the sheer number of CO patterns to cover. Writing individual program analyses for each different CO pattern will be impractical to do.

We need a program analysis framework that can do compositional analysis where we can express high-level rules of CO pattern by combining results from multiple small program analysis. Compositional analyses also enable us to reuse the results of program analyses shared by different CO pattern analysis. These design principles underline how we build our program analysis framework, COBE, which we will explain in the next section.

### 4.2 COBE

We present COBE, static program analysis framework to detect CO bugs pattern in distributed systems. COBE is a program analysis stack that combines several program analyses to reveal certain patterns of CO bugs. The goal of COBE analysis stack is to extract system properties such as crash paths, recovery paths, and so on, from the code and correlate between them to reveal CO pattern.

CO bugs have diverse patterns, and each of them has different characteristics. Instead of writing individual program analysis to capture each of the different CO patterns, we design COBE to detect different CO patterns in compositional manners.

Figure 4.1 shows the idea of compositional program analysis in COBE. At the very bottom layer, is a database of program facts. COBE gathers these program facts by extract it from the binaries of the target system. These program facts range from a list of classes, methods, class
Figure 4.1: **Cobe Analysis Stack.** *The figure above show the high level idea of COBE analysis framework.*

Hierarchies, method invocations, and so on. At the top layer is a set of CO pattern queries. CO pattern query is a high-level query to describe a CO pattern. The top and the bottom layer is glued by the middle layer, which is a layer of program analysis libraries.

These three layers have a resemblance with RDBMS, where the CO pattern query layer is similar to SQL query, the middle library layer is like subqueries, and the bottom program facts layer is like the database. When we execute a CO pattern query, it will lookup for a set of program analyses required to capture that pattern from the program analysis library. Each activated program analysis will then read several program facts from the bottom layer and correlate them to produce results. Similarly, the results from program analysis libraries then returned to the high-level CO pattern query for it to correlate and produce a list of bug reports.

Different CO pattern query may share the same program analysis library to run. Similarly, a program analysis library might depend on or share common analysis with other program analysis library. When such common dependency occurs, the analysis result from the shared library is not recomputed but retained for all of its dependents.
4.3 Implementation

COBE is implemented in 11K LOC of Java and 2K LOC of Datalog. Figure 4.2 shows the architecture of COBE framework implementation. COBE analyzes the target system in two phases: parsing phase and query phase.

In the parsing phase, COBE will parse the code and extract comprehensive facts about target systems program structures and store them into facts database. These facts ranging from the system’s class hierarchy, function call graph, control flow, basic blocks, instruction list, mappings between variable definitions and uses, and so on. This phase also includes built-in pointer analysis that required to compute the function call graph.

The second step in COBE is the query phase. In this phase, we are querying for a particular CO bug pattern by correlating program facts gathered from the previous phase using Datalog as our query language. All program analyses in the library layer and high-level CO pattern queries are expressed as a set of connected Datalog rules. Currently, we have implemented the analysis of race in master and transient network error CO pattern using Datalog language. We will review more about Datalog in Section 4.3.2.

4.3.1 Program Facts Extraction

In the parsing phase, COBE extracts structural information of the target system such as class hierarchies, function call graphs, and control-flow graphs. COBE leverage WALA [35] to do this
Table 4.1: Program facts domain. This table lists domain of individual attribute in program facts relations that extracted by COBE parser. Domains can have type number (integer) or symbol (string). For clarity, we define domain Num and Sym to represent generic number and symbol data.

<table>
<thead>
<tr>
<th>Domains</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cm</td>
<td>symbol</td>
<td>Methods reference</td>
</tr>
<tr>
<td>Cb</td>
<td>number</td>
<td>Basic block number</td>
</tr>
<tr>
<td>Ceb</td>
<td>number</td>
<td>Exploded basic block node number</td>
</tr>
<tr>
<td>Ct</td>
<td>symbol</td>
<td>Type/class reference</td>
</tr>
<tr>
<td>Cf</td>
<td>symbol</td>
<td>Field reference</td>
</tr>
<tr>
<td>Cii</td>
<td>number</td>
<td>instruction index</td>
</tr>
<tr>
<td>Cv</td>
<td>number</td>
<td>Value number</td>
</tr>
<tr>
<td>Cs</td>
<td>symbol</td>
<td>Method selector</td>
</tr>
<tr>
<td>Cc</td>
<td>symbol</td>
<td>WALA CGNode context</td>
</tr>
<tr>
<td>Num</td>
<td>number</td>
<td>Generic number</td>
</tr>
<tr>
<td>Sym</td>
<td>symbol</td>
<td>Generic symbol</td>
</tr>
</tbody>
</table>

initial program parsing. WALA is a mature industrial-level program analysis tool that already has versatile tools inside it such as pointer analysis, call graph generation, and control/data flow analysis that we can use out of the box. COBE then read the generated call graph and control/data flow information from WALA and parse it into a set of program facts relation.

Specifically, each relation is stored as tab-separated values (tsv). Each attribute (column) of the tsv file belongs to one of the domains listed in Figure 4.1. These tsv files later then feed into Datalog solver as the database for the query phase. Table 4.2 lists some of the resulting relations from this parsing phase. There are in total 55 program facts relations that COBE can extract from system binaries.

