

ABSTRACT

HE, JINGZHU. Automatically Fixing Performance Bugs and Extracting Bug Signatures for Cloud Systems. (Under the direction of Dr. Xiaohui (Helen) Gu.)

Cloud systems are becoming increasingly complex and performance bugs are inevitable. Performance bugs are notoriously difficult to debug and fix due to lack of diagnostic information. In this dissertation, we investigate the problems of performance bug detection, fixing and signature extraction.

First, we use the timeout bug as a case study of performance bug detection. We present TScope, an automatic timeout bug identification tool for cloud systems. TScope leverages kernel-level system call tracing and machine learning based anomaly detection and feature extraction schemes to achieve timeout bug identification. TScope introduces a unique system call selection scheme to achieve higher accuracy than existing generic performance bug detection tools. We have implemented a prototype of TScope and conducted extensive experiments using 19 real-world performance bugs, including 12 timeout bugs and 7 non-timeout performance bugs. The experimental results show that TScope correctly classifies 18 out of 19 bugs. Compared to existing runtime bug detection schemes, TScope reduces the average false positive rate from 47.24% to 0.8%. TScope is light-weight and does not require application instrumentation, which makes it practical for production server performance bug identification.

Second, we present our work on fixing timeout bugs caused by misconfigured variables, and hang bugs caused by infinite loops and blocking operations. We develop TFix and HangFix respectively. TFix is a tool that automatically fixes misused timeout bugs with proper timeout configurations in cloud systems. TFix adopts a drill-down bug analysis protocol that can narrow down the root cause of a timeout bug and produce configuration recommendations for correcting the root cause. TFix first employs a system call frequent episode mining scheme to check whether a timeout bug is caused by a misused timeout variable. TFix combines application tracing and static taint analysis to pinpoint the misused timeout variable. TFix produces recommendations for proper timeout variable values based on the tracing results during normal runs. We have implemented a prototype of TFix and conducted extensive experiments using 13 real world timeout bugs. Our experimental results show that TFix can correctly localize the misused timeout variables and suggest proper timeout values for fixing those bugs. HangFix is a tool that automatically fixes a hang bug that is triggered and detected in production cloud environments. HangFix first leverages stack trace analysis to localize the hang function and then performs root cause pattern matching to classify hang bugs into different types based on likely root causes. Next, HangFix generates effective code patches based on the identified root cause patterns. We have conducted an empirical study over 237 real production hang bugs to quantify the generality of our root cause patterns. Our empirical bug study shows that our likely root cause patterns cover 76% of 237 hang bugs. HangFix can correct the hang symptom for all of those matched bugs and completely fix the root cause for 75% of them. We have

implemented a prototype of HangFix and evaluated the system on 42 real-world software hang bugs in 10 commonly used cloud server applications. Our results show that HangFix can successfully fix 40 out of 42 hang bugs in seconds.

Lastly, we present PerfSig, a multi-modality performance bug signature extraction tool which can identify principal anomaly patterns and root cause functions for performance bugs. PerfSig performs fine-grained anomaly detection over various machine data such as system metrics, system logs, and function call traces. We then conduct causal analysis across different machine data using information theory method to pinpoint the root cause function of a performance bug. PerfSig generates bug signatures as the combination of the identified anomaly patterns and root cause functions. We have implemented a prototype of PerfSig and conducted evaluation using 20 real world performance bugs in six commonly used cloud systems. Our experimental results show that PerfSig captures various kinds of fine-grained anomaly patterns from different machine data and successfully identifies the root cause functions through multi-modality causal analysis for 19 out of 20 tested performance bugs.

© Copyright 2021 by Jingzhu He

All Rights Reserved

Automatically Fixing Performance Bugs and Extracting Bug Signatures
for Cloud Systems

by
Jingzhu He

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2021

APPROVED BY:

Dr. Guoliang Jin

Dr. Christopher Parnin

Dr. Huiyang Zhou

Dr. Xiaohui (Helen) Gu
Chair of Advisory Committee

DEDICATION

To my family and friends.

BIOGRAPHY

Jingzhu He received her Bachelor degree from Nanjing University in 2013 and M.Phil. degree from Hong Kong Baptist University in 2016 respectively. At North Carolina State University, she worked under the direction of Dr. Xiaohui (Helen) Gu. Her research interest lies in software system reliability with a focus on automatically detecting, diagnosing and fixing performance bugs on cloud systems.

ACKNOWLEDGEMENTS

This dissertation could not be completed without the encouragement and support of several people, who I would like to acknowledge as follows.

First and foremost, I would like to express my deep gratitude to my advisor Dr. Xiaohui (Helen) Gu. Her deep knowledge of cloud computing and passion about research gave me strong inspiration to work hard on my research projects. She not only guided me to conduct high-quality research, but also taught me how to present my ideas and communicate with other researchers. I truly enjoyed working with her and I will recall the experience in the future.

I would like to thank my committee members, Dr. Guoliang Jin, Dr. Christopher Parnin and Dr. Huiyang Zhou, for their insightful suggestions on my dissertation. In particular, I would like to thank Dr. Guoliang Jin for the guidance in our cooperated project and the help in my faculty application.

I would next like to thank several current and former labmates, Dr. Ting Dai, Fogo Tunde-Onadele and Yuhang Lin, for their precious friendship and help. I would like to thank my friends in our department, Rui Shu, Tianpei Xia, Xi Yang and Shudi Shao for their precious friendship and encouragement.

I would like to thank the National Science Foundation for the funding they have provided which makes my research possible.

Last but not least, I would like to thank all my family and friends for their endless love and support.

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	viii
Chapter 1 INTRODUCTION	1
1.1 Motivation	1
1.2 Summary of the State of the Art	2
1.3 Thesis Statement	2
1.3.1 Research Challenges	3
1.4 Summary of Contributions	3
Chapter 2 AUTOMATIC PERFORMANCE BUG DETECTION FOR CLOUD SYSTEMS	5
2.1 Introduction	5
2.1.1 Motivating Example	5
2.1.2 Our Contribution	6
2.2 System Design	8
2.2.1 Approach Overview	8
2.2.2 System Call Selection	9
2.2.3 System Call Feature Extraction	10
2.2.4 Timeout Bug Identification	11
2.3 Experiment Evaluation	12
2.3.1 Evaluation Methodology	12
2.3.2 Results Analysis	17
2.3.3 Case Study	22
2.4 Summary	23
Chapter 3 AUTOMATIC PERFORMANCE BUG FIXING FOR CLOUD SYSTEMS	24
3.1 Introduction	24
3.1.1 A Motivating Example	25
3.1.2 Contribution	25
3.2 Fixing Timeout Bugs Caused by Misconfigured Variables	27
3.2.1 Approach Overview	27
3.2.2 Misused Timeout Bug Classification	28
3.2.3 Timeout Affected Function Identification	29
3.2.4 Misused Timeout Variable Identification	31
3.2.5 Timeout Value Recommendation	32
3.3 Experimental Evaluation for Fixing Timeout Bugs	32
3.3.1 Methodology	33
3.3.2 Results	35
3.3.3 Overhead	39
3.3.4 Case Study	40
3.4 Fixing Hang Bugs Caused by Infinite Loops and Blocking Operations	41
3.4.1 Hang Function Localization	41
3.4.2 Root Cause Pattern Matching and Patch Generation	43
3.4.3 Discussion	50
3.5 Experimental Evaluation for Fixing Hang Bugs	51

3.5.1	Evaluation Methodology	51
3.5.2	Empirical Study Results	54
3.5.3	Experimental Results	55
3.6	Summary	58
Chapter 4	EXTRACTING PERFORMANCE BUG SIGNATURES VIA MULTI-MODALITY ANALYSIS	61
4.1	Introduction	61
4.1.1	A Motivating Example	63
4.1.2	Contribution	63
4.2	System Design	64
4.2.1	Approach Overview	64
4.2.2	System Metric Anomaly Pattern Detection	65
4.2.3	System Log Anomaly Pattern Detection	67
4.2.4	Root Cause Analysis via Multi-modality Causal Analysis	69
4.3	Experimental Evaluation	73
4.3.1	Evaluation Methodology	73
4.3.2	Implementation	75
4.3.3	Alternative Approaches	75
4.3.4	Results	76
4.3.5	Overhead and Diagnosis Time	81
4.3.6	Case Studies	81
4.4	Summary	82
Chapter 5	RELATED WORK	83
5.1	Performance Bug Detection and Diagnosis	83
5.2	Static Bug Detection and Diagnosis	84
5.3	Dynamic Bug Detection and Diagnosis	84
5.4	Configuration Bug Detection and Diagnosis	85
5.5	Automatic Bug Fix	85
5.6	Single-modality Data Analysis	86
5.7	Causal Analysis	86
Chapter 6	CONCLUSION AND FUTURE WORK	88
6.1	Contributions	88
6.2	Future Work	89
	BIBLIOGRAPHY	91

LIST OF TABLES

Table 2.1	An example of how to calculate time vectors from processed system call lists.	10
Table 2.2	System description.	13
Table 2.3	Timeout bugs' description.	14
Table 2.4	Non-timeout performance bugs' description.	15
Table 2.5	Detection result of TScope and four alternative approaches for timeout bugs.	17
Table 2.6	Classification result for the 19 bugs. No. 1 to 12 are timeout bugs, while No. 13 to 19 are non-timeout performance bugs. ✓ means the bug is identified as a timeout (non-timeout) bug and it is indeed a timeout (non-timeout) bug. ✗ means the bug is identified as a timeout (non-timeout) bug but it is a non-timeout (timeout) bug.	20
Table 2.7	Computation time of TScope on timeout bugs.	22
Table 3.1	System description.	33
Table 3.2	Timeout bug benchmarks.	34
Table 3.3	TFix's classification result of timeout bugs.	36
Table 3.4	The timeout affected functions.	37
Table 3.5	The fixing result of TFix.	38
Table 3.6	The runtime overhead of TFix.	40
Table 3.7	42 reproducible hang bug benchmarks. Although some bugs have the same description, they happen in different functions or classes.	56
Table 3.8	The comparison of HangFix and manual fixing.	60
Table 4.1	Bug benchmark. bug ID* represents it is a distributed system performance bug.	74
Table 4.2	Signature extraction results for bugs which have system metric anomalies. Pearson ^f is pearson method with filter. Similarly, Spearman ^f is spearman method with filter. For each method, we present the root cause function's rank using causal analysis.	77
Table 4.3	Signature extraction results for bugs which have system log anomalies. ✗ represents the anomaly pattern and the root cause function cannot be identified.	79
Table 4.4	The runtime overhead and diagnosis time. Hadoop represents the four Hadoop system, i.e., Hadoop Common, Hadoop MapReduce, Hadoop HDFS and Hadoop Yarn.	80

LIST OF FIGURES

Figure 2.1	Root cause of Hadoop-11252 bug. (The DataNodes miss timeout on RPC connection, leading to system hanging. Hadoop system reports no error message.)	6
Figure 2.2	The overall architecture of TScope.	8
Figure 2.3	System call trace example collected by LTTng.	9
Figure 2.4	False positive rate of TScope and four alternative approaches.	18
Figure 3.1	When the blocks are corrupted, the <code>recoverLease()</code> function keeps polling the recovery results, getting “false” and sending a new recovery request, hanging in an infinite loop. “+” means added code, representing the manual patch for this bug.	25
Figure 3.2	The architecture of TFix.	27
Figure 3.3	A web search example.	29
Figure 3.4	The Dapper trace.	29
Figure 3.5	A trace example of Dapper.	30
Figure 3.6	TFix uses the static taint analysis to identify the misused timeout variable for the HDFS-4301 bug.	31
Figure 3.7	The MapReduce-6263 timeout bug. The ApplicationMaster is forcefully killed, losing all job history after the bug is triggered. The root cause of this bug is the too small timeout value for killing job request sent from YarnRunner to the ApplicationMaster.	41
Figure 3.8	The system architecture of HangFix.	42
Figure 3.9	Continuously dumped stack traces of <code>main</code> thread for the Compress-451 hang bug.	42
Figure 3.10	Example of hang bug pattern #1 and its fixing strategy. When <code>InputStream dis</code> is inaccessible or corrupted by bad encoding <code>dis.skip</code> can return -1 or 0, and -1/0 is used as the stride. “→” represents the function call invocation. “-” means deleted code and “+” means added code, representing the patch generated by HangFix.	43
Figure 3.11	When the blocks are corrupted, the <code>recoverLease()</code> function keeps polling the recovery results, getting “false” and sending a new recovery request, hanging in an infinite loop. “+” means added code, representing HangFix’s patch for this bug.	44
Figure 3.12	Example of hang bug pattern #2 and its fixing strategy. Misconfiguration causes <code>bufferSize</code> to be 0, which in turn makes the <code>InputStream in</code> perform <code>read</code> operation on a zero-size byte array and return 0. “→” represents the function call invocation, while “→” represents the data dependency flow. “+” means added code, representing the patch generated by HangFix.	46
Figure 3.13	Example of hang bug pattern #3 and its fixing strategy. Data corruption causes <code>readWithShortLength()</code> to throw exception at line #130-131, which makes the loop skip the index updating statement (i.e., zero-stride) at line #134. “→” represents the function call invocation, while “→” represents the control flow. “-” means deleted code and “+” means added code, representing the patch generated by HangFix.	47

Figure 3.14	Example of hang bug pattern #4 and its fixing strategy. <code>Inflater.inflate()</code> is a blocking-pone function. When an ORC file is corrupted, conducting the <code>inflate()</code> operation on a corrupted file causes an infinite loop in the underlying JNI code. “→” represents the function call invocation. “-” means deleted code and “+” means added code, representing the patch generated by HangFix.	49
Figure 3.15	The pattern matching and fixing results of the 237 hang bugs in our empirical study. 76% of them fall into HangFix’s four root cause patterns. 75% of them can be completely fixed.	53
Figure 3.16	Example of a pattern #3 hang bug which cannot be fixed by HangFix. A corrupted file <code>f</code> associated with the lease path <code>p</code> makes the <code>internalReleaseLease</code> function fail for recovering the lease for <code>f</code> . When it happens, <code>p</code> is not removed from <code>sortedLeases</code> (skip updating loop index), <code>LeaseManager</code> keeps recovering lease for the file <code>f</code> endlessly. “→” represents the function call invocation, while “-->” represents the control flow. “+” means added code, representing the patch generated by HangFix.	58
Figure 4.1	The code snippet of the Hadoop-11252 Bug. The buggy code is invoked at the <code>DataNode</code>	62
Figure 4.2	Logs generated by Hadoop-11252 bug. The logs are produced at the <code>NameNode</code>	63
Figure 4.3	The architecture overview of <code>PerfSig</code>	64
Figure 4.4	Three commonly seen system metric anomaly patterns.	66
Figure 4.5	The log entry classification framework.	68
Figure 4.6	The log entries’ embedding vectors forms two clusters, one for each task. . .	68
Figure 4.7	Logs generated by HDFS-4301 bug.	70
Figure 4.8	The extracted function call traces. “r” represents the function name, “b” represents function’s start timestamp and “e” represents function’s end timestamp. All the traces have the start time no earlier than 00:02:54 and end time no later than 00:03:30.	70
Figure 4.9	Hadoop-6133 bug’s time series.	71
Figure 4.10	Hadoop-11252 bug’s time series.	73
Figure 4.11	The code snippet of the Hadoop-6133 Bug.	81
Figure 4.12	Logs generated by HDFS-7005 bug.	82

CHAPTER

1

INTRODUCTION

1.1 Motivation

Cloud server systems (e.g., Hadoop [Hada], Cassandra [Cas], HDFS [Hdf], Spark [Spa]) have become increasingly complex, which often consist of many inter-dependent components. It is challenging to achieve reliability in cloud server systems because 1) different components need to communicate frequently with each other via unreliable networks and 2) individual component may fail at any time. Performance bugs are very common and they impede users to migrate the applications onto the cloud. Compared with functionality bugs which cause system crashing immediately, performance bugs often cause system hanging or slowdown. They are difficult to identify and fix, because the system often does not produce any error messages when the bugs happen. Previous work focuses on generic performance bug detection [Dea14; Dea16]. However, performance bugs cannot be truly resolved until the root causes of the bugs are pinpointed and further correctly fixed. Otherwise, the bugs will happen again when the bug-triggering conditions are met again in the cloud system. Therefore, it is essential to develop a framework for diagnosing and fixing performance bugs.

During our empirical bug study using popular bug repositories such as Jira and Bugzilla [Jir; Bug], we observe that many performance bugs repeatedly occur in different versions of open source systems, which causes the community to perform redundant debugging over the same bug. Moreover, micro-services using containers [Doc] make the bug replication easier than ever – the same bug occurs in multiple containers that are created from the same container image. To this end, developing an automatic tool to extract signatures for different performance bugs can help operators identify repeatedly occurring bugs and provide useful diagnostic information for developers.

1.2 Summary of the State of the Art

Existing work has been done on generic performance bug detection and diagnosis [Att12; Dea14; Bar04; Ste10; FD10]. They are divided into two categories, that are program analysis based methods [Yu16a; Zha17] and machine learning based methods [Vot17; Coh04]. These tools are useful to detect generic performance bugs. However, these tools cannot identify the root causes of bugs. Once a performance alert is generated, our work can identify the root cause and automatically fix it.