Selective Program Analysis

Full program analysis can be costly and produce too many program facts that may be unrelated to CO bugs patterns. Since we are targetting CO bugs that involve the global system states, we are not doing a whole program analysis in this phase. Instead, we only focus on analyzing namespaces that contain most of the global system states modification and coordination, namely the server-level codes. For example, in HBASE system, this server-level code resides in namespace
### Program facts relation

<table>
<thead>
<tr>
<th>Program facts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bbDef(Cm, Cb, Cv, Num)</code></td>
<td>Method <code>Cm</code> block <code>Cb</code> define value <code>Cv</code></td>
</tr>
<tr>
<td><code>bbEbbTupleInt(Num, Cb, Ceb)</code></td>
<td>In method <code>Num</code>, block <code>Cb</code> contains exploded block <code>Ceb</code></td>
</tr>
<tr>
<td><code>bbExit(Cm, Cb)</code></td>
<td><code>Cb</code> is exit block of method <code>Cm</code></td>
</tr>
<tr>
<td><code>bbUse(Cm, Cb, Cv, Num)</code></td>
<td>Method <code>Cm</code> block <code>Cb</code> use value <code>Cv</code></td>
</tr>
<tr>
<td><code>bbInstTuple(Cm, Cb, Cii)</code></td>
<td>Method <code>Cm</code> block <code>Cb</code> has instruction with index <code>Cii</code></td>
</tr>
<tr>
<td><code>callWithContext((Cm1, Cc1, Cb1, Cm2, Cc2))</code></td>
<td>Method <code>Cm1</code> context <code>Cc1</code> call method <code>Cm2</code> context <code>Cc2</code> through block <code>Cb1</code></td>
</tr>
<tr>
<td><code>classDeclareMethod(Ct, Cm)</code></td>
<td><code>Ct</code> declare method <code>Cm</code></td>
</tr>
<tr>
<td><code>dictCm(Num, Cm)</code></td>
<td>Method <code>Cm</code> has id <code>Num</code></td>
</tr>
<tr>
<td><code>dominateInt(Num, Cb1, Cb2)</code></td>
<td><code>Block Cb1</code> dominate <code>Cb2</code> in method id <code>Num</code></td>
</tr>
<tr>
<td><code>ifaceSelector(Ct, Cs)</code></td>
<td>Interface <code>Ct</code> define method with selector <code>Cs</code></td>
</tr>
<tr>
<td><code>implement(Ct1, Ct2)</code></td>
<td><code>Ct2</code> is immediate subclass of <code>Ct1</code></td>
</tr>
<tr>
<td><code>instInvoke(Cm1, Cb1, Sym, Cm2)</code></td>
<td>Method <code>Cm1</code> block <code>Cb1</code> has invocation to <code>Cm2</code></td>
</tr>
<tr>
<td><code>instNew(Cm, Cb, Ct)</code></td>
<td>Method <code>Cm</code> block <code>Cb</code> allocate object of type <code>Ct</code></td>
</tr>
<tr>
<td><code>interf ace(Ct)</code></td>
<td><code>Ct</code> is an interface</td>
</tr>
<tr>
<td><code>killerExHandling((Cm, Ceb1, Ceb2, Ceb3, Num))</code></td>
<td>Exception thrown in block <code>Ceb1</code> is handled in <code>Ceb2</code> and trigger abort in <code>Ceb3</code></td>
</tr>
<tr>
<td><code>methodSelector(Cm, Cs)</code></td>
<td>Method <code>Cm</code> has selector <code>Cs</code></td>
</tr>
<tr>
<td><code>mtdUnhandledInvoke((Cm1, Cb1, Cm2)</code></td>
<td>Method <code>Cm</code> invoked in <code>Cb1</code> may throw exception and not handled locally</td>
</tr>
<tr>
<td><code>rpcInterface(Ct)</code></td>
<td><code>Ct</code> is an RPC interface</td>
</tr>
<tr>
<td><code>rpcMethod(Cm)</code></td>
<td><code>Ct</code> is an RPC method</td>
</tr>
</tbody>
</table>

**Table 4.2: Examples of extracted program facts.** Each attribute has their own domain (see Table 4.1). Subscript number signify attributes that correlated with each other. For example, `(Cm1, Cb1)` together represent a basic block number `Cb` within method `Cm`.

---

org.apache.hadoop.hbase.master for code related to HMaster and org.apache.hadoop.hbase.regionserver for RegionServer. While in the HDFS system, we can focus on namespace org.apache.hadoop.hdfs.server for both NameNode and DataNode codes. Besides of the server-level codes, we also add namespae that defines IPC communications into our analysis scope, so we can capture the interaction between server nodes.

By default, WALA will include all main methods found in the target system as default entry points to start building the call graph. However, building a call graph just from these default entry points is not enough, especially if the target system is a distributed system. In distributed systems, many parts of the codes are not necessarily reachable directly through the main methods, such as RPC server methods and periodic task threads. RPC server methods usually only called
by RPC client or the worker nodes and not called internally by the server itself. Similarly, from main method entry points, WALA will only see the instantiation and start invocation of the thread, but not the runnable body of the thread itself. COBE provides two options to broaden WALA’s visibility over the target system code. First, COBE provides a configuration file where users can specify additional entry points by either specifying important types or method names that should be added as additional entry points. Such type can be an RPC protocol interface or abstract type of event handler threads. After WALA’s class hierarchy analysis and before call graph analysis, COBE will search for names specified in the user configuration file and inject them as additional entry points if they are found. The second option is by supplying harness code to COBE. Harness code is an additional main method code to help guide WALA to find the important entry points that are not directly reachable through existing system main methods. This harness code should contain explicit invocation to methods that should be included as entry points. It needs to be compileable, but does not need to be executable. Harness code can be written outside of target system code, therefore it is not intrusive. Figure 4.3 shows an example of a harness code that we use to analyze hb16367.

Context Sensitivity

Another aspect that plays roles in scalability and precision of COBE parser is context-sensitivity selection. There are two popular kinds of context sensitivity that usually employed in object-oriented languages: call-site-sensitivity (k-cfa) [109] or object-sensitivity (k-obj) [97]. K-cfa uses the last k call site into the method call as context elements, while k-obj uses the last k allocation site of the receiver object as context elements. Both k-cfa and k-obj offer high precision in analysis. However, our experience in using any of them for distributed system yields in either exponential fact results, long parsing time, or loss in function calls. The exponential result and long parsing time are due to the combination of the target system volume and selection of number k. High precision context also causes WALA call graph builder to stop the call path exploration if within
```java
public class CobeHarness extends Thread {

    @Override
    public void run() {
        try {
            ExecutorService executor = Executors.newFixedThreadPool(5);
            Configuration conf = new Configuration();
            CoordinatedStateManager cp =
                CoordinatedStateManagerFactory.getCoordinatedStateManager(conf);
            HMaster master = new HMaster(conf, cp);
            executor.execute(master);
        } catch (Exception ex) {
        }
    }

    public static void main(String[] args) throws Exception {
        CobeHarness ch = new CobeHarness();
        ch.start();
    }
}
```

Figure 4.3: **Example of Harness Code.** The listing above shows example of COBE harness used to analyze `hb16367`.

a procedure it can not determine the concrete type of a call site target or receiver object, causing some losses in the resulting call graph.

To get better scalability and more complete call graph, we choose a more relax type-sensitivity [111]. Type-sensitivity is almost similar to object-sensitivity, but where an object-sensitive analysis will keep an allocation site as a context element, a type-sensitive analysis will keep the type instead. COBE achieves this type-sensitivity by using WALA built-in ZeroCFA pointer analysis policy and `ReceiverTypeContextSelector`, a context selector that takes the receiver object type as the context element. Type-sensitivity is especially helpful in creating a complete function call graph. When WALA finds out that there is more than one possible concrete type of receiver object in a particular all site, it will not stop path exploration, but instead, it will explore all the possibilities of receiver object types where that call site might go. For COBE, it is more important to receive a function call graph that reveals wider call paths but contain some ambiguity, rather than losing those call paths at all.
Pre-computed analysis

While the majority of COBE analysis resides later in the query phase, COBE also does some pre-computation analyses in the parsing phase such as dominance analysis and exception propagation. These pre-computation analyses are usually a type of intraprocedural analysis that can be done on-the-fly while reading program structure, such as dominance analysis, exception propagation analysis, IPC call analysis, and so on.

4.3.2 Datalog-based Analysis

Given the program facts we gain from the parsing phase, we can proceed to correlate these facts and search the existence of CO bug patterns in the system. In our initial COBE implementation, we write this phase as a Java program, along with the parsing code. However, the sheer complexity of rules to describe a CO bug pattern makes it difficult to express the analysis query algorithm in Java program.

In recent years, there has been a resurgence of interest in Datalog as a query language for a wide range of new applications. This include data integration [26, 17, 22], declarative networking [29, 28, 27], program analysis [13], information extraction [19, 38], network monitoring [5], security [31, 25], and cloud computing [7]. Furthermore, prior success in adopting Datalog for program analysis motivates us to use Datalog query language to express our COBE analysis.

Datalog is a declarative logic programming language that syntactically is a subset of Prolog, where a predicate is defined as a conjunction of other predicates. For example, the Datalog rule

\[ A(w, z) : \neg B(w, x), C(x, y), D(y, z). \]

says that “\( A(w, z) \) is true if \( B(w, x), C(x, y), D(y, z) \) are all true”. Variables in the predicates can be replaced with constants, which are surrounded by double quotes, or don’t-cares, which are signified by underscores. Predicates on the right side of the rules can be inverted.

Datalog is more powerful than SQL because Datalog predicates can be recursively defined[118]. Take an example of this Datalog query to query all class hierarchies in COBE.
extends(super, sub) :-
    immediateSubclass(super, sub).

extends(super, subsub) :-
    extends(super, sub),
    immediateSubclass(super, subsub).

Program analyses are highly recursive in nature, making Datalog a natural fit for it.

Predicates in Datalog are divided into two classes: extensional database (EDB), the relation that is stored in the database; and intentional database (IDB), that is all relation defined by one or more rules. All program facts extracted by COBE and domain knowledge relations defined by users are EDBs, while program analysis libraries and high-level CO pattern rules are IDBs. A collection of datalog rules is also called a Datalog program.

In the current iteration of COBE, we have implemented high-level rules to detect race in master and transient network error CO patterns in Souffle[31], a variant of Datalog language. Figure 4.4 illustrates activated Datalog rules and their dependencies across COBE analyses stack for these two high-level CO pattern rules. We will describe these high-level rules to more detailed in Section 4.3.3 and 4.3.4 respectively.

### 4.3.3 Race in Master Analysis

For race in master CO pattern, the high-level idea is a race condition that happens in the master node will cause it to crash or hang. The race may be triggered by inter-node communication or interaction between the master node’s internal periodic threads. To capture this CO pattern, our Datalog analyses will build a static happen-before model of the system and find concurrent and conflicting memory access in the happen-before model that can lead to system crash. Two events are said to be concurrent if there are no happen-before causality relationships between them, while conflicting means multiple access are touching the same memory location with at least one write.
Figure 4.4: **Dependency graph among COBE’s Datalog rules.** The figure above show activated rules and their dependencies across COBE analyses stack. The dashed box represent a Datalog program file that contain specific rules or program facts. The arrows coming out from the box means one or more rules in the Datalog program depends on rule pointed by the arrow head.

Happen-before (HB) model has been thoroughly studied in the literature [78, 79, 89, 95, 103, 107]. To build our HB model, we use HB rules from DCatch[89], specifically the synchronous RPC (Rule-\(M^{rpc}\)), custom push-based synchronization (Rule-\(M^{push}\)), synchronous multi-threaded concurrency (Rule-\(T^{fork}\)), and sequential program ordering (Rule \(P^{reg}\)). However, DCatch HB rules can not be directly applied in COBE static analysis settings. There are two problems that we need to address.

The first problem is how to select HB nodes and edges in static analysis settings. DCatch uses
real program execution trace as the basic building block to build their HB graph. COBE, on the other hand, builds its static HB model based on program facts retrieved from the parsing phase. It will use the HB rules to guides which functions that should be added to the HB graph. The HB edges then applied between them based on the logical relationship defined by the HB rules, the call graph, and the class hierarchy relationship. The result from this process is a static HB graph that almost looks like a call graph, but with most of the function call unrelated to the HB rules left out. Each vertex in our HB graph is a program point represented as a tuple \((m, b, h)\), where \(m\) is a call-graph node from WALA (which is pair of a method name and its type-context), \(b\) is a basic block number, and \(h\) is the type of HB node. The edges are the HB relationship between two program point, saying that program point \((m_1, b_1, h_1)\) happen before program point \((m_2, b_2, h_2)\).