Much work has been done on automatic bug fixing [Jin12b; Jin11; Per09; Tuf18]. These approaches either focused on functional bugs, e.g., concurrency bugs, or provided fixes based on existing patches of similar bugs. Therefore, those approaches cannot fix performance bugs directly, or their effectiveness need to be improved in the case that they do not consider the performance bug characteristics. Compared to the existing work, our work focuses on providing effective solutions to fix two types of performance bugs, i.e., timeout bugs and hang bugs.

Previous work [Coh05; Coh04; Dea14; Aru13] mainly focuses on depicting performance bugs via analysis over single data type such as system metrics, system calls, system logs, or performance counters. However, a performance bug may manifest as anomalies in different data types. For example, an infinite loop bug may cause a persistently high CPU usage while a timeout bugs can cause abnormal log sequences. Thus, it is likely that we may fail to extract bug signatures for some performance bugs if we only focus on analyzing one data type. Moreover, extracting anomalies alone often cannot uniquely characterize a performance bug because different performance bugs may exhibit similar anomaly patterns in one data modality. For example, different infinite loop bugs can all show increased CPU consumption. Compared to the existing work, our work conducts multi-modality analysis to extract performance bug signatures.

1.3 Thesis Statement

The goal of this dissertation is to understand the existing tools in performance bug detection, fixing and signature extraction and to propose new techniques for advancing the state of the art. The discoveries and findings made during our studies formulate the following thesis statement:

Leveraging both a program's runtime execution information (e.g., system metrics, system logs, function call traces, system call traces) and the program's control-flow/data-flow graph can enable effective automatic performance bug detection, fixing, and signature extraction in production cloud systems.

Parts of this thesis statement are presented in each of our work. We believe it is possible to build a holistic framework to automate performance bug detection, fixing, and signature extraction with low runtime overhead.

The applications we have studied are all real world bugs that have occurred in the cloud. The software bugs we have studied all commonly occur in the cloud. Performance bugs usually consist of two categories: slowdown bugs and hang bugs. In the first piece of our work, we use the timeout

bugs that usually cause system slowdown or hanging as a case study and develop a timeout bug detection tool. In the second piece of our work, we develop two tools to fix the timeout bugs and hang bugs, respectively. In the third piece of our work, we develop a generic tool to extract bug signatures for both hang bugs and slowdown bugs.

1.3.1 Research Challenges

Developing a practical and effective performance bug fixing and signature extraction system requires overcoming the following research challenges:

- **Domain agnostic:** Cloud environments often involve multiple parties where infrastructure, distributed platforms (e.g., MapReduce, Cassandra and HDFS), and applications are developed by different parties. Therefore, it is essential to design domain-agnostic techniques that can be applied to different cloud systems without requiring domain knowledge.
- **Runtime operation:** Due to the severe financial cost performance bugs cause, it is desirable to detect and fix them as quickly as possible. As a result, any tools should be operated during system runtime and they are expected to generate results in seconds or minutes.
- **Low overhead:** Typically, a cloud system has strict performance requirements on tools running inside it. Interacting with applications running in the cloud must be done with a bare-minimum of overhead to those applications.

1.4 Summary of Contributions

In this dissertation, we make the following contributions:

- We present TScope, an automatic runtime timeout bug identification system. TScope combines a unique system call selection scheme with unsupervised machine learning based anomaly detection to achieve higher detection accuracy and precise timeout bug identification. TScope is light-weight and does not require application instrumentation. We have implemented a prototype of TScope and conducted experiments on 19 real-world performance bugs, including 12 timeout bugs and 7 non-timeout performance bugs. The experimental results show that TScope correctly classifies 18 out of 19 bugs. Compared to existing runtime bug detection schemes, TScope reduces the average false positive rate from 47.24% to 0.8%.
- We present TFix, a tool that automatically fixes timeout bugs with proper configurations in cloud systems. TFix adopts a *drill-down bug analysis protocol* that can narrow down the root cause of a timeout bug and produce configuration recommendations for correcting the root cause. TFix employs a system call frequent episode mining scheme to check whether a timeout bug is caused by a misused timeout variable. TFix combines dynamic application tracing and static taint analysis to pinpoint the misused timeout variable. TFix performs timeout

variable value recommendation based on the profiled execution time of the pinpointed function during normal runs. We have implemented a prototype of TFix and conducted extensive evaluation using 13 real world timeout bugs. Our results show that TFix can correctly classify all tested misused timeout bugs and pinpoint the exact timeout variable that has caused the timeout bug. The timeout values suggested by TFix can effectively correct all the tested misused timeout bugs.

- We present HangFix, a domain-agnostic, byte-code-based software hang bug fixing tool. HangFix first leverages stack trace analysis to localize the hang function and then performs root cause pattern matching to classify hang bugs into different types based on likely root causes. Next, HangFix generates effective code patches based on the identified root cause patterns. Our empirical bug study shows that our likely root cause patterns cover 76% of 237 hang bugs. HangFix can correct the hang symptom for all of those matched bugs and completely fix the root cause for 75% of them. We have implemented a prototype of HangFix and evaluated our system using 42 reproduced real-world software hang bugs in 10 commonly used cloud systems. HangFix successfully fixes 40 of them in seconds.
- We present PerSig, a new multi-modality performance bug signature extraction framework. PerfSig first employs signal processing techniques and unsupervised machine learning methods to identify fine-grained anomaly patterns in various machine data. Next, PerfSig performs causal analysis between abnormal metric/log patterns and function call traces using information theory method mutual information (MI) [Liz14]. The goal is to identify the root cause function which is the top contributor to the metric or log anomaly. We have implemented a prototype of PerfSig and evaluated it over 20 real-world bugs that are discovered in six commonly used cloud systems. The results show that PerfSig can produce precise signatures for 19 out of 20 performance bugs.

This dissertation is organized as follows. Chapter 2 describes the performance bug detection framework. Chapter 3 presents the performance bug fixing framework. Chapter 4 presents the performance bug signature extraction framework. Chapter 5 discusses the related work. Finally, Chapter 6 concludes our discussion and describes the future directions for our work.

CHAPTER

2

AUTOMATIC PERFORMANCE BUG DETECTION FOR CLOUD SYSTEMS

In this chapter, we present our bug detection framework. We use the timeout bug as a case study and develop an tool named TScope. Our tool adopts a unique system call selection scheme and an unsupervised machine learning based anomaly detection method. We give an introduction of TScope and then present the design and experimental evaluation details.

2.1 Introduction

Timeout is commonly used as a failover mechanism in complex server systems. For example, when a server component s_1 sends a request to another component s_2 , s_1 can use the timeout mechanism to avoid infinite waiting in case s_2 fails to respond. However, recent studies [Gun14; Hua15a; Dai18b] show that timeout bugs widely exist in real world server systems and can cause server to hang or slowdown. As an example of real world production system failure, a timeout bug caused the Amazon DynamoDB server to experience a five-hour service outage in 2015 [Dyn]. Furthermore, our previous detailed timeout bug study [Dai18b] shows that 80% timeout bugs produce no error message or misleading error messages, which makes them difficult to identify.

2.1.1 Motivating Example

We use Hadoop-11252 [Hadb] as a motivating example to illustrate how timeout bugs happen. This bug occurs in Hadoop common, a standard utility library for configuring Hadoop cluster. This

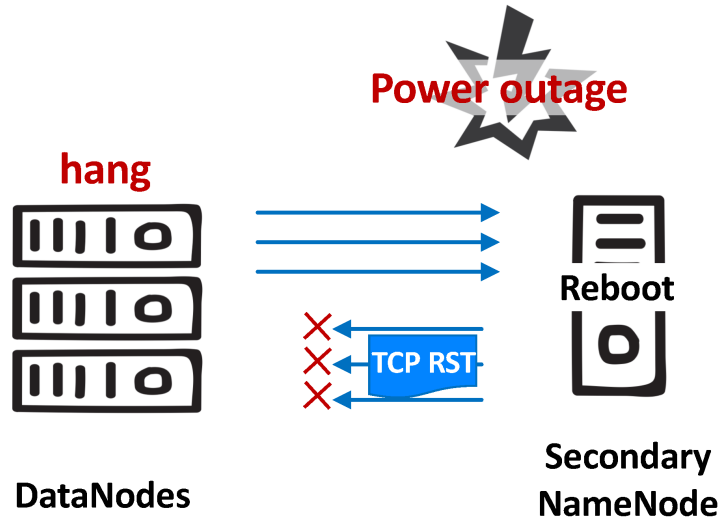


Figure 2.1 Root cause of Hadoop-11252 bug. (The DataNodes miss timeout on RPC connection, leading to system hanging. Hadoop system reports no error message.)

timeout bug is caused by missing timeout checking for the remote procedure call (RPC) connection between different nodes. In Hadoop clusters, nodes communicate with each other using RPC connections. In the bug shown by Figure 2.1, when the RPC connection from the Secondary NameNode to the DataNode is broken, the Secondary NameNode should send a TCP reset message (i.e., the RST message) to inform the DataNode to close the connection. However, in the case of a power outage occurred on the Secondary NameNode, the RST messages get lost and the DataNode keeps its connection open forever and waits endlessly for the response from the Secondary NameNode even after the Secondary NameNode recovers from the power outage and reboots. As a result, the whole Hadoop system hangs and no error message is produced. To fix the bug, the developer added a one-minute timeout on the RPC connection from the DataNode to the Secondary NameNode. So the DataNode will close the RPC connection after it waits for a response from the Secondary NameNode for one minute. After timeout, the DataNode reconnects to the Secondary NameNode by establishing a new RPC connection with the Secondary NameNode after it recovers from the power outage.

2.1.2 Our Contribution

In this paper, we present TScope, a runtime timeout bug identification tool for detecting and classifying timeout bugs in server systems. When a server system experiences software hang or performance slowdown¹, TScope is triggered to identify whether the server performance anomaly is caused by a timeout bug. To be practical for production servers, TScope leverages kernel level system call tracing [DD06] to collect runtime system behaviors and performs bug identification based on the system call traces only. As a result, TScope does not require application source code or

¹Those performance anomalies can be detected by various online anomaly detection tools (e.g., [Dea12; Tan12])

any application instrumentation.

Different from previous generic production server performance bug detection tools [Dea14], TScope focuses on identifying timeout bugs for higher bug identification precision. To do so, TScope first introduces a unique feature selection scheme that performs filtering on the system call trace based on whether the system call is related to the timeout problem. TScope decides whether a system call is related to timeout based on three criteria: 1) a system call includes timeout related parameters (e.g., `timeout_msecs` in `sys_poll` system call); 2) a system call is related to network or synchronization; or 3) a system call is used in timeout configuration functions. The rationale for the first selection criteria is intuitive since a system call including a timeout related parameter will be likely invoked in the timeout mechanism with a high chance. The second selection criteria is derived from our previous timeout bug study [Dai18b] where we found many timeout bugs are triggered during network or synchronization operations. This observation is also expected since timeout is used for handling failures during inter-component communications or coordinations. The third criteria allows us to include those system calls that are used by timeout operations but not necessarily include timeout related parameters.

After selecting timeout related system calls, TScope extracts a list of feature vectors consisting of total execution time values of different system call types during each sampling interval (e.g., 1 second) from the raw system call traces. The execution time value of each system call type (e.g., `sys_poll`) reflects how long the system call gets executed during the sampling interval. TScope then performs multivariate anomaly detection using an unsupervised behavior learning method [Dea12]. If any anomalies are detected, TScope then check whether those anomalies involve any system calls that include timeout related parameters. If the detected anomalies indeed include those timeout specific system calls, TScope infers the detected bug is a timeout related bug. Specifically, this paper makes the following contributions.

- We present a specialized timeout bug identification tool that combines timeout related feature selection and unsupervised machine learning based anomaly detection to achieve higher detection accuracy and more precise bug identification than previous generic bug detection tools.
- We introduce a timeout related system call selection scheme that comprehensively considers the system call parameters, where the system calls are invoked (e.g., network/synchronization operations), and when the system calls are invoked (e.g., during timeout configuration) to cover all the system calls that might be related to timeout bug identification.
- We have implemented a prototype of TScope and conducted extensive evaluation using 19 real-world performance bugs (12 timeout bugs and 7 non-timeout bugs) reported on 10 popular server systems. Our results show that TScope provides correct timeout bug identifications for 18 out of 19 performance bugs while 17 out of those 19 performance bugs produce no error message or misleading error messages. Compared to existing generic performance bug detection tools (e.g., PerfScope [Dea14]), TScope can reduce the average false positive

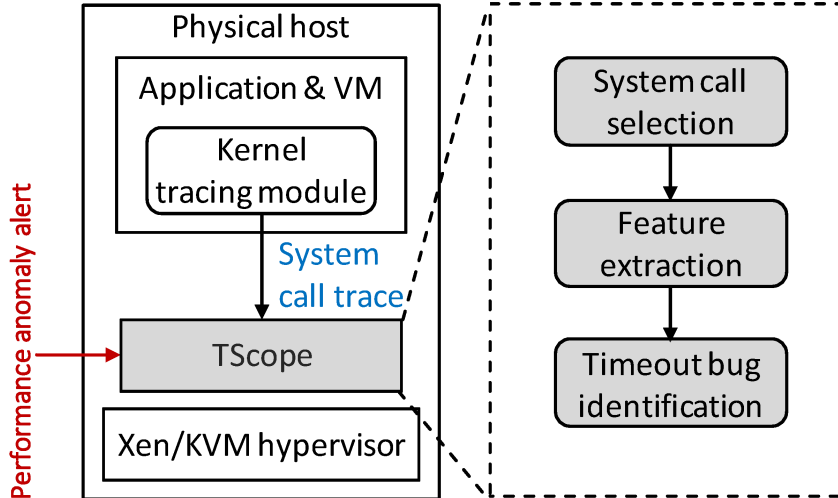


Figure 2.2 The overall architecture of TScope.

rate from 47% to 0.8%. TScope imposes negligible overhead to production server runtime executions and can produce timeout bug identification results in minutes.

The rest of the paper is organized as follows. Section 2.2 describes design details of TScope. Section 2.3 presents the experimental evaluation. Finally, the paper concludes in Section 2.4.

2.2 System Design

In this section, we present the design details of the TScope system. We first provide an overview about TScope. We then describe the system call selection scheme followed by the timeout bug identification details.

2.2.1 Approach Overview

TScope consists four major components as shown by Figure 2.2. When a server system experiences software hang or performance slowdown, TScope first retrieves a window of system call trace from the kernel tracing module LTTng [DD06]. We chose LTTng because it incurs negligible overhead to the server system. In contrast, other system call tracing tools such as KProbes [Kpr], DProbes [Moo01] and SystemTap [Sysb] often impose high overhead to the system. The output of LTTng contains each system call’s timestamp, parameters, and other detailed system call information. Normally, an application can generate tens of millions of system call records per minute, which often provides sufficient data for analyzing performance bugs. Next, TScope performs system call selection to extract those system calls which are related to timeout issues. Third, TScope performs feature extraction over the raw system call trace to extract a list of feature vectors consisting of total execution time values of those selected system calls during each sampling interval. Next, TScope performs anomaly detection over the feature vectors to identify anomalous system calls and check whether

those anomalous system calls are timeout related in order to infer whether the identified bug is a timeout bug.

2.2.2 System Call Selection

Linux provides over 300 system calls for users to use various privileged kernel functions. As mentioned in the Introduction, TScope performs timeout related system call selections using three criteria: 1) system calls that include timeout related parameters, 2) system calls that are related to network communications or synchronizations, and 3) system calls that are used by timeout configuration functions. We now discuss those selection strategies in detail.

System calls with timeout related parameters. We manually examine all the Linux system calls [Sysa] and discover all of those system calls that contain timeout related parameters. For example, `sys_select` contains a timeout argument to determine how long a program should wait for files to become ready for I/O operations. During our previous timeout bug study [Dai18b], we observe that those system calls are often directly invoked when timeout bugs are triggered.

System calls related to network and synchronization. Similar to the first selection step, we manually extract all the system calls which are used by network communication or synchronization. For example, `sys_connect` is used to connect a socket to a specified address and `sys_fsync` is called for synchronizing a file's state with storage devices. The rationale for this selection criteria is that timeout is widely used for handling failures during inter-component communications or synchronizations. So we observe that many timeout bugs are triggered during network or synchronization operations [Dai18b]. As a result, those system calls related to network and synchronizations are likely to provide important hints for us to identify timeout bugs.

```
[14:24:43.520759222] syscall_entry_read: {cpu_id=...}, {..., pid=5004, ..., tid
=5038}, {fd=3, ...}
[14:24:43.520759222] syscall_exit_read: {cpu_id=...}, {..., pid=5004, ..., tid
=5038}, {ret = 30, ...}
[14:24:43.520760005] syscall_entry_write: {cpu_id=...}, {..., pid=5004, ..., tid
=5038}, {fd=5, ...}
[14:24:43.520760218] syscall_exit_write: {cpu_id=...}, {..., pid=5004, ..., tid
=5038}, {ret=1, ...}
[14:24:43.520943737] syscall_entry_poll {cpu_id=...}, {..., pid=5004, ..., tid
=5038}, {..., timeout_msecs=60000}
[14:24:43.520943940] syscall_exit_poll: {cpu_id=...}, {..., pid=5004, ..., tid
=5038}, {ret = -516, ...}
```

Figure 2.3 System call trace example collected by LTTng.