Memory accesses are also represented as HB nodes. However, this information needs to be specified by COBE user. User must specify which field member need to be checked for write and read access or, in case of global data structure, which method is used to write and read.

Given the HB rules, we first search for all related methods than can be applied to the rules. For example, given a Synchronous RPC (Rule-M\(^{rpc}\)) HB rule, we will search all method implementation of an RPC interface and all methods that contain call-site to that RPC interface. The basic block containing that RPC call-site, along with both entry and exit blocks of the RPC implementation, then taken as HB node. We then use the definition of the HB rule to properly add HB edges between them. For synchronization rules that do not share an interface such as the case in Custom push-based synchronization protocol (Rule-M\(^{push}\)), COBE allows users to manually specify the correlation between methods that represent Update and the corresponding method representing Pushed.

However, this resulting static HB graph is not yet enough to reflect concurrencies that happen in the target system. Because of our selection of type-sensitivity in the parsing phase, different call paths into a function are not distinguished and will be collapsed into the same node. Different chain of events should be represented in different HB subgraph, and not collapsed into single subgraph.
To solve this second problem, COBE will do another pass over the static HB graph to insert a second call-site sensitive context into HB nodes that we refer to as path-context. Path-context of an HB node is implemented as a list of HB nodes from previous HB rules that leading to it. When two HB nodes are logically connected by an HB rule, the path-context of the predecessor HB node is copied as path-context of the successor HB node. Additionally, if the HB rule is either of $\text{Create}(r, n_1) \xrightarrow{M^{rpe}} \text{Begin}(r, n_2)$, $\text{Create}(t) \xrightarrow{T^{fork}} \text{Begin}(t)$, or $\text{Update}(s, n_1) \xrightarrow{M^{push}} \text{Pushed}(s, n_2)$, the id of successor HB node is prepended into path-context of the successor HB node. If two HB nodes with different path-context have happen-before relationships to the same successor HB node, the successor HB node will be duplicated for that two different path. The addition of path-context also helps remove cycle (ie., recursive call) from the HB graph. When inserting the path-context to the HB graph, we check if the destination node already has a path-context inserted. If it does, and it shares the same path-context with the origin node, then we will not connect them with an HB edge, as it will cause a cycle. After this path-context insertion, the HB nodes will be a tuple of $((m, b, h), px)$, with the addition of $px$ as the path-context.

From the final static HB graph, we search for all pairs of memory access nodes $(q,r)$ that do not have happen-before relationships between them (concurrent) and at least one of them is write access (conflicting). To further prune benign pairs from the harmful pairs, COBE will do another filtering to only report pair where either $q$ or $r$ may reach failure instructions, such as an invocation of abort or exit function (e.g., System.exit), through exception throwing or implicit flow.

Note that these $(q,r)$ pairs may be duplicated with each other. Two different pairs $(q_1,r_1)$ and $(q_2,r_2)$ might be differentiated by their different path-context, but $q_1$ and $q_2$ might represent the same program point, as well as both $r_1$ and $r_2$. To remove this duplication, COBE will do further reduction by stripping the context out of $(q,r)$ pairs into a smaller set of unique program point pairs $(s,t)$. This list of $(s,t)$ pairs is what COBE report as bugs for race in master CO pattern. Figure 4.5 shows the high-level rule to detect this pattern.
.decl memAccConflict(node1:PsHbNode,node2:PsHbNode)
memAccConflict(n1,n2) :-
    psHbNode(n1),
    psHbNode(n2),
    n1 != n2,
    n1 = [[[m1,c1],b1,t1],ctx1],
    n2 = [[[m2,c2],b2,t2],ctx2],
    hbNodeMemAccess([[m1,c1],b1,t1],_,accessType1),
    hbNodeMemAccess([[m2,c2],b2,t2],_,accessType2),
    (accessType1 = "memwrite"; accessType2 = "memwrite")
    , (terminateOnException(_,_,_,_,_,m1,b1); terminateOnException(_,_,_,_,_,m2,b2))
    , 'happenBeforeChain(n1,n2),
    'happenBeforeChain(n2,n1).

.decl raceInMaster(t1:symbol,st1:symbol,b1:Cb,t2:symbol,st2:symbol,b2:Cb)
.output raceInMaster
raceInMaster(t1,m1,b1,t2,m2,b2) :-
    memAccConflict([[m1,c1],b1,t1],[[m2,c2],b2,t2]],
    hbNodeStrLessThanOrEqual([[m1,c1],b1,t1],[[m2,c2],b2,t2]).

Figure 4.5: **High level rule for race in master CO pattern.** The bug report \((s,t)\) pair maps into \(((t1,m1,b1),(t2,m2,b2))\) from relation \(raceInMaster\).

### 4.3.4 Transient Network Error Analysis

For ** transient network error** CO pattern, we take two high-level insight. First, an invalid value is obtained during a network communication error. Second, the invalid value is propagated further to the next operation that will, in turn, trigger a system crash or hang. Values can be local variables or class field members. For this pattern analysis, we will focus on class field members.

For the first insight, our first intuition is to do inter-procedural, inter-node, taint analysis to ask what is the possible value definition assigned to a class field member in a certain basic block of a method when there is no conditional path taken. Specifically, we do a traversing over the control-
Figure 4.6: **High level rule for transient network error CO pattern.**

flow graph of a method. If in the control-flow graph, a class field is assigned twice in succession, then querying a possible value of that field at any basic block after the second assignment will return only value definition from the second assignment. But if there are two possible assignments in two different conditional branches, querying a possible value at the end of the two branch will return both value definition as possible values at the queried block.

If a value definition was obtained from a method invocation, we continue our taint analysis to the originating method. If a value definition is passed to the next method invocation, we also continue our taint analysis to that next method invocation. This goes crossing inter-node message communication by analyzing RPC interface calls and permuting the possible destination RPC implementations.

Along with this inter-procedural, inter-node, taint analysis, we also permute one RPC call error. Our analysis note if a particular RPC method call caught a network error, is there any possible field values that retained or overridden after that network error happens. We do this taint analysis for each different RPC method, where we assume a single network error for RPC call to that method.