System calls used by timeout configuration functions. In this step, TScope tries to identify those system calls that are used by timeout configuration functions. To do so, TScope checks those functions which provide timeout configurations in standard C or Java libraries. For example, in Java 8 standard library, we check all the packages and identify those functions that provide timeout

Table 2.1 An example of how to calculate time vectors from processed system call lists.

System call	Entry timestamp	Exit timestamp
sys_socket	1523750400	1523750500
sys_read	1523750476	1523750542
sys_socket	1523750502	1523752040
sys_exit	1523751056	1523752037

configurations such as `wait()` in `java.lang.Object` package that makes the current thread to delay execution for a certain time period. It contains a timeout parameter for users to define the maximum interval the thread should wait. Other examples include `sleep()` and `join()` in `java.lang.Thread` package, which define the sleeping time and maximum alive time of a thread, respectively. To collect system calls provided by those timeout configuration functions, we write simple test cases to run those functions and use LTTng to collect the system calls produced by those functions.

Using the above three selection criteria, TScope selects 125 system calls for timeout bug identification. Although we cannot guarantee that our scheme can include all the system calls that are related to timeout, we believe that the system calls we select are all highly correlated to real-world timeout bugs.

2.2.3 System Call Feature Extraction

Each system call entry in the system call trace collected by LTTng consists a pair of records, i.e., `syscall_entry` and `syscall_exit` shown by Figure 2.3. Each record contains the timestamp when the system call occurs, the detailed system information, e.g., CPU ID, process ID and thread ID, and the system call's parameters. Some system calls may contain timeout arguments, e.g., `timeout_msecs` in `syscall_entry_poll`. We find that application-level timeout values are usually passed into these arguments. After we obtain system call trace using LTTng, we first extract the system calls occurred in the specific system using `procname`. For example, if we collect system calls for MySQL server system, we extract the system calls with `procname="mysqld"`. We then perform system call filtering based on the timeout related system call set derived by the previous step.

Next, we extract system call name, system call entry timestamp, thread ID and system call exit timestamp for each selected system call. System call name and thread ID are easy to extract directly. To determine the exact exit timestamp for each system call, we group system calls into different lists according to the thread ID. After we sort the system call list in each thread, the entry timestamp is determined by `syscall_entry` and the nearest corresponding `syscall_exit` of the same system call determines the exit timestamp. After exit timestamps are extracted, we sort all the system calls based on their entry timestamps.

We then segment the selected system call list into different samples based on a certain sampling interval (e.g., 1 second). To create feature vector for each sample, we use execution time of system calls. The rationale of choosing execution time over other features such as frequency is that we

aim at identifying timeout bugs and timeout bugs often cause anomalies in system call execution time. Specifically, we extract a time vector $V = [x_1, \dots, x_n]$ for each sample where x_i denotes the total execution time of all the occurrences of a system call s_i which appeared in the sample. For example, in Table 2.1, `sys_socket` appears twice with the execution time values of 100 and 1538 milliseconds. So in the time vector of this sample, the value of `sys_socket` is set to $100 + 1538 = 1638$ milliseconds. The `sys_read` and `sys_exit` have the execution time of 66 and 981 milliseconds, respectively. So the final time vector for this sample is $[1638, 66, 981]$ corresponding to three selected system calls of $[\text{sys_socket}, \text{sys_read}, \text{sys_exit}]$. Since we consider 125 different system calls, the time vector will have 125 dimensions. If a system call type does not appear in one sample, its value is set to be 0.

2.2.4 Timeout Bug Identification

We now describe how TScope identifies a performance bug as a timeout bug. To do so, TScope first performs anomaly detection over extracted feature vectors to detect any system call execution time anomalies preceding the system hang or performance slowdown. If any anomalies are detected, TScope checks whether those pinpointed abnormal system calls are directly related to timeout, that is, whether the system call includes timeout related parameters.

TScope leverages an unsupervised behavior learning (UBL) [Dea12] to achieve efficient anomaly detection over high dimensional datasets (i.e., 125 dimensions). By choosing unsupervised methods, TScope can perform online anomaly detection without labeled training data. UBL is built on top of Self-Organizing Map (SOM) learning methods, which is one type of artificial neural network. The original implementation of UBL processes system-level metrics such as CPU, memory to detect system-level anomalies. TScope adapts the model to process system call execution time vectors. Compared to other anomaly detection algorithms such as clustering-based methods, SOM can map a high dimensional space into a low dimensional space while preserving the topological properties of original data space, which makes it work well for high dimensional data. Moreover, SOM can achieve higher accuracy than other approaches [Dea15a] by performing multi-variate anomaly detection over high dimensional data.

We now describe how TScope performs anomaly detection using SOM. The SOM model consists of a set of neurons each of which is associated with a weight vector. The weight vector has the same dimension as the time vector, which is 125 dimensions. During the model training phase, the SOM model uses a competitive learning method to update all the neurons. When a new training sample arrives, SOM calculates the Euclidean distance from the training vector to all the neurons and select the best matching neuron that has the smallest distance to the training vector. The weight vectors of the best matching neuron and its neighbors are updated using the training vector. During the anomaly detection phase, SOM finds the best matching neuron for the input time vector in the same way as the training phase. To decide whether an input vector is abnormal or not, we define a neighborhood area size (NAS) for each neuron as the sum of the Euclidean distance from the neuron to a set of its neighbors. We derived a threshold value for the NAS metric based on a user defined percentile value. If the NAS value of the best matching neuron for the input vector exceeds

the threshold, we say the input vector is abnormal. The intuition behind the approach is that normal neurons are trained together many times, which all have small NAS values. In contrast, abnormal neurons are rarely trained, which have large NAS values.

To perform online anomaly detection, TScope splits the system call feature vector list into two halves with equal sizes. The first half is used as training data to create the SOM model while the whole set is used for anomaly detection. Note that SOM is resilient to a small number of noises in the training data. So our training is still valid even if the training data consist of abnormal system calls as long as the abnormal system calls are rare compared to normal system calls.

In addition to detecting anomalies, SOM also identifies which system calls attribute to the detected feature vector anomaly. TScope then checks whether those identified system calls include timeout related parameters to classify the detected bug as a timeout bug or non-timeout bug. For example, the timeout bug MapReduce-5066 is caused by missing timeout setting. When the bug is triggered, we observe that the execution time of `sys_epoll_wait` significantly increases. We observe that `-1` passes into the timeout parameter of the `sys_epoll_wait`, causing the `sys_epoll_wait` block indefinitely. As an example of non-timeout performance bug, Cassandra-5064 bug is caused by an incorrect return value. When the bug occurs, the system falls into an infinite loop. We observe that the `sys_sched_yield` is called continually because the system is doing context switches endlessly consuming 100% CPU resources. However, `sys_sched_yield` does not include any timeout related parameters.

Notice that TScope uses different selection criteria for anomaly detection and timeout bug classification. We choose to use an expanded set of system calls (i.e., system calls including timeout related parameters, system calls related to network/synchronization, system calls invoked during timeout configuration) during anomaly detection, since just considering system calls with timeout related parameters sometimes makes the training dataset too small to yield accurate anomaly detection results. Our experimental evaluation results show the problem of limited system call selections.

2.3 Experiment Evaluation

This section presents our experiment evaluation. We implement a prototype of TScope and conduct our experiment on a host which is equipped with a quad-core Xeon 2.53Ghz CPU along with 16GB memory and runs 64-bit Ubuntu 16.04. The system call trace is collected using LTTng 2.0.1. We introduce the evaluation methodology and the identification results. At the end of this section, we give two detailed examples of how TScope helps diagnose the timeout bugs.

2.3.1 Evaluation Methodology

In this subsection, we describe our 19 real world bug samples and system call trace collection.

Table 2.2 System description.

System	Setup Mode	Description
Hadoop	Distributed	The utilities and libraries for Hadoop modules
HDFS	Distributed	Hadoop distributed file system
MapReduce	Distributed	Hadoop big data processing framework
Cassandra	Distributed	Distributed database management system
Phoenix	Distributed	Parallel and relational database engine
MySQL	Distributed	Scalable database
Zookeeper	Standalone	Synchronization service
Flume	Standalone	Log data collection/aggregation /movement service
Tomcat	Standalone	Java servlet container
Apache	Standalone	HTTP server system

2.3.1.1 Real World Bug Samples

Table 2.3 Timeout bugs' description.

Bug ID	System version	Root cause	Impact	Workload
Hadoop-11252	v2.5.0	Timeout is missing for the RPC connection	Hang	Word count for 765MB file
Hadoop-11252	v2.6.4	Timeout is misconfigured for the RPC connection	Hang	Word count for 765MB file
HDFS-10223	v2.6.4	Timeout setting is ignored and timeout value is hardcoded to a large one	Slowdown	Word count for 765MB file
Phoenix-2496	v4.6.0	Timeout is missing for synchronization	Slowdown	Database queries
MapReduce-5066	v2.0.3-alpha	Timeout is missing when JobTracker calls a URL	Hang	Word count for 765MB file
Cassandra-7886	v2.1.9	Wrong timeout handling	Hang	Database queries
Flume-1842	v1.3.0	Timeout is not calculated correctly	Slowdown	Writing log events to a file repeatedly
Zookeeper-1366	v3.5.0	Clock drifting	Crash	Checking expired events
Tomcat-56684	v6.0.39	Timeout value is set too high when accepting socket connection	Hang	Website browsing
Flume-1819	v1.3.0	Timeout is missing for reading data	Slowdown	Writing log events to a file repeatedly
Flume-1316	v1.1.0	Timeout is misconfigured	Slowdown	Writing log events to a file repeatedly
MapReduce-5724	v2.3.0	Timeout is missing	Hang	Word count for 765MB file

Table 2.4 Non-timeout performance bugs' description.

Bug ID	System version	Root cause	Impact	Workload
Cassandra-5064	v1.2.0-beta	Incorrect return value handling causes an infinite loop	Hang	Database queries
Apache-37680	v2.0.55	Incorrect flag causes infinite loop	Hang	Website browsing
Tomcat-48827	v6.0.24	Error in validating empty tag	Job failure	Website browsing
Tomcat-53450	v7.0.28	Upgrade a read lock to a write lock wrongly	Hang	Website browsing
MapReduce-3738	v0.23.1	Wait for an atomic variable to be set endlessly	Hang	Word count for 765MB file
MySQL-65615	v5.6.5-m8	Truncating tables causes disk flushing	Slowdown	Sysbench (benchmark)
MYSQL-54332	v5.5.5-m3	Two threads are deadlocked due to a locked table	Hang	Sysbench (benchmark)

We collect all the bugs from ten open source systems. All the systems' names, description and setup modes are listed in Table 2.2. These systems vary from back-end to front-end applications, constituting typical representatives and providing a wide coverage of server systems. We set up six systems in distributed modes, to investigate timeout issues occurring on the communication among different nodes in distributed systems.

We firstly describe timeout bug collection. We reproduce 12 timeout bugs, which are collected from bug repositories, e.g., Apache JIRA [Jir] and Bugzilla [Bug]. Each report contains detailed information, e.g., version number, target platform and system's log information. Our reproduced bugs have a wide coverage of root causes and system impacts in our previous timeout bug study [Dai18b]. We cover four categories of root causes, i.e., misused timeout value, missing timeout, improper timeout handling and clock drifting, which occupy top 95% root causes. We cover three categories of impacts, i.e., system unavailability, job failure and performance degradation, which occupy top 98% of impacts brought by timeout bugs. We list the bugs' description in Table 2.3.

To evaluate our classification result on non-timeout bugs, we reproduce 7 non-timeout performance bugs. We list them in Table 2.4. These bugs can also cause system calls' anomaly, making it difficult to distinguish those anomalies raised from timeout bugs.

For each bug, we start LTTng to collect system call trace just when the application is started. After the application runs for two to three minutes normally, we trigger the bug and record the bug triggering time. Then the system continues to run for about two to three minutes and we end up collecting the system calls. We separate the dataset into two halves. We use the first half of the dataset, representing the data generated in the normal state of the system, to train the anomaly detection model. We try our best to run some workloads during normal run, with the goal to reduce the false positives caused by high workloads. The workloads are also listed in Table 2.3.

2.3.1.2 Alternative Approaches

To evaluate TScope's performance, we compare TScope to two SOM based approaches, an existing bug detection and diagnosis tool, i.e., PerfScope, and another clustering approach, i.e., the DBScan clustering.

SOM-all and SOM-parameter: SOM-all and SOM-parameter refer to the approaches of using SOM model to identify timeout bugs under different selection sets. For SOM-all approach, we consider all the system calls. For SOM-parameter, we select the system calls containing timeout related parameters only. Since SOM-all and SOM-parameter select different sets of system calls, the false positive rates are different.

PerfScope: PerfScope is a performance bug detection and diagnosis tool. To be mentioned, we extract the execution units as the samples. An execution units refer to the system call clusters generated by the same function. We divide the system calls based on thread ID. Besides that, we divide the system call list according to time gaps between two consecutive system calls. We define the time gap threshold as the mean + 2×standard deviation. If the interval between two consecutive system calls is larger than this value, we segment the trace between the two system calls. We use the

Table 2.5 Detection result of TScope and four alternative approaches for timeout bugs.

Bug ID	PerfScope	Clustering	SOM-all	SOM-parameter	TScope
Hadoop-11252 (v2.5.0)	✓	✓	✓	✗	✓
Hadoop-11252 (v2.6.4)	✓	✓	✓	✗	✓
HDFS-10223	✓	✗	✓	✓	✓
Phoenix-2496	✗	✓	✓	✓	✓
MapReduce-5066	✓	✗	✓	✗	✓
Cassandra-7886	✓	✓	✓	✗	✓
Flume-1842	✓	✓	✓	✓	✓
Zookeeper-1366	✓	✓	✓	✓	✓
Tomcat-56684	✓	✓	✓	✓	✓
Flume-1819	✓	✓	✓	✗	✓
Flume-1316	✓	✓	✓	✓	✓
MapReduce-5724	✓	✓	✓	✓	✓

appearance vector as the features to cluster similar samples [KR09]. The appearance vector has a similar form as the time vector. The difference is that it only contains boolean variables and they represent whether a system call occurs in a sample. To identify the abnormal samples within clusters, we use the frequency vector and the time vector as two metrics to perform the nearest neighbor algorithm. The time vector is the same as we use. The frequency vector represents how many times a system call appears in a sample. We calculate the Euclidean distance of each sample's frequency (time) vector to its nearest neighbor's within each cluster. We set the threshold as the mean + 2× standard deviation. If one sample's distance to the nearest neighbor is larger than the threshold considering either the frequency vector or time vector, the sample is identified as anomaly.

Clustering: We implement DBScan clustering to identify timeout bugs. The implementation of sampling and feature vector extraction is same as TScope. The advantage of DBScan is that it does not require the number of clusters as input. DBScan algorithm's learning result is sensitive to the parameter ϵ and minimal points. ϵ defines the radius of a cluster. It is the threshold of Euclidean distance from the point to the cluster center. The false positive is reduced with the decreasing of ϵ . To reduce the false positive rate to the minimum, we set the ϵ to 1. The minimum points refers to the minimal number of points to form a cluster. We change the minimum points and find that it influence the identification result little. In our experiment, we set the minimal points to 5.

2.3.2 Results Analysis

We evaluate the accuracy of TScope from detection result, the false positive rate and the classification result of timeout bugs and non-timeout bugs.

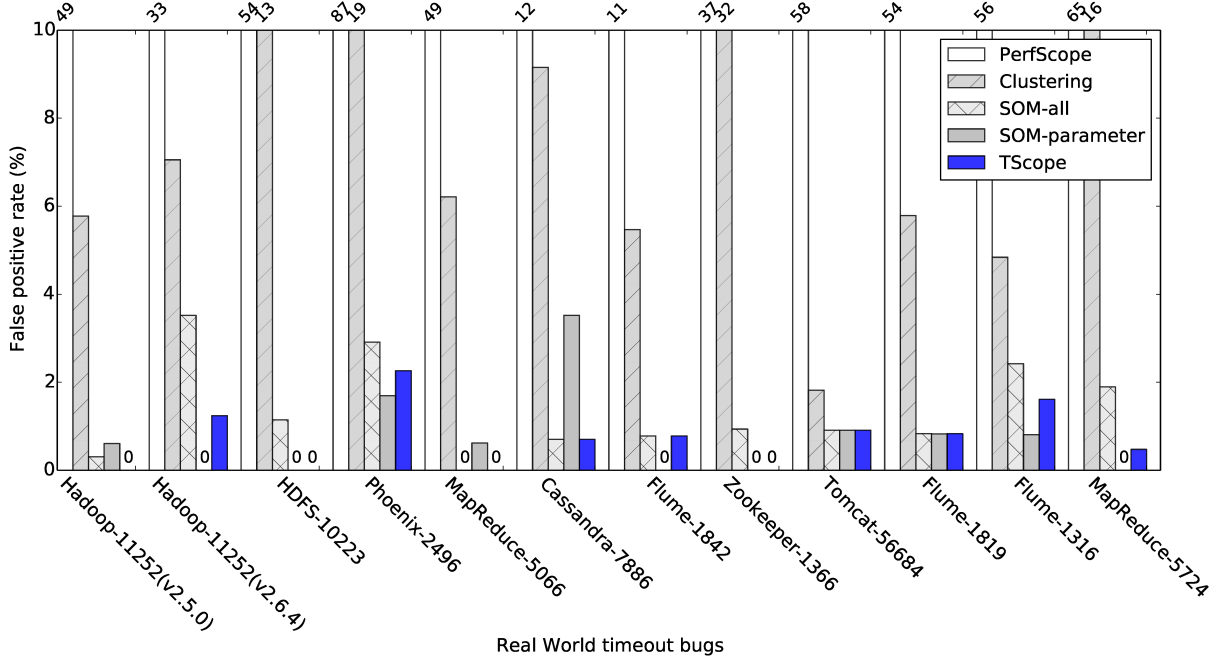


Figure 2.4 False positive rate of TScope and four alternative approaches.

2.3.2.1 Accuracy

We use the whole system call trace as input of SOM model. When SOM model raises a alarm, we check whether the alarms come from the samples after the bug is triggered. If an anomaly is reported after the bug is triggered, then the bug is viewed as truly detected. We use the standard false positive rate A_F to measure the experiment results. The equations are given in Equation 2.1. N_{fp} is the number of false positives, which means that TScope raise a false alarm on the a normal sample. N_{tn} is the number of true negatives, which means that TScope do not raise an alarm and it is a normal sample actually. To make the four approaches comparable, we all use the number of time slots in the same system call trace to calculate the A_F . The division of time slots are the same as sampling of TScope. N_{fp} represents the number of time slots with the false alarms. Similarly, N_{tn} represents the number of normal time slots without alarms. To be mentioned, for PerfScope, the time slot is identified as false positive if one execution unit, occurring on the time slot, reports a false alarm.

$$A_F = \frac{N_{fp}}{N_{fp} + N_{tn}} \quad (2.1)$$

The detection results are shown in Table 2.5 and the false positive rates are listed in Fig. 2.4. We observe that TScope can achieve 100% detection, while PerfScope and clustering can detect 11 and 10 out of 12 timeout bugs correspondingly. However, considering the false positive rate, TScope outperforms PerfScope and clustering a lot. For TScope, the average false positive rate is around 1%, while PerfScope and clustering have average false positives of 50% and 15%. PerfScope is also based

on clustering methods. The clustering methods do not work well to find patterns for time vectors. It can be explained by the curse of dimensionality problem in the machine learning field. In our case, the time vectors have 125 dimensions, representing 125 system calls in our selection set. For a per-second trace sample, we find usually only 10% of the 125 system call occur. For the remaining system calls, the corresponding time vector values are all set to 0. The time vectors constitute a high dimensional sparse matrix. When we use clustering methods, it is difficult to organize the data and analyze the correlation between them. Therefore, the false positive rate is high. Another reason of over 50% false positive for PerfScope is that, there exist some false positive execution units which last for the whole system running time. The consequence is that PerfScope reports false alarms on many time intervals, when the system is running normally.

As shown from the detection results, compared with SOM-all, using TScope reduces the false positive rate by over 30%. The false positive rate is generally decreasing with the decreasing of the number of system calls in the system call selection set. It is intuitive that the anomalies should be reduced when we consider less system calls. The detection result of SOM-parameter approach proves that our selection set is most appropriate. For SOM-parameter approach, although the false positive rate is generally lower, the detection result is very bad, with only 7 out of 12 bugs detected. The reason is that the set of system calls with timeout related parameters is too small. Even though timeout value is passed into timeout related parameter, not all the anomalies caused by timeout bugs can manifest in these system calls. These system calls can call other system calls, further causing anomalies in those system calls. For example, in Flume-1819 bug, the execution time of `sys_wait4`, a system call without timeout parameter, is abnormally prolonged, while the system calls with timeout parameters do not report an anomaly. For Cassandra-7886, MapReduce-5066 and Hadoop-11252(v2.5.0) bugs, the false positive rates of SOM-parameter are even higher than TScope's. The reason is that there are only less than five system calls after selection for SOM-parameter approach, which makes training data too small to train a good model.

Table 2.6 Classification result for the 19 bugs. No. 1 to 12 are timeout bugs, while No. 13 to 19 are non-timeout performance bugs. ✓ means the bug is identified as a timeout (non-timeout) bug and it is indeed a timeout (non-timeout) bug. ✗ means the bug is identified as a timeout (non-timeout) bug but it is a non-timeout (timeout) bug.

ID	Bug ID	Error Message	PerfScope	Clustering	SOM-all	SOM-parameter	TScope
1	Flume-1316 (Timeout)	Missing	✓	✗	✓	✓	✓
2	Flume-1819 (Timeout)	Missing	✓	✗	✓	✗	✓
3	MapReduce-5066 (Timeout)	Missing	✓	✗	✓	✗	✓
4	Hadoop-11252(v2.6.4) (Timeout)	Missing	✓	✗	✗	✗	✓
5	HDFS-10223 (Timeout)	Missing	✓	✗	✓	✓	✓
6	Tomcat-56684 (Timeout)	Missing	✓	✓	✓	✓	✓
7	Zookeeper-1366 (Timeout)	Missing	✓	✓	✓	✓	✓
8	Phoenix-2496 (Timeout)	Missing	✗	✗	✗	✓	✗
9	Hadoop-11252(v2.5.0) (Timeout)	Missing	✓	✓	✓	✗	✓
10	Cassandra-7886 (Timeout)	Misleading	✓	✓	✓	✗	✓
11	Flume-1842 (Timeout)	Missing	✓	✓	✓	✓	✓
12	MapReduce-5724 (Timeout)	Misleading	✓	✓	✓	✓	✓
13	Cassandra-5064 (Non-timeout)	Missing	✗	✗	✓	✗	✓
14	Apache-37680 (Non-timeout)	Missing	✗	✗	✓	✗	✓
15	Tomcat-48827 (Non-timeout)	Correct	✗	✗	✗	✗	✓
16	Tomcat-53450 (Non-timeout)	Correct	✗	✓	✓	✓	✓
17	MapReduce-3738 (Non-timeout)	Missing	✗	✗	✗	✗	✓
18	MySQL-65615 (Non-timeout)	Missing	✗	✗	✗	✓	✓
19	MySQL-54332 (Non-timeout)	Missing	✗	✗	✗	✓	✓

We conduct the classification experiments on 12 timeout bugs and 7 non-timeout performance bugs. The 7 non-timeout performance bugs cause either system hang or performance degradation. All the classification results are shown in Table 2.6. The results show that TScope can correctly classify 18 out of the 19 bugs. The one miss classification is Phoenix-2496 bug. This bug causes 10 seconds delay on the system, which is too short, compared with other timeout bugs causing system slowdown. TScope captures the short delay but it does not consider it as an anomaly caused by timeout bugs. TScope outperforms other four approaches in bug classification. For PerfScope and clustering approaches, the low classification accuracy is caused by the curse of dimensionality problem in high dimensional sparse datasets, which we have already discussed. SOM-all approach can identify 10 out of 12 timeout bugs and 3 out of 7 non-timeout bugs. SOM-all approach's low precision on non-timeout bugs is caused by no selection on system calls. SOM-parameter approach can only classify 10 out of the 19 bugs. It shows that the selection set of SOM-parameter cannot cover all the system calls related to timeout bugs. We can see that most of the bugs produce no error messages, even misleading messages, about the root causes. In this case, TScope provides useful guidance for developers to diagnose the bug. We introduce the Hadoop-11252(v2.6.4) case in detail in the next subsection. TScope can localize the buggy timeout variable for this bug.

Apart from checking the abnormal system calls reported by SOM model, there are another two reasons that TScope can classify timeout bugs. The first reason is that TScope uses time vectors to feed into the anomaly detection model. Our observation is that timeout bugs usually cause anomaly in system call's execution time, while non-timeout performance bugs can not. The manifestation of non-timeout bug is usually that the frequency of a particular system call's occurrence is changed or some rarely seen system calls occur when the bug is triggered. In our study, we also extract frequency vector, which represents how many times each system call appears in a sample. We find that the classification result is worse than that of using time vector. For example, Apache-37680 bug is mistakenly classified as timeout bugs using the frequency vector. The second reason is that TScope uses a unique system call selection strategy. Those anomalies that are not caused by timeout related system calls are filtered. For example, MySQL-54332 bug is mistakenly classified as timeout bugs before selection, while it is correctly classified as non-timeout bugs after selection.

2.3.2.2 Overhead

The LTTng tracing overhead is less than 1%, which is negligible. TScope does not require application profiling, which can impose significant overhead on systems. We list the computation time of log analysis on timeout bugs in Table 2.7. We can see that the average log size of the 12 timeout bugs is near 1000MB. Since the log size is large, it is significant to reduce the overhead to apply TScope in the real world cloud systems. In Table 2.7, feature extraction refers to the combination of sampling, system call selection, extracting the time vectors and training the anomaly detection model. The identification refers to using the built model to detect anomaly and further identify timeout bugs. The majority of the total computation time is spent on feature extraction. The reason is that traversing millions of system calls and categorizing them according to the system information

Table 2.7 Computation time of TScope on timeout bugs.

Bug ID	Log size (MB)	Feature Extraction (seconds)	Identification (seconds)
Hadoop-11252 (2.5.0)	1480	156.17	1.75
Hadoop-11252 (2.6.4)	1602	172.58	1.54
HDFS-10223	461	83.80	1.34
Phoenix-2496	710	37.79	2.04
MapReduce-5066	928	144.37	1.65
Cassandra-7886	355	70.75	1.64
Flume-1842	410	5.26	1.63
Zookeeper-1366	1206	112.58	1.52
Tomcat-56684	96	2.25	1.73
Flume-1819	463	8.05	1.65
Flume-1316	2687	26.24	2.62
MapReduce-5724	1304	96.31	2.21

is time consuming. We observe that the average computation time is tens to hundreds of seconds, which is fast enough to apply TScope in real-world cloud systems.

2.3.3 Case Study

In this subsection, we discuss two examples in detail to show how TScope's identification results help to diagnose the timeout bugs.

2.3.3.1 Hadoop-11252(v2.6.4)

The root cause of this bug is misconfiguring RPC timeout value on connection among Hadoop cluster nodes. The misconfigured timeout value is Integer's maximum value, which is too long for the timeout value. When the bug occurs, the system hangs, producing no error message. TScope can detect the bug and identify it as a timeout bug. Besides, TScope reports five abnormal system calls, including the `sys_epoll_wait`. To find out the misconfigured value, we collect all the timeout related parameters of the five abnormal system calls. In this case, we can easily find that the `sys_epoll_wait` has an abnormal parameter, i.e., `Integer.MAX_VALUE`. Through matching all the timeout values of the corresponding variables in the Hadoop's `.xml` configuration files, we find that the misconfigured timeout value, i.e., `Integer.MAX_VALUE`, comes from the timeout variable `ipc.client.rpc-timeout.ms`, that exactly is the misconfigured timeout variable causing the bug.

2.3.3.2 Cassandra-7886

The root cause of this bug is missing timeout on the connection between the data node and the coordinator. The data node simply drops the data, when the input data is overwhelming. It does not inform the coordinator of the dropping data operation, causing the coordinator hanging on waiting for the acknowledgment response of successfully reading the data. The fixed version adds a timeout on the waiting response operation. When the bug occurs, the system reports the overwhelming exception, which is not relevant to the timeout mechanism. However, TScope detects the bug and identifies the bug as a timeout bug.

2.4 Summary

In this chapter, we have presented TScope, an automatic timeout bug identification tool for production server systems. TScope combines timeout related feature selection and runtime anomaly detection to achieve higher bug identification precision than previous generic performance bug detection tools. TScope does not require any application instrumentation for bug detection, which makes it practical for production server systems. We have implemented a prototype of TScope and conducted extensive experiments using 19 real world performance bugs. The experimental results show that TScope achieves much higher timeout bug identification accuracy than existing alternative schemes. TScope is light-weight and efficient, which imposes less than 1% runtime overhead to the production server and produces timeout bug identification results within minutes.

CHAPTER

3

AUTOMATIC PERFORMANCE BUG FIXING FOR CLOUD SYSTEMS

In this chapter, we present the bug fixing framework for two kinds of bugs, i.e., timeout bugs caused by misconfigured variables and hang bugs caused by infinite loops and blocking operations. We present two practical hybrid solutions, i.e., TFix and HangFix. We introduce the two tools followed by the detailed design and experimental evaluation.

3.1 Introduction

Many production server systems (e.g., Cassandra [Cas], HBase [Hba], Hadoop [Hada]) are migrated into cloud environments for lower upfront costs. However, when a production software bug is triggered in cloud environments, it is often difficult to diagnose and fix due to the lack of debugging information. Particularly, timeout bugs caused by misconfigured variables [Gun14; Hua15a; Dai18b] and software hang bugs causing unresponsive or frozen systems instead of system crashing are extremely challenging to fix, which often cause prolonged service outages. For example, in 2015, Amazon DynamoDB experienced a five-hour service outage [Awsb; Dyn] affecting many AWS customers including Netflix, Airbnb, and IMDb. The root cause of the service outage was a software hang bug where an improper error handling kept sending new requests to the overloaded metadata server, causing further cascading failures and retries. In this chapter, we focus on fixing timeout bugs which are caused by misconfigured timeout variables and software hang bugs which are caused by infinite loops and blocking operations in Java programs.

```

//FSHDFSUtils.java                                     HBase-8389(v0.94.3)
    long recoveryTimeout = conf.getInt("hbase.lease.recovery.timeout", 900000) +
        startWaiting;
48 public void recoverFileLease(..., final Path p,
49     ...) throws IOException {
    ...
62 boolean recovered = false;
63 int nbAttempt = 0;
64 while (!recovered) {
+   for (int nbAttempt = 0; !recovered; nbAttempt++){
65     nbAttempt++;
    ...
71 recovered = dfs.recoverLease(p); //send to HDFS
+   if (recovered) break;
+   if (checkIfTimedout(conf, recoveryTimeout,
+       nbAttempt, p, startWaiting))
+       break;
    ...
104 }}

+boolean checkIfTimedout(final Configuration conf,
+ final long recoveryTimeout, final int nbAttempt,
+ final Path p, final long startWaiting) {
+   if (recoveryTimeout <
+       EnvironmentEdgeManager.currentTimeMillis()) {
+       LOG.warn(...);
+       return true;
+   }
+   return false;
+}

```

Figure 3.1 When the blocks are corrupted, the `recoverLease()` function keeps polling the recovery results, getting “false” and sending a new recovery request, hanging in an infinite loop. “+” means added code, representing the manual patch for this bug.

3.1.1 A Motivating Example

We use the HBase-8389 bug as an example to describe how the performance bug is triggered and how it causes cloud service outage. Figure 3.1 shows the buggy code snippet and the patch. HBase inquires the previous recovery results of the path `p` from HDFS at line #71. Due to an unexpected data corruption, the return results from all the inquiries are always false, which causes HBase to get stuck in this infinite loop between lines #64-104. This bug is difficult to fix because HBase does not produce any log information and HDFS provides many misleading error messages. It took the developer 24 days to fix the bug after submitting 10 versions of unsuccessful patches. The final patch created by the developer is actually quite simple, that is, adding a timeout check mechanism to break out of the loop when the recovery operation persistently fails after a certain number of retries.

3.1.2 Contribution

In this chapter, we present TFix and HangFix, an automatic timeout bug fixing system and a software hang bug fixing system.

TFix leverages TScope [He18] to detect a timeout bug in server systems. After a timeout bug is detected, TFix executes a novel *drill-down bug analysis* protocol to automatically narrow down the

root cause of the detected timeout bug and produce recommendations for fixing the timeout bug. TFix first determines whether the detected timeout bug is caused by mis-using a timeout scheme. To achieve this goal, we leverage a system call frequent episode mining scheme [Dea14] to check whether any commonly used timeout functions (e.g., `MonitorCounterGroup` function in Flume system [Flu]) are invoked when the bug is triggered. If the timeout bug is classified as a misused timeout bug, TFix further localizes the functions that are affected by the timeout bug. Intuitively, when a timeout bug is triggered, the affected function will either run longer time or run more frequently. TFix employs an application performance tracing tool called Dapper[Sig10] to identify timeout affected functions. After the function is identified, we use the taint analysis tool [Che] to narrow down which timeout variable(s) are used by the affected function. We then perform timeout variable value recommendation based on the profiled execution time of the pinpointed function during normal runs.

HangFix takes the application byte-code and hang bug triggering test cases as inputs, and produces a hang bug fix patch as the output. To make HangFix practical for production cloud environments, we do not require any application source code or application-specific knowledge. In order to create effective and non-intrusive fixes [Nis15], HangFix consists of four closely integrated steps: 1) *hang function localization* which leverages stack trace analysis to pinpoint exact function which causes the application to get stuck in the hang state; 2) *likely root cause pattern matching* which leverages static code analysis over the identified hang function to automatically match a set of common hang bug root cause patterns; 3) *patch generation* which automatically produces a hang bug fix patch based on the matched likely root cause pattern by inserting existing exceptions or timeout mechanisms to break the application out of the stuck state; and 4) *patch validation* which applies the initial bug detection tool and the hang function detection to the patched code to validate whether the hang bug still exists. We also test the patched code with the whole regression test suites to check whether our fix violates any required regression tests. The patch is deemed to be successful only if the patched code passes both bug detection and regression tests.