Now, for the second insight, how do we define value as invalid? And how do we know if that
invalid value will trigger a system crash? This assertion can be different between systems and protocols being tested. For some system, it might be enough to verify whether the value definition is obtained from the local node or remote node [110]. For other system or protocol, a more precise assertion might be needed.

In our initial implementation, we create an assertion to target the DataNode registration protocol in HDFS. HDFS NameNode maintains several global data structures. One of them is DataNode mapping. A valid DataNode registration object must be updated by NameNode, added to this DataNode mapping, and then returned to DataNode for further communication with NameNode. If NameNode can not find a registration object in this data structure, it will raise a remote exception back to DataNode as a response to abort the DataNode. We write Datalog rules to express this DataNode map, its put method and get method. We extend our assertion to verify that a registration object is invalid if it does not reach the put method in NameNode but later on reach the get method through the next RPC call.

COBE will report the invalid value definition, the location where it is first defined, and the failed RPC method that triggers this invalid value definition as one bug report. Figure 4.6 shows the high-level rule for this pattern.

4.4 Evaluation

4.4.1 Methodology

Benchmarks

We evaluate COBE on five problems reported by users in HBASE and HDFS system. HBASE is a distributed NoSQL database, while HDFS is a distributed file system. The description of these five bugs can be seen in Table 4.3.

These five bugs are chosen from our CO bug study listed in Section 2.2.1. Four of them are HBASE bugs containing race in master CO pattern with either order violation (OV) or atomicity
<table>
<thead>
<tr>
<th>Bug ID</th>
<th>CO Pattern</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>hb4539</td>
<td>Race in Master (OV)</td>
<td>Master crash</td>
</tr>
<tr>
<td>hb4729</td>
<td>Race in Master (AV)</td>
<td>Master crash</td>
</tr>
<tr>
<td>hb16367</td>
<td>Race in Master (OV)</td>
<td>Master crash</td>
</tr>
<tr>
<td>hb14536</td>
<td>Race in Master (AV)</td>
<td>Master hang</td>
</tr>
<tr>
<td>hd8995</td>
<td>Transient network error</td>
<td>Worker crash</td>
</tr>
</tbody>
</table>

Table 4.3: **Benchmark bugs.**

violation (AV) root cause. The last one is an HDFS bug with *transient network error* pattern.

Experiment settings

For all bug benchmarks, we configure COBE to focus its analysis only at server level codes. For HBASE, we focus the analysis on package `o.a.h.hbase.master`, `o.a.h.hbase.regionserver`, `o.a.h.hbase.zookeeper`, `o.a.h.hbase.ipc`, and `o.a.h.hbase.executor`, while in HDFS, we focus our analysis on package `o.a.h.hdfs.server` and `o.a.h.hdfs.protocolPB` (*o.a.h* stands for *org.apache.hadoop*).

In each bug analysis, all main methods found in the focused packages are added as entry points in the program parsing phase. For HBASE bugs except `hb16367`, we also add all methods that implement RPC interface between HMaster to RegionServer and HBase Client to HMaster (*o.a.h.ipc.HRegionInterface* and *o.a.h.ipc.HMasterInterface* respectively), ZooKeeper listener interface (*o.a.h.ZooKeeperListener*), and methods extending EventHandler abstract class *o.a.h.executor.EventHandler*. For `hb16367`, since the bug only involves concurrency between HMaster internal thread and does not involve inter-node messaging, we do not add those communication endpoints as additional entry points. Instead, we add harness code from Figure 4.3 as our additional entry point. For `hb14536`, both harness code and additional inter-node messaging entry points. Similarly, for HDFS system (*hd8995*), we add RPC interface between DataNode to NameNode (*o.a.h.hdfs.server.protocol.DatanodeProtocol*) as additional entry points.

For `hbhb4539`, `hbhb4729`, and `hbhb16367` we use the same assertion to capture *race in master* pattern as we explain in Section 4.3.3. However, for `hb14536`, since it is a hang bug, we report all
Table 4.4: **Evaluation result.** The table show result statistics for each bug benchmark. All true positive cases were found by COBE. For *hd8995* query time, the Datalog program was run in compiled mode, while the others was run in interpreter mode.

(s,t) pairs without requiring reachability to failure instructions.

We run our experiments on a single node machine. The machine is installed with Ubuntu 18.04.2, OpenJDK 1.8.0.222, having AMD FX™-4130 Quad-Core CPU and 8GB of RAM. For Datalog analyses, we use Souffle 1.5.1. Both the parsing phase and query phase are run with the same machine.

**Evaluation metric**

We will evaluate COBE by the number of reported bugs and its ability to find the true positive case that reported from the original issues. We will also review the size of extracted program facts, time to run parsing and query. Both parsing and query run time are obtained by averaging measurement from 5 runs. For *race in master* analysis, we will also discuss about the correlation between the reported bug and the resulting static HB graph.

**4.4.2 Bug detection result**

Table 4.4 shows the result of our experiment. In all bug benchmark, COBE was able to capture the true positive case that is reported in the original issue. Analysis for *transient network error* pattern in *hd8995* can show that in method *BPServiceActor.register()* an invalid local variable definition was saved off to field *BPOfferService.bpRegistration* when a network error occurs in RPC call.
DatanodeProtocol.registerDatanode().

Analysis for hb4539, hb4729, and hb14536 for race in master pattern reveals more than one bug report (pairs of \((s,t)\)). For bugs other than what reported in the original issues, we have not yet able to claim whether all of them are false positive or some of them are indeed a true positive that was unknown before.

Ideally, we should able to verify these bug reports by observing the resulting static HB graph, because the HB graph can show us the chain of events that need to happen to lead to the bug. However, the large size of our resulting static HB graph hinders us from doing so. The last two rows of Table 4.4 shows the number of \((q,r)\) pairs before reduced into bug candidate \((r,s)\), and the size of resulting static HB graph in terms of the number of nodes and edges. A large number in these two measures compared with the number of bug candidates indicate that our static HB graph tends to repeat some of the HB subgraphs several times.