Specifically, this chapter makes the following contributions.

- We describe a holistic *drill-down bug analysis* framework which can automatically narrow down the root cause of a timeout bug and provide recommendations for fixing the bug.
- We describe a hybrid scheme that combines dynamic application performance tracing and static taint analysis to localize the misused timeout variable and provide proper timeout value recommendations for fixing the timeout bug. The timeout values suggested by TFix can effectively correct all the tested misused timeout bugs.
- We present a new hang bug fixing framework to automatically fix a hang bug that is triggered in production cloud environments.
- We describe a hang bug root cause pattern matching scheme that can quickly classify hang bugs into different likely root cause types. We develop an automatic hang fix patch generation

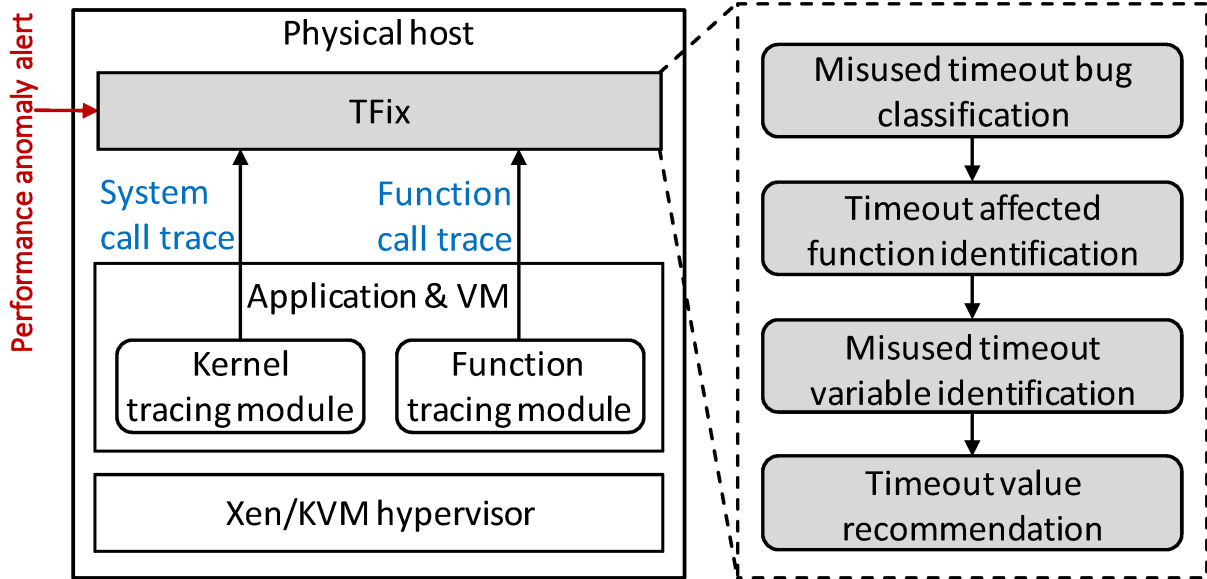


Figure 3.2 The architecture of TFix.

system that can produce proper patched code according to identified likely root cause patterns. The experimental results show that HangFix successfully fixes 40 of 42 reproduced real-world software hang bugs in seconds.

The rest of the chapters is organized as follows. Section 3.2 describes TFix’s design details. Section 3.3 presents TFix’s experimental evaluation. Section 3.4 describes HangFix’s design details. Section 3.5 presents HangFix’s experimental evaluation. Finally, the chapter concludes in Section 3.6.

3.2 Fixing Timeout Bugs Caused by Misconfigured Variables

In this section, we present the design details of the TFix system. We first provide an overview about TFix. We then describe the misused timeout bug classification scheme followed by the timeout affected function identification. Next, we talk about the misused timeout variable identification and timeout value recommendation.

3.2.1 Approach Overview

TFix provides a drill-down bug analysis framework for fixing misused timeout bugs, which consists of four major components as shown by Figure 3.2. When a server system experiences software hang or performance slowdown, TFix leverages TScope [He18] to identify whether the anomaly is caused by a timeout bug by analyzing a window of system call trace collected by the kernel tracing module LTTng [DD06]. If TScope confirms that the performance anomaly is caused by a timeout bug, TFix is triggered to conduct further drill-down analysis. TFix first performs timeout bug classification to determine whether the timeout bug is caused by mis-using certain timeout mechanisms (Section

3.2.2). If the classification result is positive, TFix employs the application function tracing framework Dapper [Sig10] to identify which functions are affected by the misused timeout bug (Section 3.2.3). Next, TFix leverages static taint analysis to localize which timeout variables are used by the identified timeout affected function (Section 3.2.4). Lastly, TFix produces recommendation for the mis-used timeout variable for fixing the timeout bug (Section 3.2.5). The whole drill-down bug diagnosis protocol is executed automatically without requiring any human intervention. We will describe each component in details in the following subsections.

3.2.2 Misused Timeout Bug Classification

TFix leverages TScope [He18] to determine whether a detected system anomaly is caused by a timeout bug. Timeout bugs can be broadly classified into two groups: 1) *misused timeout bug* where the system anomaly is caused by some incorrectly used timeout variables; and 2) *missing timeout bug* where the system anomaly is caused by lack of timeout mechanisms. In this chapter, TFix focuses on fixing misused timeout bug by identifying root cause timeout variables and suggesting proper timeout values for fixing the bug. To achieve this goal, TFix first needs to classify a detected timeout bug as a misused timeout bug. Intuitively, a misused timeout bug is triggered when a certain timeout related function (e.g., `URL.openConnection`, `ServerSocketChannel.open`, `ReentrantLock.tryLock`) is executed. Thus, TFix performs misused timeout bug classification by checking whether timeout related functions are invoked when the bug is triggered.

TFix first provides an offline comparative analysis to extract timeout related functions for each server system. We observe that different server systems often employ different timeout related classes. However, although each system has multiple timeout variables to guard connections, those timeout mechanisms are usually configured by common timeout configuration classes. For example, Flume's timeout mechanisms are built on top of `MonitorCounterGroup` inside the `instrumentation` class, which is used for monitoring system state and building timers. To identify those timeout configuration classes, we employ a dual testing scheme. For each system, we produce a set of test cases each of which consists of two dual parts: one part uses timeout and the other part does not employ timeout. For example, we build a socket connection between the client and HDFS server to write data into HDFS. The difference between the two counterparts is that one has socket write timeout while the other does not. We use HProf [Hpr] to trace the invoked Java functions during the execution of those dual test cases. We compare the lists of the Java functions produced by the two dual test cases in order to extract those functions which only appear in the profiling result of those test cases with timeout mechanisms. To further narrow down the scope of timeout related functions, we only keep those functions that are related to timeout configuration, network connection and synchronization since timeout mechanisms need timers to monitor the elapsed time and timeout mechanisms are often used in network connection and synchronization operations.

After identifying those timeout related functions, TFix needs to employ an efficient scheme to match with those functions during production run. To avoid expensive application function instrumentation, TFix employs a system call frequent episode mining scheme [Dea14] to match

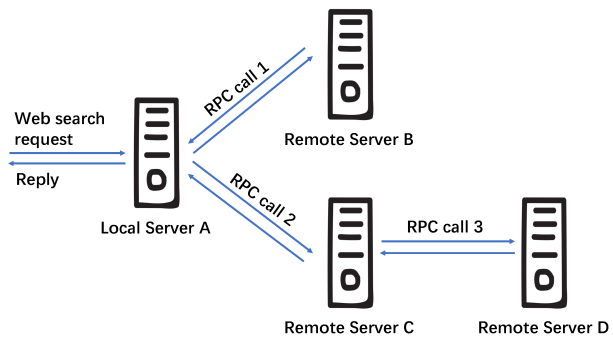


Figure 3.3 A web search example.

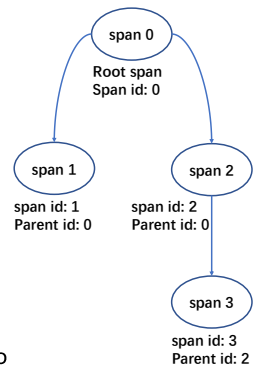


Figure 3.4 The Dapper trace.

with those timeout related functions. The basic idea is to extract unique system call sequences produced by those timeout related functions during the offline analysis. During production run, TFix performs the frequent episode mining over runtime system call sequences and checks whether the frequent system call sequences produced by those timeout related functions exist in the runtime trace. If we find one or more timeout related function matching, TFix classifies the detected bug as a misused timeout bug.

3.2.3 Timeout Affected Function Identification

After classifying a detected bug as a misused timeout bug, TFix wants to identify which functions are affected by this misused timeout bug. To achieve this goal, TFix leverages a commonly used application performance tracing tool, i.e., Google’s Dapper [Sig10] framework. Dapper allows us to trace the beginning and ending timestamps of all function calls and the control flow graph for the diagnosed bug. We choose Dapper tracing tool because it supports distributed systems and incurs low runtime overhead to production systems. The existing implementations of Dapper tracing can only be applied to RPC related functions. TFix augments the Dapper tracing tool to support all timeout related functions.

Dapper’s tracing can be modeled as a tree. The tree nodes are called spans and the edges indicate the control flow between spans. Each span contains a span id and a parent id. The root span does not have a parent id. All spans in the tree share the same trace id. A span represents a RPC connection or a function call, containing the information of both the caller and callee (or the client and server).

For example, in Figure 3.3, a user issues a web search request to the local server A. Server A receives the request and sends the request to the remote server B and C to retrieve the results. Server B stores the data locally and sends the result back to Server A. Server C does not contain the data, thus it sends a request to the remote server D to get the result before responding to Server A. In this case, a simple web search contains four RPC calls between the user and the local server or between servers. When we apply the Dapper framework to trace the control flow of the example in Figure 3.3, we get a RPC tree, shown in Figure 3.4. The root span (i.e., Span 0) represents the RPC request and

```
{
  "i": "1b1bdfddac521ce8", "s": "df4646ae00070999",
  "b": 1543260568612, "e": 1543260568654,
  "d": "org.apache.hadoop.hdfs.protocol.ClientProtocol.getDatinodeReport",
  "r": "RunJar", "p": ["84d19776da97fe78"]
}
```

Figure 3.5 A trace example of Dapper.

response between the user and Server A. Span 1 indicates the RPC connection between Server A and Server B. Span 2 represents the RPC connection between Server A and Server C. Span 3 illustrates the RPC connection between Server C and Server D. The edge between each span indicates a control flow. For example, Server A receives the user’s request and issues RPC call 1 and 2 to Server B and C, respectively. Thus, Span 1 and Span 2 share the same parent span (i.e., Span 0).

After retrieving a Dapper trace for a target bug, we first extract the execution time and frequency of all the functions invoked when the bug happens. Specifically, we calculate the frequency of each function by simply counting how many times it is invoked in the Dapper trace. We calculate the execution time of each function by subtracting the beginning time from the ending time. Figure 3.5 shows a Dapper trace example. We can see the Dapper trace is well structured. The trace contains various labels indicating different kinds of information. Among them, “b” and “e” indicate the beginning timestamp and the ending timestamp of a function, respectively. “d” represents the function name and “r” represents the process name.

We further identify the timeout affected functions by checking the abnormality of the functions’ execution time and frequency. We need to consider two cases: 1) a timeout value is set to be too large or 2) a timeout value is set to be too small. If the timeout value is set to be too large, the execution time of the timeout affected function is much longer than its execution time during the system’s normal run. If the timeout value is set to be too small, the system experiences repeated failures due to frequent timeout. Therefore, the frequency of the timeout affected function is much higher than its frequency during the system’s normal run.

For the first case where the timeout value is too large, we identify a function as a timeout affected function by checking whether its execution time is much larger than the maximum execution time during system’s normal run. For example, in HBase-13647 and HBase-6684, the timeout value for the RPC connection is misconfigured to be `Integer.MAX_VALUE`. The system works fine under normal state where an HBase client exchanges messages with an HBase server (e.g., HMaster, RegionServer) within tens of seconds successfully. However, when the HBase server fails, the HBase client hangs for about 24 days, causing the execution time of the HBase client’s RPC function significantly prolonged. We identify the RPC function as the timeout affected function based on its increased execution time.

For the second case where the timeout value is too small, the system experiences repeated failures due to frequent timeout. Therefore, the frequency of the root cause function greatly increases when the timeout bug is triggered while the execution time of the affected function is similar to the maximum execution time during the system normal run. We thus use frequency to identify

```

//hdfs-site.xml
1327 <property>
1328   <name>dfs.image.transfer.timeout</name>
1329   <value>60000</value>
...
1336 </property>

/* tainted variables */

//DFSConfigKeys class
862 public static final String
863   DFS_IMAGE_TRANSFER_TIMEOUT_KEY
864   = "dfs.image.transfer.timeout";
865 public static final int
866   DFS_IMAGE_TRANSFER_TIMEOUT_DEFAULT = 60 * 1000;

/* timeout affected function */

//TransferFsImage class
258 public static ... doGetUrl(...) throws IOException {
    /* timeout variable */
    ...
271   timeout = conf.getInt(
272     DFSConfigKeys.DFS_IMAGE_TRANSFER_TIMEOUT_KEY,
273     DFSConfigKeys.DFS_IMAGE_TRANSFER_TIMEOUT_DEFAULT);
    ...
277   connection.setReadTimeout(timeout);
    ...
319   InputStream stream = connection.getInputStream();
    ...
358   num = stream.read(buf);
    ...
401 }

```

Figure 3.6 TFix uses the static taint analysis to identify the misused timeout variable for the HDFS-4301 bug.

those timeout related functions. For example, in the HDFS-4301 bug, the system experiences continuous failures. We identify the `doGetUrl`, `getFileClient`, `uploadImageFromStorage`, and `doCheckpoint` functions as timeout affected functions because of their invocation frequencies significantly increase.

3.2.4 Misused Timeout Variable Identification

In this subsection, we describe how TFix identifies the misused timeout variables contributing to the misused timeout bugs. Specifically, we adopt the static taint analysis to correlate the timeout variables with the timeout affected functions to identify the misused timeout variables.

To localize which timeout variable is used when the bug happens, we first retrieve all the timeout variables in the target system. In large scale distributed systems, timeout variables along with other configurable parameters are often stored in specific configuration files [XZ15]. For example, in a Hadoop system, all the configurable variables are defined with default values in configuration files, such as `HConstant` and `DFSConfigKeys` classes. These variables' value can be overridden and customized by users in `.xml` configuration files. Thus, all the variables appear in systems' configuration files and contain "timeout" keyword in their names are potentially related to misused timeout bugs. Next, we taint all these timeout variables and conduct data flow dependency analysis

on them to extract all related variables statically. We then check whether the timeout affected functions use the timeout related variables. If a timeout affected function f uses a timeout related variable v_t , we consider v_t as a misused timeout variable candidate. To achieve high accuracy, we also compare the execution time of f with the value of v_t . If they match, we consider v_t as the misused timeout variable.

For example, Figure 3.6 shows how TFix uses the static taint analysis to identify the misused timeout variable for the HDFS-4301 bug. In this bug, the default timeout value is set to 60 seconds in `DFS_IMAGE_TRANSFER_TIMEOUT_DEFAULT` in `DFSConfigKeys.java`. If users configure the timeout variable `dfs.image.transfer.timeout` in `hdfs-site.xml`, the system uses the configured value. Otherwise, the system uses the default value. We annotate both `dfs.image.transfer.timeout` and `DFS_IMAGE_TRANSFER_TIMEOUT_DEFAULT` as tainted. After applying static taint analysis, we find that the timeout affected function `doGetUrl` uses both tainted variables at line #271-273. Since the user configures the value of `dfs.image.transfer.timeout` in `hdfs-site.xml`, we determine that the misused timeout variable is `dfs.image.transfer.timeout`. We also perform cross validation between the timeout variable value and the execution time of the timeout affected function to confirm whether our timeout variable identification is accurate.

3.2.5 Timeout Value Recommendation

After pinpointing the misused timeout variable, TFix recommends a proper timeout value to fix the timeout bug. The timeout value recommendation considers two different cases: 1) the timeout value is too large or 2) the timeout value is too small. As mentioned in Section 3.2.3, if the timeout affected function experiences significant execution time increase, TFix infers that the timeout bug is caused by a too large timeout value. In those cases, TFix recommends to set the timeout value to be the maximum execution time of the affected function right before the bug is detected. Such in-situ profiling results should reflect the system's current environment such as network bandwidth, I/O read/write speed, and CPU load. If the timeout bug is caused by a too small timeout value, we should observe the frequency of the function execution increases. Under those circumstances, TFix suggests a larger timeout value by continuously multiplying the current timeout value by a ratio α , $\alpha > 1$ until the timeout bug is corrected. α is a user configurable parameter which represents the tradeoff between fast fix and larger timeout delay. In our experiments, we set α to be 2.

3.3 Experimental Evaluation for Fixing Timeout Bugs

In this section, we present experimental evaluation results. We have implemented a prototype of TFix and conducted our experiments on a cluster in our research lab. Each host is equipped with a quad-core Xeon 2.53Ghz CPU along with 16GB memory and runs 64-bit Ubuntu 16.04. The system call trace is collected using LTTng v2.0.1. The function call trace is collected using Google's Dapper framework. We first introduce our evaluation methodology. We then present the results of misused timeout bug classification, timeout affected function identification, misused timeout

Table 3.1 System description.

System	Setup Mode	Description
Hadoop	Distributed	The utilities and libraries for Hadoop modules
HDFS	Distributed	Hadoop distributed file system
MapReduce	Distributed	Hadoop big data processing framework
HBase	Standalone	Non-relational, distributed database
Flume	Standalone	Log data collection/aggregation /movement service

variable identification, timeout value recommendation, and overhead. We also present three case studies to show how TFix correct timeout bugs in details.

3.3.1 Methodology

Table 3.2 Timeout bug benchmarks.

Bug ID	System Version	Root Cause	Bug Type	Impact	Workload
Hadoop-9106	v2.0.3-alpha	“ipc.client.connect.timeout” is misconfigured	Misused too large timeout	Slowdown	Word count
Hadoop-11252	v2.6.4	Timeout is misconfigured for the RPC connection	Misused too large timeout	Hang	Word count
HDFS-4301	v2.0.3-alpha	Timeout value on image transfer operation is small	Misused too small timeout	Job failure	Word count
HDFS-10223	v2.8.0	Timeout value on setting up the SASL connection is too large	Misused too large timeout	Slowdown	Word count
MapReduce-6263	v2.7.0	“hard-kill-timeout-ms” is misconfigured	Misused too small timeout	Job failure	Word count
MapReduce-4089	v2.7.0	“mapreduce.task.timeout” is set too large	Misused too large timeout	Slowdown	Word count
HBase-15645	v1.3.0	“hbase.rpc.timeout” is ignored	Misused too large timeout	Hang	YCSB
HBase-17341	v1.3.0	Timeout is misconfigured for terminating replication endpoint	Misused too large timeout	Hang	YCSB
Hadoop-11252	v2.5.0	Timeout is missing for the RPC connection	Missing	Hang	Word count
HDFS-1490	v2.0.2-alpha	Timeout is missing on image transfer between primary NameNode and Secondary NameNode	Missing	Hang	Word count
MapReduce-5066	v2.0.3-alpha	Timeout is missing when JobTracker calls a URL	Missing	Hang	Word count
Flume-1316	v1.1.0	Connect-timeout and request-timeout are missing in AvroSink	Missing	Hang	Writing log events
Flume-1819	v1.3.0	Timeout is missing for reading data	Missing	Slowdown	Writing log events

We collected all the bugs from five open source systems. All the systems' names, description and setup mode are listed in Table 3.1. We set up three systems in distributed modes to investigate timeout issues occurring on the communication among different nodes in distributed systems.

We reproduce 13 real-world timeout bugs, including 8 misused timeout bugs and 5 missing timeout bugs. These bugs are collected from bug repositories, e.g., Apache JIRA [Jir] and Bugzilla [Bug]. Each report contains detailed information, e.g., version number and system's log information. We list the bugs' description in Table 3.2. In our previous timeout bug identification work [He18], the bug benchmarks covered all different root causes presented in the timeout bug study paper [Dai18b]. In contrast, TFix focuses on misused timeout bugs. Moreover, TFix only supports Java application systems currently.

We run workloads when the system is in the normal state, in order to approach the real-world system running. The workloads are also listed in Table 3.2. Specifically, for the Hadoop, HDFS and MapReduce systems, we run word count job on a 765MB text file. For the HBase system, we use the YCSB workload generator to make insertion, query and update operations on a table. For the Flume system, we write log events to the log collection tool and distribute the logs repeatedly. These workloads invoke the timeout related functions all of our tested timeout bugs.

3.3.2 Results

In this subsection, we first present the classification results for timeout bugs, followed by the identification results for timeout affected function, and then describe the results of localizing the misused timeout variable and timeout value recommendation.

3.3.2.1 Classification Results for Timeout Bugs

Table 3.3 TFix's classification result of timeout bugs.

Bug ID	Bug Type	Matched Timeout Related Functions	Correct Timeout Bug Classification?
Hadoop-9106	misused	System.nanoTime, URL.<init>, DecimalFormatSymbols.getInstance, ManagementFactory.getThreadMXBean	Yes
Hadoop-11252 (v2.6.4)	misused	Calendar.<init>, Calendar.getInstance, ServerSocketChannel.open	Yes
HDFS-4301	misused	AtomicReferenceArray.get, ThreadPoolExecutor	Yes
HDFS-10223	misused	GregorianCalendar.<init>, ByteBuffer.allocateDirect	Yes
MapReduce-6263	misused	DecimalFormatSymbols.initialize, ReentrantLock.unlock, AbstractQueuedSynchronizer, ConcurrentHashMap.PutIfAbsent, ByteBuffer.allocate	Yes
MapReduce-4089	misused	charset.CoderResult, AtomicMarkableReference, DateFormatSymbols.initializeData	Yes
HBase-15645	misused	CopyOnWriteArrayList.iterator, URL.<init>, System.nanoTime, AtomicReferenceArray.set, ReentrantLock.unlock, AbstractQueuedSynchronizer, DecimalFormat.format	Yes
HBase-17341	misused	ScheduledThreadPoolExecutor.<init>, DecimalFormatSymbols.initialize, System.nanoTime, ConcurrentHashMap.computelfAbsent	Yes
Hadoop-11252 (v2.5.0)	missing	None	Yes
HDFS-1490	missing	None	Yes
MapReduce-5066	missing	None	Yes
Flume-1316	missing	None	Yes
Flume-1819	missing	None	Yes

Table 3.4 The timeout affected functions.

Bug ID	Timeout affected functions
Hadoop-9106	Client.setupConnection()
Hadoop-11252 (v2.6.4)	RPC.getProtocolProxy()
HDFS-4301	TransferImage.doGetUrl()
HDFS-10223	DFSUtilClient.peerFromSocketAndKey()
MapReduce-6263	YARNRunner.killJob()
MapReduce-4089	TaskHeartbeatHandler.PingChecker.run()
HBase-15645	RpcRetryingCaller.callWithRetries()
HBase-17341	ReplicationSource.terminate()

TFix classifies a misused timeout bug by checking whether it invokes commonly used timeout functions. As mentioned in Section 3.2.2, our classification scheme matches the runtime system call traces with the frequent system call episodes produced by timeout related functions. Table 3.3 shows the classification results. TFix successfully classifies all the 13 timeout bugs. Table 3.3 also shows the matched timeout related functions, which are often used for network communications (e.g., `ServerSocketChannel.open`, `URL.<init>`), synchronization operations (e.g., `AtomicReferenceArray.get`, `ReentrantLock.unlock`), and timer settings (e.g., `GregorianCalendar.<init>`, `System.nanoTime`). The results match our assumption that timeout mechanisms are used to protect communications and synchronizations.

3.3.2.2 Timeout Affected Function Identification Results

We use the Dapper framework to trace the function calls for tested misused timeout bugs. Dapper has various implementation on different production systems. For example, an implementation of Dapper, HTrace [Htr] is integrated into Hadoop and HBase. However, the existing Dapper implementation targets at RPC libraries only. We augment the Dapper implementation by inserting the instrumentation points on synchronization operations and IPC calls. For example, the `setupConnection` function in Hadoop's `ipc.Client` class sets up a connection with IPC server. This `setupConnection` function cannot be traced by the existing HTrace implementation. We formulate the `setupConnection` as a span, which contains all the IPC connection activities, and add annotations to label the function.

Table 3.4 shows the timeout affected functions identified by TFix in all tested misused timeout bugs. For Hadoop-9106, Hadoop-11252(v2.6.4), HDFS-10223, MapReduce-4089, HBase-15645 and HBase-17341 bug, the timeout affected functions have larger execution time compare with that during normal runs. For HDFS-4301 and MapReduce-6263 bug, the timeout affected functions have higher occurrence frequencies with identical execution time during each function run.

Table 3.5 The fixing result of TFix.

Bug ID	Localize the misused timeout variable	Recommended timeout value	Timeout value in the patch	Is bug fixed after applying TFix recommendation?
Hadoop-9106	ipc.client.connect.timeout	2s	20s	Yes
Hadoop-11252 (v2.6.4)	ipc.client.rpc.timeout.ms	80ms	0ms	Yes
HDFS-4301	dfs.image.transfer.timeout	120s	60s	Yes
HDFS-10223	dfs.client.socket.timeout	10ms	1min	Yes
MapReduce-6263	yarn.app.mapreduce.am.hard-kill-timeout-ms	20s	10s	Yes
MapReduce-4089	mapreduce.task.timeout	100ms	10min	Yes
HBase-15645	hbase.client.operation.timeout	4.05s	20min	Yes
HBase-17341	replication.source.maxretriesmultiplier	27ms	1s	Yes

3.3.2.3 Localizing the Misused Timeout Variable and Timeout Value Recommendation

We adopt existing static taint tracking framework, i.e., Checker [Che], to localize the misused timeout variable. Checker includes various useful plugins running on the Java compiler. The plugins can check null exceptions, invalidate inputs, tainted variables, etc. We apply the tainted checker on javac compiler to localize misused variables. Specifically, we select all the timeout variables in configuration files. For each timeout variable, we annotate it as tainted. We compile the system's source code on the javac compiler. If Checker catches the tainted variable in a timeout affected function, we consider it as the misused timeout variable. Table 3.5 shows the misused timeout variables localized by TFix for 8 misused timeout bugs.

Table 3.5 also shows the recommended timeout value by TFix. After adopting TFix's recommended value in the system, we find the anomaly does not occur on the system anymore under the same workload. We also list the timeout value in the bugs' patch file in Table 3.5. We observe that the timeout values in the patches are not always correct. When patching misused timeout bugs, developers usually make the timeout variable configurable for users and set the default value to be same as the buggy version before patching. However, it is challenging to make the correct configurations, even for experienced engineers. For example, in the patch of Hadoop-11252(v2.6.4), the default value of the `ipc.client.rpc-timeout.ms` variable is configured to be 0 milliseconds. Developers expose the variable for users to configure. If users do not configure the variable, the timeout bug still happens in a fixed version. As shown in Table 3.5, TFix can fix all the misconfigured timeout bugs. However, TFix's fixing strategy may be different from the patch. We use HDFS-4301 bug as an example. In the patch of HDFS-4301, the default value of `dfs.image.transfer.timeout` is still set to 60 seconds, which is identical with the timeout value before patching. However, the patch limits the chunk size for image transfer, that matched the timeout value. In comparison, TFix changes the timeout value to 120 seconds, that can also fix the problem.

We should note that, the recommended timeout value by TFix might be different under different workloads. This is our design choice, because a fixed timeout setting cannot handle unexpected workload changes or environment fluctuations. For example, in HBase-15645 bug, the misused timeout variable `hbase.client.operation.timeout` defines the time to block a certain table to prevent concurrency issues. Since the table size is small for YCSB workload in our evaluation, the recommended value by TFix is only 4.05 seconds. If we use 20 minutes in the patch under the same YCSB workload, the user will still experience a noticeable delay (about 20 minutes) in the system.

3.3.3 Overhead

In this subsection, we discuss the runtime overhead of TFix. TFix's runtime overhead comes from two tracing modules, i.e., system call tracing and function call tracing. Kernel level system call tracing only incurs less than 1% overhead to the system [DD06]. TFix enables function call tracing (the Dapper tracing) only on a small number of functions which are related to timeout configuration, network connection, and synchronization. We run the workloads on each system with and without

Table 3.6 The runtime overhead of TFix.

System	Workload	Average CPU Overhead	Standard Deviation of CPU Overhead
Hadoop	Word count	0.29%	0.023%
HDFS	Word count	0.44%	0.050%
MapReduce	Word count	0.33%	0.012%
HBase	YCSB	0.41%	0.024%

tracing. We use the typical benchmarks for all server systems and impose the same types of workloads that trigger the tested timeout bugs. We measure the tracing overhead and list the results in Table 3.6. We observe that the overhead of TFix in terms of additional CPU load is less than 1%, which makes it practical to apply TFix in real-world production systems.

3.3.4 Case Study

In this subsection, we discuss three real world bugs in detail to show how TFix works.

Hadoop-9106: This bug is caused by setting too large value to `ipc.client.connect.timeout` variable. The IPC client sets up a connection to the IPC server and the connection timeout value is set to 20 seconds. When the bug happens, the IPC server fails to respond to the IPC client and the IPC client relies on the timeout mechanism to close the connection. Therefore, too large timeout value causes a noticeable delay on the system.

TFix first successfully classifies the bug as a misused timeout bug, because TFix finds the matched timeout related functions `System.nanoTime`, `URL.<init>`, `DecimalFormatSymbols.getInstance` and `ManagementFactory.getThreadMXBean`, when the bug is triggered. TFix then identifies a timeout affected function `Client.setupConnection()` because of its prolonged execution time. Next, TFix pinpoints the misused timeout variable as `ipc.client.connect.timeout` via static taint analysis, because it is used by the `setupConnection()` function. Last, TFix recommends the timeout value as 2 seconds, that is the maximum execution time of `Client.setupConnection()` during system's normal run. We set the `ipc.client.connect.timeout` to 2 seconds and re-run the system. We observed the bug does not happen.

MapReduce-6263: This bug is caused by a too small timeout value for killing MapReduce jobs. As shown by Figure 3.7, the YarnRunner sends a killing job request to the ApplicationMaster with the timeout value set to 10 seconds. However, when the workers are processing a large MapReduce job with limited resources, it takes the ApplicationMaster longer than 10 seconds to finish the job and respond to the YarnRunner. Instead of keeping waiting for the response from the ApplicationMaster, the YarnRunner sends a request to the ResourceManager to kill the ApplicationMaster by force. This force kill results in job history data loss and unavailability of the deployed application.

TFix first successfully classifies the bug as a misused timeout bug, because TFix finds the matched timeout related functions `DecimalFormatSymbols.initialize`, `ReentrantLock.unlock`, `AbstractQueuedSynchronizer`, `ConcurrentHashMap.PutIfAbsent` and `ByteBuffer.allo`

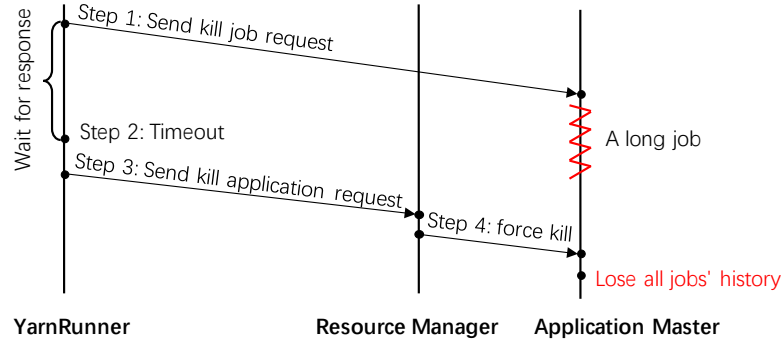


Figure 3.7 The MapReduce-6263 timeout bug. The ApplicationMaster is forcefully killed, losing all job history after the bug is triggered. The root cause of this bug is the too small timeout value for killing job request sent from YarnRunner to the ApplicationMaster.

cate, when the bug is triggered. TFix then identifies a timeout affected function `YARNRunner.killJob()` because of its increased frequency. Next, TFix pinpoints the misused timeout variable as `yarn.app.mapreduce.am.hard-kill-timeout-ms` via static taint analysis. Last, TFix recommends the timeout value as 20 seconds by doubling the current timeout value. We replace 10 seconds with 20 seconds and re-run the system. We observe the bug does not happen and the job finishes successfully.

3.4 Fixing Hang Bugs Caused by Infinite Loops and Blocking Operations

In this section, we describe the HangFix’s design details, which is shown in Figure 3.8. We first describe the hang function localization scheme. Then we describe our pattern-driven approach to quickly identifying likely root causes of hang bugs and automatically generating hang fix patches. HangFix performs root cause pattern matching by analyzing intermediate representation (IR) code produced by Java Soot compiler [Soo]. The patched code is created by inserting proper IR code into the hang function.

3.4.1 Hang Function Localization

After a hang bug is detected in production cloud environments, HangFix leverages stack traces to localize the hang function. Since the hang function often repeatedly appears in the stack trace, we can capture the abnormal behavior of the hang function after the hang bug is triggered. Specifically, HangFix uses the `jstack [Jst]` tracing tool to perform continuous trace dumps after a hang bug is reported by existing hang bug detection tools (e.g., [Dai18a; He18]). We then analyze the stack trace to localize the hang function. Intuitively, the hang function will repeatedly appear in the stack trace as the application’s execution gets stuck in the hang function.

Figure 3.9 shows the dumped stack traces of `main` thread when Compress-451 bug is triggered. The trace contains the running status of each thread’s status and the invoked Java functions with the invoking line number. The trace also indicates the call stack of the invoked functions in each thread.