There is some explanation as to why this repetition happens in our static HB graph. The first reason is because of our path-context addition to the HB graph. Our main goal to add this second context is to differentiate happen-before relation to the same method but coming from different call path origins. In the conventional \(k\)-cfa sensitivity, only the last \(k\) call-site is saved for distinction comparison. But in our case, we don’t limit the \(k\) yet. Our call-site context able to achieve maximum distinction, but with the cost of growing the HB graph deep. The second reason is we have not added logic in our HB graph building to verify whether a certain path is reachable control-flow wise, considering the path taken in the previous steps. This cause our HB graph to grow wide, as it consider all path from one point of HB node to be a valid-possible call path. Employing some path-refutation algorithm [54] might help us reduce the width of the HB graph.
Figure 4.7: **Minimal static HB graph to reveal hb4539.** Both yellow and cyan node is a memory access to the same location. The yellow nodes can reach failure instructions while the cyan node is not. The two yellow nodes with bold red border is the true positive case reported by the original issue.
In our current iteration, COBE’s static HB graph is more observable and helpful when we exclude some noise constructs from analyses. Figure 4.7 is a minimal static graph to reveal hb4539 from a rerun of the same analysis, but excluding event handler threads and RPC protocols that do not directly involve with RegionServer failover. The left subgraph is the chain of events when the RegionServer signal region has opened, and the right subgraph is the chain of event triggered by that RegionServer terminating shortly after the signaling region opened. The two yellow nodes with a bold red border are the pair of conflicting memory access that can lead to the master node crash. If the bold node in the right subgraph happens before the left bold node, the HMaster will crash.

### 4.4.3 Performance result

Table 4.4 also shows COBE performance for parsing and query time. In terms of parsing time, the speed of parsing is highly dependent on the volume of the program being parsed. For HBASE system, the parsing phase for 5 packages is quite fast, around 30 seconds. For HDFS, the size of programs under our 2 package selection is quite high. Therefore, the parsing time and the extracted program facts relations are higher than HBASE.

In terms of query time, the performance is highly dependent on how efficient we write the Datalog program and whether we run it in interpreter or compiler mode. Souffle support running queries in either interpreter or compiler mode. In interpreter mode, Souffle translates the Datalog program to a RAM program and executes the RAM program on-the-fly. While in compiler mode, Souffle will compile the Datalog program into C++. For computationally intensive Datalog programs, the interpretation is slower than the compilation into C++. However, the interpreter has no costs for compiling a RAM program to C++ and invoking the C++ compiler, which is expensive for large Datalog program.

For race in master analysis, we are getting relatively fast query performance partly due to the static HB graph abstraction that reduces numbers of program facts that we need to be correlated.
On the opposite, for transient network error analysis, our queries are not optimal. We are doing taint analysis for all field members, traversing all basic blocks in every method along the entire call graph. Running transient network error analysis in Souffle interactive mode does not finish after more than 30 minutes. The bigger amount of program facts to analyze also makes the speed worse. Therefore we switch to compiler mode for hd8995 and gain much faster performance compared to interactive mode. On the contrary, we are not able to run transient network error analysis in compiled mode due to compilation error by Souffle in our experiment environment.

In terms of extracted program facts, we get tens to hundreds of MB of data. The large size of data most likely happen due to duplicates string. For example, $Cm$ domains are encoded as a full method reference string. For relation that contain an attribute with domain $Cm$, many of these method reference strings will be repeated. One technique that we do to reduce these repeated strings by making a dictionary relation $dictCm$ that maps integer id with a unique method reference, as shown in Table 4.2. Other relation having lots of rows such as $bbEbbTupleInt$ and $dominateInt$ may refer to method reference through their id. However, we let other relations to keep the $Cm$ domain attributes for ease of debugging.

### 4.5 Related Work

We now discuss some works related to COBE. We contrast them between specific analysis framework (dynamic and static), specific Datalog framework, and general Datalog framework. We also contrast their target system between distributed systems and other specific systems. Table 4.5 show this comparison between COBE and other related works.

**Specific Framework:** FindBugs[77] is one of the popular static analysis tools to capture bugs in Java. It has hundreds of checks to capture many bug patterns including multithreaded correctness but mostly limited for intraprocedural, single-machine applications. Both RacerD[55] and Sierra[79] are static analysis framework targeting race bugs in the Android system. Sierra uses a static HB graph, similar to what COBE use for race in master analysis. While Sierra’s static
HB graph model is based on Android message passing and event handling, COBE model its static HB graph based on inter-node communication such as RPC. RacerD is based on Infer\cite{infer}, thus inherits Infer’s compositional analysis. RacerD compositional granularity is procedure summaries, while COBE composition is program facts and Datalog rules. DCatch\cite{dcatch} and FCatch\cite{fcatch} are both dynamic analysis framework for distributed systems. DCatch target distributed concurrency bugs, while FCatch target time of fault (TOF) bugs. Both DCatch and FCatch works by instrumenting the target system to produce traces and analyze that trace to find bugs. COBE uses some ideas from DCatch in terms of static HB graph building, but runs entirely in static manners and does not relies on traces.

**Specific Datalog:** Datalog has been adopted as the foundation for applied, domain-specific languages in a wide variety of areas. Network Datalog (NDlog)\cite{ndlog} is a language for declarative network specifications. It enables declarative specification and deployment of distributed protocols and algorithms via distributed recursive queries over network graphs. PQL\cite{pql} is a query language that translates into Datalog, aimed to capture errors and security flaws such as SQL injection vulnerabilities. Dedalus\cite{dedalus} is a declarative language that enables a specification of rich distributed system concepts. Dedalus reduces to a subset of Datalog with negation, aggregate functions, successor and choice, and adds an explicit notion of logical time to the language. Bloom\cite{bloom} is a declarative language to build a program that can runs naturally on distributed machines. Bloom programs are bundles of declarative statements about collections of facts or tuples, similar to Datalog rules.