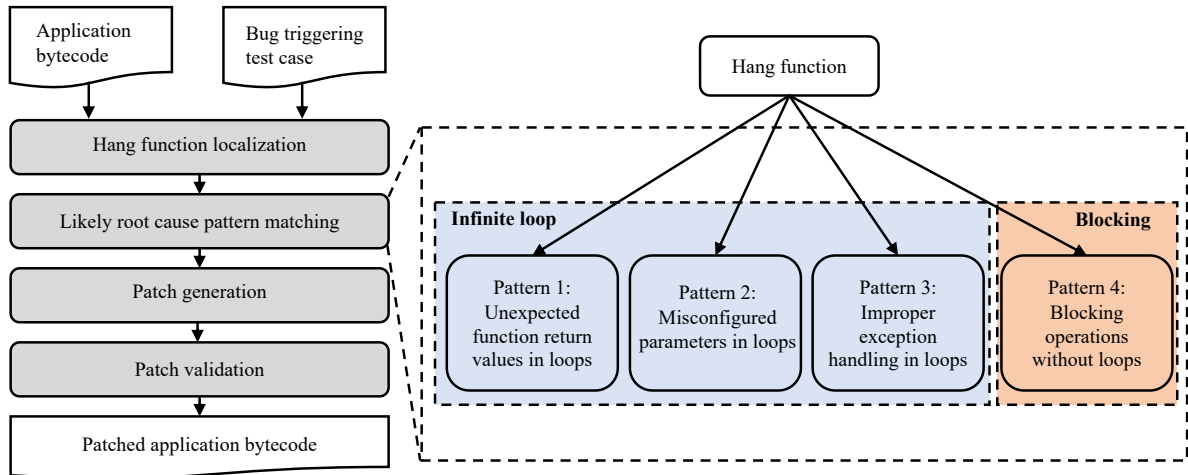


Figure 3.8 The system architecture of HangFix.

```
//Dump 1
"main" #1 prio=5 os_prio=0 tid=0x00007f899c00b000 nid=0x76b9 runnable [0
  x00007f89a27fa000]
 java.lang.Thread.State: RUNNABLE
 at java.io.FileInputStream.readBytes(Native Method)
 at java.io.FileInputStream.read(FileInputStream.java:233)
 at org.apache.commons.compress.utils //hang function
   .IOUtils.copy(IOUtils.java:47)
 at testcode.testCopy(testcode.java:32)
 at testcode.main(testcode.java:12)

//Dump 2
"main" #1 prio=5 os_prio=0 tid=0x00007f899c00b000 nid=0x76b9 runnable [0
  x00007f89a27fa000]
 java.lang.Thread.State: RUNNABLE
 at java.io.FileOutputStream.writeBytes(Native Method)
 at java.io.FileOutputStream.write(FileOutputStream.java:326)
 at org.apache.commons.compress.utils //hang function
   .IOUtils.copy(IOUtils.java:49)
 at testcode.testCopy(testcode.java:32)
 at testcode.main(testcode.java:12)
...
```

Figure 3.9 Continuously dumped stack traces of main thread for the Compress-451 hang bug.

The testcode.java is the bug triggering test case. The stack trace shows that two functions of testcode.java are invoked, i.e., main() and testCopy(). The testcode.testCopy() function invokes the compress.utils.IOUtils.copy() function in the Compress system.

To localize the hang function, we extract the repeated function(s) by comparing the function names among different stack trace dump files. In each thread, we extract the repeated function at

```

//StreamReader.java                                Cassandra-7330(v2.0.8)
73 public SSTableWriter read(ReadableByteChannel channel) throws IOException {
    ...
81   DataInputStream dis = new DataInputStream(new
      LZFINputStream(Channels.newInputStream(channel)));
    ...
96   drain(dis, in.getBytesRead());
    ...
102  }

114 protected void drain(InputStream dis, long bytesRead) throws IOException {
115   long toSkip = totalSize() - bytesRead;
116   toSkip = toSkip - dis.skip(toSkip);
117   while (toSkip > 0) {
118-    toSkip = toSkip - dis.skip(toSkip);
+    long skipped = dis.skip(toSkip);
+    toSkip = toSkip - skipped;
+    if (skipped <= 0){ //immediate termination
+      throw new IOException("Unexpected return"
+        + "value causes the loop stride to be"
+        + "incorrectly updated.");}
119   }
120  }

```

Figure 3.10 Example of hang bug pattern #1 and its fixing strategy. When `InputStream` `dis` is inaccessible or corrupted by bad encoding `dis.skip` can return `-1` or `0`, and `-1/0` is used as the stride. “→” represents the function call invocation. “-” means deleted code and “+” means added code, representing the patch generated by HangFix.

the top of the call stack as the hang function. For example, in the Compress-451 bug stack trace shown in Figure 3.9, HangFix extracts three repeatedly invoked functions: the `testcode.main()`, `testcode.testCopy()` and `compress.utils.IOUtils.copy()` functions. The top of the call stack, that is, the last function on the call path, is identified as the hang function. For example, in the Compress-451 bug, the `compress.utils.IOUtils.copy()` function is extracted as the hang function since it repeatedly appears and is the last hop on the call path. The `compress.utils.IOUtils.copy()` function invokes different statements (i.e., `FileInputStream.read()` and `FileOutputStream.write()`) inside the loop at two different dumps.

3.4.2 Root Cause Pattern Matching and Patch Generation

3.4.2.1 Likely Root Cause Pattern #1: Unexpected Function Return Values in Loops

Root cause pattern description: If the hang function involves loops and the loop stride depends on the return values of some functions which could be either Java library functions (e.g., `InputStream.skip`) or application-specific functions (e.g., `dfs.recoverLease(p)`), we infer that the hang is caused by unexpected function return values. For example, the HBase-8389 bug shown by Figure 3.11 falls in this root cause pattern. For this pattern, HangFix extracts the loop index, stride, and bound abstraction as “`while(index == 0){stride = Method(); index += stride;}`”. The loop stride depends on an application-specific RPC invocation `dfs.recoverLease(p)`, matching this likely root cause pattern.

Patch generation: If the *loop stride impacting* function f_i is a known Java library function, our

```

//FSHDFSUtils.java                                     HBase-8389(v0.94.3)
+private String RECOVERY_TIMEOUT_KEY = "recover.timeout";
+private int DEFAULT_RECOVER_TIMEOUT = 900000;
+private long timeout = conf.getInt(
+ RECOVER_TIMEOUT_KEY, DEFAULT_RECOVER_TIMEOUT);
48 public void recoverFileLease(..., final Path p,
49     ...) throws IOException {
    ...
62 boolean recovered = false;
63 int nbAttempt = 0;
+ long st = System.currentTimeMillis();
64 while (!recovered) {
65     nbAttempt++;
    ...
71     recovered = dfs.recoverLease(p); //send to HDFS
    ...
85     if (!recovered) {
        ...
96         Thread.sleep(nbAttempt < 3 ? 500 : 1000);
        ...
103    }
+ long elapsed = System.currentTimeMillis() - st;
+ if (timeout > 0 && elapsed >= timeout)
+     throw new TimeoutException("Timeout."
+         + "Breaking infinite polling!");
    //delayed termination
104 }}

```

Figure 3.11 When the blocks are corrupted, the `recoverLease()` function keeps polling the recovery results, getting “false” and sending a new recovery request, hanging in an infinite loop. “+” means added code, representing HangFix’s patch for this bug.

patch is to add proper checks over possible incorrect return values and break out of the hang state by throwing an exception that has been declared in the hang function in order to minimize unwanted side-effect from our patch. If function f_i does not contain “throws exception” clause in its signature, HangFix checks the call stack of f_i backwards until it identifies the n -hop caller function of f who declares a checkable exception in its function signature. HangFix then inserts the same checkable exception in the signatures of the function f_i and its i -hop callers, $i = 1, 2, \dots, n - 1$. If there are more than one checkable exceptions, HangFix chooses the first and most specific one.

We choose to fix this type of hang bug through immediate loop termination by throwing exceptions because we observe that Java library functions typically produce persistent errors. For example, the `InputStream.skip(long len)` function can return any value in the following three subsets: $\{-1\} \cup \{0\} \cup [1, |n|]$. However, once it returns -1 (i.e., end of file) or 0 (i.e., corruption), it cannot return any positive values in the subsequent invocations.

If the loop stride impacting function f_i is an application-specific function (e.g., RPC calls to a remote server), our fix is to insert a timeout checking code at the tail of the loop to break out of the loop if the loop execution time exceeds a certain timeout threshold. In contrast to Java library functions which typically have persistent return errors, the errors from application-specific functions can be transient (e.g., file reading errors due to transient network failures). Therefore, instead of terminating the loop right away, HangFix allows the application to retry the function invocation within a certain time limit but prevents the application from entering a hang state by

enforcing the timeout check. Specifically, the timeout checking code consists of an `if` branch with the condition of $var_e \geq var_{new}$. The var_e denotes the elapsed time of the loop's execution while var_{new} is a pre-set timeout variable. Inside the `if` branch, a known exception is thrown with a timeout error message. We will discuss how HangFix picks the right timeout value in Section 3.4.2.4.

Examples: Figure 3.10 shows the patch produced by HangFix for the Cassandra-7330 bug. The loop stride impacting function is the Java library function `InputStream.skip()`. HangFix inserts a check after the invocation of `skip()`, which includes introducing a local variable `skipped` to store the return value, generating an extracted error value set¹ (i.e., ≤ 0), comparing `skipped` with " ≤ 0 ", and throwing an `IOException` which is already defined in the hang function with an error message.

Figure 3.11 shows the patch produced by HangFix for the HBase-8389 bug. The loop stride impacting function is the application-specific function `dfs.recoverLease()`. The patch adds a timeout checking code block at the end of the `while` loop body, which is similar to the manual patch that is produced by the developer shown by Figure 3.1.

3.4.2.2 Likely Root Cause Pattern #2: Misconfigured Parameters in Loops

Root cause pattern description: If the hang function involves loops and the loop stride depends on a configurable, constant variable, we infer the hang is caused by some misconfigured or incorrectly hard-coded values. For example, the Hadoop-15415 bug shown by Figure 3.12 falls in this root cause pattern where the `bufferSize` parameter is misconfigured to be 0 at line #97 and passed in as an argument at line #74. `InputStream.in` performs the `read()` operation on a zero-size byte array and returns zero at line #84, indicating nothing being read from the continuous flow and the end of the flow can never be reached (i.e., $bytesRead < 0$). As a result, the `copyBytes()` function endlessly spins in the loop. HangFix identifies this bug as a root cause pattern #2 bug because the hang function `copyBytes()` at line #74 contains a loop and the loop stride is constantly updated by a configurable parameter `buffSize` in each loop iteration. Specifically, HangFix first retrieves the loop index, stride and bound abstraction for the `InputStream.read(byte[] buf)` function as "`while(index < bound){index += buf.size;}`", where `bound` is the end of the `InputStream`. After analyzing the data dependency flow, HangFix identifies that the stride is misconfigured at line #97-98, causing `bytesRead` to be constantly assigned with zero. Different from Pattern #1, here 0 is a legitimate return value for the `in.read()` function. So the root cause is the misconfigured parameter.

Patch generation: To fix Pattern #2 hang bugs, we add proper checkers over the misconfigured values or arbitrarily hard-coded values before they are used in the loop. HangFix cuts those error values' data flow by throwing a known exception before the fault gets propagated and manifested as system hang, which we call *early termination*. Early termination saves unnecessary resource consumption compared with immediate or delayed termination.

¹HangFix extracts the error value set by analyzing the loop body to infer the error-inducing loop stride values (e.g., ≤ 0) and possible return values of the Java library functions (e.g., $\{-1\} \cup \{0\} \cup [1, |n|]$).

```

//IOUtils.java                                     Hadoop-15415(v2.5.0)
96 public static void copyBytes(InputStream in, ..., Configuration conf) throws
    IOException {
97     int bufferSize =
98     conf.getInt("io.file.buffer.size",4096);
+     if(bufferSize == 0) //early termination
+     throw IOException("Misconfigured bufferSize"
+         + "with 0");
98     copyBytes(in, ..., bufferSize, true);
99 }

49 public static void copyBytes(InputStream in, ..., int bufferSize, boolean close)
    throws IOException {
    ...
52     copyBytes(in, ..., bufferSize);
    ...
65 }

74 public static void copyBytes(InputStream in, ..., int bufferSize) throws
    IOException {
+     if(bufferSize == 0) //early termination
+     throw IOException("bufferSize cannot be 0");
    ...
77     byte buf[] = new byte[bufferSize];
78     int bytesRead = in.read(buf);
79     while (bytesRead >= 0) {
    ...
84     bytesRead = in.read(buf);
85 }}

```

Figure 3.12 Example of hang bug pattern #2 and its fixing strategy. Misconfiguration causes bufferSize to be 0, which in turn makes the InputStream in perform read operation on a zero-size byte array and return 0. “→” represents the function call invocation, while “.....” represents the data dependency flow. “+” means added code, representing the patch generated by HangFix.

During pattern identification, HangFix parses the hang function’s call graph and the error-prone variable’s data dependency flow to locate the misconfiguration (or faulty assignment) statement s_1 . If the error-prone value is not checked properly, HangFix inserts a checker (i.e., c_1) after s_1 . The checker is an if branch with the condition of $v == v_{err}$. If v has more than one error values, HangFix generates a combined condition in the form of $v \geq v_{err_{min}}$ or $v \leq v_{err_{max}}$. Inside the if branch, a known exception is thrown with an error message such as “variable v is misconfigured/falsely assigned with v_{err} , affecting loop stride, leading to an infinite loop.”

In addition, HangFix inserts another checker (i.e., c_2) at the beginning of the hang function before the loop is executed, if 1) the statement s_1 cannot be identified or 2) the error-prone variable is a hang function’s parameter and the hang function is “public”, meaning it can be directly accessed by user-defined classes or classes in other integrated systems. This c_2 checker is similar as the c_1 checker but with a different error message, e.g., “variable v cannot be v_{err} , which can lead to an infinite loop.”

Examples: The Hadoop-15415 bug in Figure 3.12 can be fixed using this patching strategy. HangFix’s pattern identification indicates that the infinite loop happens when the bufferSize is non-positive. HangFix also detects that bufferSize is non-negative because it is used as an array’s

```

//CompactionManager.java      Cassandra-9881(v2.0.8)
436 private void scrubOne(...) throws IOException {
    ...
444     scrubber.scrub();
    ...
459 }

//Scrubber.java
103- public void scrub(){
+ public void scrub() throws IOException {
    ...
120     while (!dataFile.isEOF()){
        ...
+     int index = 0;
129     try{
130         key = sstable.partitioner.decorateKey(ByteBufferUtil.readWithShortLength(
            dataFile));
+         index += 3;//trace index change
        ...
134         dataSize = dataFile.readLong();
+         index += 8;//trace index change
        ...
139     } catch (Throwable th){
140         ...; //ignore Exception
+         if(index == 0) //no index update,
+         throw th; //immediate termination
141     }
    ...
}}

```

Figure 3.13 Example of hang bug pattern #3 and its fixing strategy. Data corruption causes `readWithShortLength()` to throw exception at line #130-131, which makes the loop skip the index updating statement (i.e., zero-stride) at line #134. “ \rightarrow ” represents the function call invocation, while “ $- \rightarrow$ ” represents the control flow. “-” means deleted code and “+” means added code, representing the patch generated by HangFix.

size at line #77. Intersecting “non-positive” with “non-negative,” HangFix generates the error-prone value as $\{0\}$. After detecting that `buffSize` is passed in as a parameter and configured at line #97-98, HangFix inserts a checker after line #98, including an `if` branch with condition “`buffSize == 0`” and an `IOException` with error message “misconfigured `buffSize` with 0.” Since the `copyByte()` function at line #74 is public, it can be accessed by user-defined classes or classes in any other integrated systems (e.g., HBase, Hive, etc). The argument value of `buffSize` passed in from those classes are inaccessible in our analysis currently. Thus, another checker inside the `copyByte()` function before the loop is necessary. After line #74, HangFix inserts an `if` branch with condition “`buffSize == 0`” and an `IOException` with error message as “`buffSize` cannot be 0.”

3.4.2.3 Likely Root Cause Pattern #3: Improper Exception Handling in Loops

Root cause pattern description: If the hang function involves loops and the loop stride update is skipped due to some exceptions, we infer the hang is caused by improper exception handling in loops. For example, the Cassandra-9881 bug shown by Figure 3.13 falls in this root cause pattern. When `dataFile` is corrupted, an `IOException` is raised by the `readWithShortLength()` function at line #130. The `IOException` gives up the correct execution of `readWithShortLength()` and skips

the `readLong()` function at line #134. The above read functions are both loop index-forwarding operations. This `IOException` is then simply ignored at line #140. Without moving the loop index forwards at line #130 and #134, the `scrub()` function keeps reading from the location, spinning forever. HangFix identifies this bug as a Pattern #3 bug, because in the exception handling control flow path `120→129→130→139→140→141`, there is no loop stride update along the path. Specifically, HangFix first extracts all the invocations of the `DataInput` instance `dataFile`, including `isEOF()` at line #120, `readUnsignedShort()` and `readByte()`² at line #130, and `readLong()` at line #134. HangFix then generates the loop index, stride, and bound abstraction for the above `DataInput` functions as `while(index < bound){index += 2; index += 1; index += 8;}`. Moreover, all the abstractions for the loop stride (i.e., `index += 2` and `index += 1` at line #130, `index += 8` at line #134) do not appear in the exception handling control flow, matching this root cause pattern.