<table>
<thead>
<tr>
<th>Specific System</th>
<th>Specific Framework</th>
<th>Specific Datalog</th>
<th>General Datalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sierra\cite{sierra}, RacerD\cite{racerd}, FindBugs\cite{findbugs}</td>
<td>NDlog\cite{ndlog}, PQL\cite{pql}</td>
<td>EC-Diff\cite{ecdiff}, Chord\cite{chord}</td>
<td></td>
</tr>
<tr>
<td>Distributed System</td>
<td>DCatch\cite{dcatch}, FCatch\cite{fcatch}</td>
<td>Bloom\cite{bloom}, Dedalus\cite{dedalus}</td>
<td>COBE</td>
</tr>
</tbody>
</table>

Table 4.5: **Related Work (COBE).** The table categorizes works that relate to failure analysis, race analysis, and Datalog in the space of program analysis grouped by either targeting specific system or distributed system.
**General Datalog:** EC-Diff[114] is a static analysis for computing synchronization differences of two programs. It leverage Datalog to compute *differentiating* data-flow edges in large multi-threaded C programs. Chord[101] is a static race detection analysis tool for multithread Java programs. Chord detects race in four stages where all four of them are expressed in Datalog language based on bddbddb[122]. Unlike COBE, both EC-Diff and Chord are targeting single-machine applications.

### 4.6 Conclusion

We reveal a new class of outage-causing bugs in distributed systems that we refer to as *cascading outage (CO) bugs*. Specifically, CO bugs 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 do CO bugs study by collecting CO bugs reported in publicly accessible issue repositories of open-source distributed systems and group them by their CO pattern. We present COBE, static program analysis framework to detect CO bugs pattern in distributed systems. We have implemented COBE prototype to detect *race in master* and *transient network error* CO pattern.
CHAPTER 5
OTHER WORKS

In this chapter, I will briefly describe about other works that I have contributed to during my Ph.D. program.

5.1 Why Do the Clouds Stop? Lessons from Hundreds of High-Profile Outages

We conducted a cloud outage study (COS)[70] of 32 popular Internet services. We analyzed 1247 headline news and public post-mortem reports that detail 597 unplanned outages that occurred within a 7-year span from 2009 to 2015. We analyzed outage duration, root causes, impacts, and fix procedures. This study reveals the broader availability landscape of modern cloud services and provides answers to why outages still take place even with pervasive redundancies.

5.2 MittOS: Operating System Supports for Millisecond Tail Tolerance in Data-Parallel Storage

MittOS[73] provides operating system support to cut millisecond-level tail latencies for data-parallel applications. In MittOS, we advocate a new principle that operating system should quickly reject IOs that cannot be promptly served. To achieve this, MittOS exposes a fast rejecting SLO-aware interface wherein applications can provide their SLOs (e.g., IO deadlines). If MittOS predicts that the IO SLOs cannot be met, MittOS will promptly return EBUSY signal, allowing the application to failover (retry) to another less-busy node without waiting. We build MittOS within the storage stack (disk, SSD, and OS cache managements), but the principle is extensible to CPU and runtime memory managements as well. MittOS no-wait approach helps reduce IO completion time up to 35% compared to wait-then-speculate approaches. I contribute in initial implementation of MittOS
to enable application to pass SLA information to kernel IO stack.

5.3 Rivulet: Fault-Tolerant Platform for Smart-Home Applications

Rivulet\[49\] is a fault-tolerant distributed platform for running smart-home applications; it can tolerate failures typical for a home environment (e.g., link losses, network partitions, sensor failures, and device crashes). In contrast to existing cloud-centric solutions, which rely exclusively on a home gateway device, Rivulet leverages redundant smart consumer appliances (e.g., TVs, Refrigerators) to spread sensing and actuation across devices local to the home, and avoids making the Smart-Home Hub a single point of failure. Rivulet ensures event delivery in the presence of link loss, network partitions and other failures in the home, to enable applications with reliable sensing in the case of sensor failures, and event processing in the presence of device crashes. I contribute in this project by building initial implementation of Rivulet delivery service protocol and run experiments in real world home environment. This work is done during my Research Internship at Samsung Research America in 2016.

5.4 Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems

Fail-Slow\[71\] is a hardware that is still running and functional but in a degraded mode, slower than its expected performance. We present a study of 101 reports of fail-slow hardware incidents, collected from 12 institutions large-scale cluster. We show that all hardware types such as disk, SSD, CPU, memory and network components can exhibit performance faults. We made several important observations such as faults convert from one form to another, the cascading root causes and impacts can be long, and fail-slow faults can have varying symptoms. From this study, we make suggestions to vendors, operators, and systems designers. I contribute in this project by collecting and classifying anecdotal data from different institutions.
5.5 ScaleCheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems

ScaleCheck[86, 112] is an approach for discovering scalability bugs (a new class of bug in large storage systems) and for democratizing large-scale testing. ScaleCheck employs a program analysis technique, for finding potential causes of scalability bugs, and a series of colocation techniques, for testing implementation code at real scales but doing so on just a commodity PC. ScaleCheck has been integrated to several large-scale storage systems, Cassandra, HDFS, Riak, and Voldemort, and successfully exposed known and unknown scalability bugs, up to 512-node scale on a 16-core PC. I contribute my expertise in HDFS by implementing the ScaleCheck approach to find and verify scalability bugs on HDFS system.

5.6 FlyMC: Highly Scalable Testing for Complex Interleavings in Cloud Systems

FlyMC[92] is a fast and scalable testing approach for datacenter/cloud systems such as Cassandra, Hadoop, Spark, and ZooKeeper. The uniqueness of our approach is in its ability to overcome the path/state-space explosion problem in testing workloads with complex interleavings of messages and faults. We introduce three powerful algorithms: state symmetry, event independence, and parallel flips, which collectively makes our approach on average 16x (up to 78x) faster than other state-of-the-art solutions. We have integrated our techniques with 8 popular datacenter systems, successfully reproduced 12 old bugs, and found 10 new bugs all were done without random walks or manual checkpoints. I contribute my expertise in HDFS by running FlyMC experiments to detect distributed concurrency bugs in HDFS system.
CHAPTER 6
CONCLUSIONS AND FUTURE WORK

In this dissertation, we aim to mitigate disruptive cascading effects in cloud-scale distributed systems that disturb the performance and availability of the systems. For the performance aspect, we focus on improving the tail tolerance of data-parallel frameworks. For the availability aspect, we focus on eliminating cascading outage bugs. This chapter concludes this dissertation work and discusses future work in combating cascading performance failure and cascading outage bugs.

6.1 Conclusion

6.1.1 Cascading Performance Failure

The first problem we focus on in this dissertation is cascading performance failure. 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: 1) no-straggler-detected; and 2) straggling-backup. These loopholes exist due to two flaws. First, node-level network degradation is not incorporated as a fault model. Second, path progresses of a task are lumped into one task progress score. We also show how performance problems caused by these loopholes can cascade to an 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. With path progress, we increase information exposure from the storage layers to compute layers. With path diversity, we prevent potential tail-SPOF by enforcing independent, comparable paths. And lastly, we do path-straggler detection and speculation by comparing multiple reported path progresses and pinpoint the faulty node/NIC. We have implemented PBSE in Hadoop/HDFS system. PBSE can deliver 1.5-70x speedups compared to base SE.
6.1.2 Cascading Outage Bugs

The second problem we focus on in this dissertation is cascading outage (CO) bugs. We believe that the system code itself has emerged as a new single point of failure, which leads to the occurrence of CO bugs. To better understand the taxonomy of CO bugs, we begin our study by collecting samples of CO bugs from publicly accessible issue repositories of open-source distributed systems. Our bug study gathered 68 CO bugs and categorized them into 11 CO pattern categories.

The complexity and variety of CO patterns from our bug study highlight the need to build a program analysis framework that is highly expressive and composable to detect them. We present COBE, static program analysis framework to detect CO bugs pattern in distributed systems. COBE is a program analysis stack that combines several program analyses to reveal certain patterns of CO bugs. COBE achieves this by extracting program facts such as class hierarchies, crash paths, recovery paths, and so on, from the target system binaries and correlate between them to reveal CO pattern. We design COBE analyses stack into three layers: 1) Program facts layer; 2) Program analysis libraries layer; 3) and high-level CO pattern query layer. We achieve high expressivity and composability by writing our program analyses in the form of Datalog program. We have implemented COBE with analyses to detect race in master and transient network error CO patterns and were able to detect five previously known CO bugs.

6.2 Future Work

6.2.1 PBSE

We have shown the generality of PBSE by integrating them into HDFS, Apache Spark, Flume, and Quantcast File System. As distributed systems continue to evolve, we believe that PBSE concepts are still relevant for many other data-parallel frameworks, even beyond. MittOS[73], for example, applies a similar technique as PBSE to achieve millisecond tail tolerance in the operating system level.
Since we publish PBSE, we have generalized the problems of degraded-network tail problems under new broader terminology, Fail-Slow[71]. PBSE is an example solution to tackle a Fail-Slow problem caused by degraded-network. But as the Fail-Slow paper describe, many other hardware problems can lead to cascading performance failures. Future works can take account of these Fail-Slow hardware cases that are mostly still overlooked. IASO[106] is an example of work that detects Fail-Slow problem in a hyperconverged system.

In the limitation section of PBSE, we mention that PBSE may not perform well in a virtualized environment such as public cloud environments. PBSE supposedly can work if the VMs are deployed across many machines and they expose the machine number. Future works can explore this possibility, for example, by adding such interface in OpenStack[18] such that it can expose those placement and topology information back to VMs and distributed system running on top of it. Then, we also have multi-cloud and hybrid environments to further explore PBSE applicability.

### 6.2.2 COBE

We have implemented COBE to detect the existence of two CO patterns. However, there are still several other CO patterns that we have not implemented yet. Future work can aim to add more high-level CO queries to capture other CO patterns. In terms of the CO patterns itself, there are still many CO bugs pattern that we have not yet uncover, especially in a different type of distributed system architecture.

COBE still depends on human experts to add domain knowledge relations such as which class type is used for inter-node communications, which methods constitute as memory access, and so on. While we believe static program analysis over distributed systems will always need some degree of domain knowledge to help guide the analysis, future works can aim to reduce this dependency. This can be done by writing more careful or exhaustive rules, or by another automatic mechanism.

COBE can also benefit more by adding fault injection tools. For all CO vulnerabilities found in
the analysis phase, we can confirm it by running the actual system with targeted fault injection. If no outage happens after fault injected runs, we can assume that the vulnerability is benign or false positive. Only vulnerabilities that proven to crash the system are reported back to programmers.

Currently, COBE saves the extracted program facts into a set of *tab-separated value (tsv)* files. We envision that the next iteration of COBE will have an online Datalog-based database as the backend. Having an online database backend will enable us to do more creative online analyses, such as updating program properties as the actual program runs and doing analyses on-the-fly. Having full-fledged database backend will also enable us to create *program lake*, a repository of program facts of multiple distributed systems. Distributed systems often build in layers and have tight interactions with other distributed systems on different layers. A program lake will allow us to also analyze for CO vulnerabilities that span across different systems.

Lastly, another direction for future works is to augment CO containment ability into the software system itself. As major failure sometimes is inevitable, we need software systems to have capabilities of containing the cascading nature of CO bugs in a live deployment. Below are some of the containment principles that can be developed and integrated into software 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 gradually and leave some “trails” (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). The challenge is to develop domain-specific containment strategies (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 (e.g., skipping a buggy load balancer may cause unintended backlogs in some nodes), which must be taken care of properly.
REFERENCES

[17] Messaging app Telegram added 5m new users the day after WhatsApp outage (link), February 24, 2014.
[19] Personal communication from andree jacobson (chief information officer at new mexico consortium).
[20] Personal communication from datacenter operators of university of chicago it services.
[22] Personal Communication from Gary Grider and Parks Fields of Los Alamos National Laboratory.

[23] Personal Communication from H. Birali Runesha (Director of Research Computing Center, University of Chicago).


[25] Personal communication from robert ricci of university of utah.


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


[38] Xbox: Microsoft Azure and Xbox Live Services Experiencing Outages (link), November 19, 2014.


[92] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa,