Patch generation: To fix this type of hang bugs, HangFix conducts *index tracing* to check whether the loop has no stride or ineffective loop stride. If the index updating operation is not correctly executed (i.e., the loop index does not change) at one iteration, this loop does not contain effective stride along this iteration’s control flow. HangFix inserts a counter variable `index` with the value of `v` at the beginning of a loop. It then extracts the loop index and bound abstraction (i.e., `index += stride`) for each index updating operation, and inserts such abstraction after the operation.

If the loop index is not updated in the exception or error handling control flow, HangFix first tries to fix the hang bug by re-executing the loop index updating operations to restore loop stride properly. Specifically, HangFix inserts a checker inside the exception handling block (e.g., `catch` block) or after the error return statement. The checker is an `if` branch with condition of `index == v`, inside which, HangFix inserts the unexecuted index-updating operations. If the loop has multiple index-updating operations, HangFix chooses the first one in each control flow from the loop header to the exception/error handling block. The chosen operation is inserted with the conditions along its control flow. For example, for the following control flow `if (cond){op1();op2();} else{op3();}`, HangFix inserts `if (index == v){if (cond){op1();} else{op3();}}` in the checker. If the loop stride restoration fails to fix the hang bug, HangFix terminates the loop with a known exception to break out of the hang state.

Examples: Figure 3.13 shows the patch produced by HangFix for the Cassandra-9881 bug. HangFix starts index tracing by inserting a counter variable `index` with the original value 0 before the `try` block at line #129. HangFix first identifies that the `readWithShortLength()` function at line #130 and the `dataFile.readLong()` function at line #134 are index updating operations and can be abstracted as `index += 3` and `index += 8`, respectively. HangFix then inserts these abstractions after line #130 and #134. In the exception handling block at line #139-141, HangFix inserts an `if` branch with the condition of `index == 0` and a `throw` exception statement to immediately terminate the loop. This is doable because HangFix adds the `throws IOException` clause in the `scrub()` function’s signature at line #103 and relies on Cassandra’s existing exception handling mechanisms

²`readUnsignedShort()` and `readByte()` are the callees of `readWithShortLength()`. To save space, we omit them in Figure 3.13.

```

//ZlibCodec.java                                     Hive-5235(v1.0.0)
81 public void decompress(ByteBuffer in, ByteBuffer out) throws IOException {
93   try {
94-    int cnt = inflater.inflate(out.array(),
+    int cnt = inflateWithTO(inflater, out.array(),
95                          out.arrayOffset() + out.position(),
96                          out.remaining());
    ...
97   } catch (DataFormatException e) {
98     throw new IOException("Bad compressed data",e);
99   }
    ...
105 }

+private Configuration conf = new Configuration();
+private String INFLATE_TIMEOUT_KEY = "orc.zlibcodec.inflate.timeout";
+private long DEFAULT_INFLATE_TIMEOUT = 5000;
+private long timeout = conf.getLong(INFLATE_TIMEOUT_KEY, DEFAULT_INFLATE_TIMEOUT)
    ;

//a callable thread with timeout setting
+public int inflateWithTO(final Inflater inflater, final byte[] b, final int off,
    final int len) throws DataFormatException {
+ ExecutorService executor =
+     Executors.newSingleThreadExecutor();
+ Callable<Integer> callable=new Callable<Integer>(){
+   @Override
+   public Integer call() throws DataFormatException {
+     return inflater.inflate(b, off, len);
+   }
+ };
+ Future<Integer> future = executor.submit(callable);
+ int cnt = 0;
+ try {
+   //timeout setting
+   cnt = future.get(timeout, TimeUnit.MILLISECONDS);
+ } catch (Exception e) {
+   future.cancel(true); //acceptable exception
+   throw new DataFormatException("Endless blocking");
+ } finally { executor.shutdown(); }
+ return cnt;
+}

```

Figure 3.14 Example of hang bug pattern #4 and its fixing strategy. `Inflater.inflate()` is a blocking-pone function. When an ORC file is corrupted, conducting the `inflate()` operation on a corrupted file causes an infinite loop in the underlying JNI code. “→” represents the function call invocation. “-” means deleted code and “+” means added code, representing the patch generated by HangFix.

to repair failures by propagating the exception backwards to the caller function, `scrubOne()`, and we refer to it as *exception heritage*. HangFix does not re-execute the index updating operation inside the `catch` block because re-executing the operation (i.e., `readWithShortLength`) will still throw exceptions due to the corrupted `dataFile` and Cassandra will still hang.

3.4.2.4 Likely Root Cause Pattern #4: Blocking Operations Without Loops

Root cause pattern description: If the hang function does not contain any loop and the hang function consists of some blocking operation (e.g., Java library functions, JNI methods), HangFix infers that the hang is caused by blocking operations. For example, the Hive-5235 bug shown in

Figure 3.14 falls into this root cause pattern. The hang bug is triggered when the hang function `decompress` invokes a blocking Java library function `inflate()`. The underlying JNI code of the library function hangs in an infinite loop when it is invoked over a corrupted ORC file. HangFix classifies this bug as a Pattern #4 bug since the hang function `decompress()` does not contain any loop but one blocking Java library function call.

Patch generation: To fix hang bugs caused by blocking operations, HangFix first isolates the blocking operation from the main application execution using a callable or runnable thread. Next, HangFix adds a timeout check to end the blocking operation after a certain waiting period defined by var_{new} (e.g., `future.get(var_{new})`, `thread.join(var_{new})`). The newly introduced timeout variable var_{new} is first configured with the default value of a known timeout variable v . HangFix extracts the default value by searching the system’s configuration files using keywords, such as “timeout,” “interval,” “block,” and “poll.” HangFix chooses the variable which matches the most keywords, and assigns its default value v to var_{new} . The rationale is that variables share similar names most likely have similar purposes, thus similar default values. During the patch validation phase, we can adjust the timeout variable values if the patch does not pass any test. We can also leverage timeout value prediction techniques (e.g., [He19]) to infer the timeout values more efficiently.

Examples: Figure 3.14 shows the patch produced by HangFix for fixing the Hive-5235 bug. HangFix first introduces the `timeout` variable with the default value of 5000 milliseconds. This default value is read from the `HiveConf.HIVE_SERVER2_LONG_POLLING_TIMEOUT` variable. HangFix replaces the blocking operation `Inflater.inflate()` at line #94 with a new function called `inflateWithTO()` isolated by a callable thread. The timeout setting is in the `future.get()` function with the timeout variable `timeout`. To break out of the blocking state, the function uses a known exception type `DataFormatException` which has been used by this hang function with a useful log message “endless blocking.” So the patch provided by HangFix can not only prevent the service outage caused by the hang bug but also provide useful information for the developer to know the root cause of the auto-patched hang bug.

3.4.3 Discussion

We focus on hang bug fixing, rather than hang bug detection. We rely on previous work [Dai18a; He18] to detect hang bugs. It is possible that hang bug detection tools raise false alarms, which is not the focus of this paper.

We pick those four likely root cause patterns to implement in HangFix based on previous hang bug study results [Dai18b; Dai18a; Dea15b; Gun14; He18] as well as our past experiences. However, the root causes of hang bugs are definitely not limited to those patterns only. We have conducted an empirical hang bug study to understand the representativeness of our root cause patterns in real production hang bugs. Our empirical study shows that HangFix root cause patterns can cover all of the hang bugs which are not related to synchronizations. We will describe our hang bug root cause study results in detail in Section 3.5.

As mentioned in the introduction, we validate the patch produced by HangFix using existing bug

detection tools, our hang function localization tool, and the application’s regression test suites. Due to the complexity of cloud computing environments and modern server systems, HangFix currently does not provide any theoretical proof on the correctness and completeness of the auto-patched code. Although HangFix proactively takes cautious steps (e.g., reuse existing exceptions) to avoid unwanted effects, HangFix cannot guarantee the automatically generated patches do not bring any side effect to the application especially when the hang function involves application-specific stateful operations. One key objective of our empirical bug study described in Section 3.5 is to understand the coverage of our root cause patterns for real world production hang bugs.

3.5 Experimental Evaluation for Fixing Hang Bugs

In this section, we present our evaluation results on HangFix. Our evaluation consists of two parts: 1) an empirical study over 237 real hang bugs; and 2) an experimental evaluation over 42 real hang bugs that can be reproduced by us successfully. We first describe our evaluation methodology followed by detailed results and analysis.

3.5.1 Evaluation Methodology

Cloud systems: We collect hang bugs from the bug-tracking systems, e.g., Apache Jira [Jir], for 10 open source cloud systems: Cassandra key-value store, Compress I/O compression library, Hadoop common library, Hadoop Mapreduce big-data processing framework, Hadoop HDFS file system, Hadoop Yarn resource management service, HBase database management system, Hive data warehouse, Kafka streaming platform, and Lucene text searching engine. These 10 systems are representatives of popular open source production systems used in cloud environments. They cover a wide range of different systems, varying from distributed big data processing to log search.

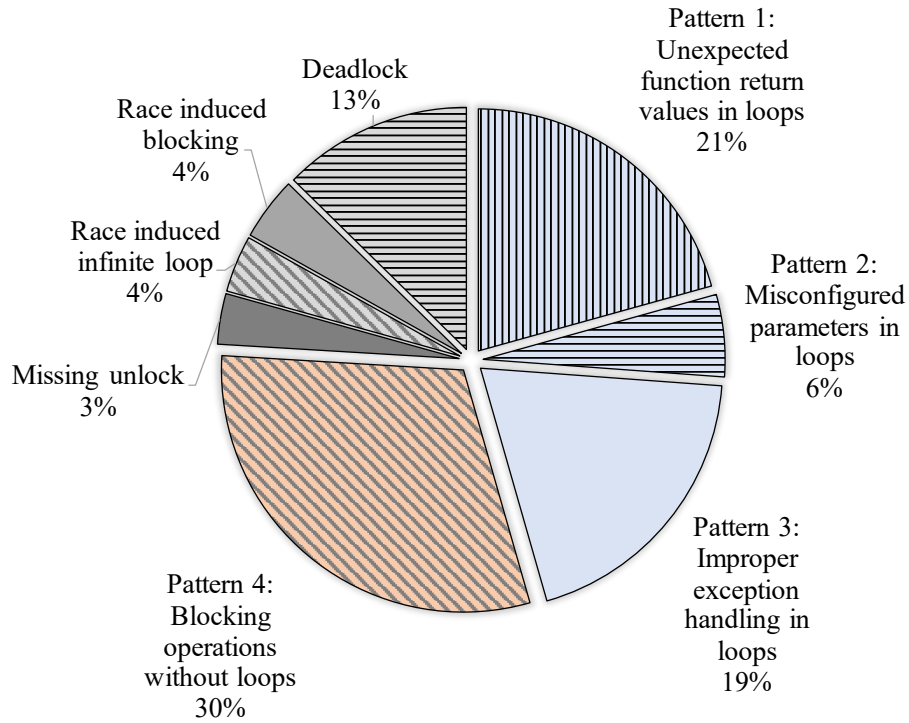
Benchmarks: We use the “hang”, “stuck” and “block” keywords to search for hang bugs. However, those keywords can appear in bug descriptions, test cases (the bug causes test cases hanging), or patch evaluation (the proposed patch causes system hanging). Therefore, we manually examine each bug to determine whether it is a real hang bug triggered in production environments. We include both fixed and open bugs, but eliminate bugs labeled as “Not a problem” or “Will not fix.” We target bugs occurred in production systems and eliminate the bugs in the test suites. To the best of our efforts, we collect 237 bugs in total from the 10 commonly used server systems.

Root cause pattern matching and patching evaluation in the empirical study: After collecting these real-world software hang bugs, we manually study each of them to match it with HangFix’s likely root cause patterns. Based on our understanding of the root cause of the bug, we analyze whether HangFix’s patch can fix the bug completely without introducing new bugs, i.e., we check whether HangFix can fix the bug and does not influence the code logic after jumping out of the infinite loop or blocking. If a manual patch exists for a bug, we compare the manual patch with HangFix’s patch. If the bug is not fixed yet or the manual patch is different from HangFix’s patch, we analyze the root cause and the patch of the bug to check whether HangFix’s patch introduces new

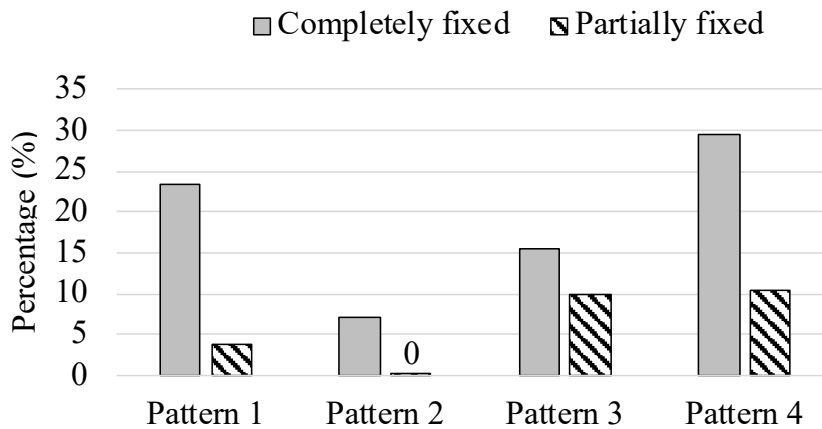
bugs or side effects.

Setup: All the experiments were conducted in our lab machine with an Intel® i7-4790 Octa-core 3.6GHz CPU, 16GB memory, running 64-bit Ubuntu v16.04 with kernel v4.4.0.

Implementation: HangFix is implemented on top of Soot compiler [Soo] in Java, using BodyTransformer and SceneTransformer to conduct intra- and inter-procedural analysis, and ForwardFlowAnalysis to conduct data flow dependency analysis.



(a) Root cause pattern distribution.



(b) Percentage of completely and partially fixed bugs.

Figure 3.15 The pattern matching and fixing results of the 237 hang bugs in our empirical study. 76% of them fall into HangFix's four root cause patterns. 75% of them can be completely fixed.

3.5.2 Empirical Study Results

Figure 3.15a shows the results of pattern matching results of the 237 studied bugs. As shown in the figure, 76% bugs, i.e., a total of 180, fall into HangFix's four patterns. The other 24% bugs are related to synchronization and concurrency.

3.5.2.1 Negative Case Study

Figure 3.15b shows the fixing results of the 180 bugs which fall into HangFix's four fixable patterns. 136 out of the 180 bugs can be fixed by HangFix completely. For the 44 bugs partially fixed by HangFix, their manual patches contain application-specific operations or it is required to restore system's state to fix the bugs.

We describe three bug examples, which cannot be fixed by HangFix. We analyze the root cause of the bugs and explain why HangFix's patch is not complete.

Yarn-3999 (Pattern 1): The bug occurs when Yarn system is draining events to an external system, e.g., Zookeeper. When the external system becomes very slow or unresponsive, the Resource Manager hangs on waiting to flush all the events into the external system. When all the events are flushed, the Resource Manager transitions to the STANDBY state. Due to the hang on draining events, the Resource Manager's transition to the STANDBY state cannot be completed, expiring all current applications on the Yarn system. HangFix identifies the bug as Pattern 1 because the bug hangs on an infinite loop and the loop stride depends on a function's invocation. HangFix's patch can ensure the Yarn system will jump out of the draining events loop by adding a timeout, even if not all the events are flushed. However, the Resource Manager still cannot enter STANDBY state. In this case, the Resource Manager cannot work after applying HangFix's patch.

Kafka-1238 (Pattern 3): The bug occurs when a client updates metadata of a Kafka cluster. The Kafka cluster updates the cluster after the metadata is received from the client each time. However, if none of the nodes are alive, the Kafka cluster reports an error and the client retries the updating request endlessly in an infinite loop. The reason is that at least one node needs to be alive to update the whole Kafka cluster. HangFix identifies the bug as the Pattern 3 because loop stride update is skipped. HangFix's patch enables the client to jump out of the requesting infinite loop. However, the Kafka cluster has no live node and the whole Kafka cluster cannot work after applying HangFix's patch. In comparison, the manual patch ensures that there is always at least one live node in the whole Kafka cluster.

HBase-8729 (Pattern 4): This bug occurs when multiple SSH handlers are replaying logs. When the assigned Region Server of one of the handlers fails, log replaying hangs because the handler keeps waiting on the response of the dead server. HangFix identifies the bug as the Pattern 4 bug because log replaying is a blocking call. HangFix's patch terminates the log replaying call directly by throwing the exception. Compared with HangFix's patch, the manual patch enables the handler to re-route the regions to another live Region Server. Therefore, log replaying job moves forward on the newly assigned Region Server.

3.5.2.2 Synchronization-Related Bug Patterns

Besides HangFix's four patterns, we found four other bug patterns that are related to synchronization and concurrency, i.e., missing unlock, race-induced infinite loop or blocking, and deadlock.

Missing unlock: These bugs are caused by programming mistakes on synchronization operations. When one thread is holding a lock but does not release the lock upon unexpected failures, other threads are hanging on acquiring on the lock. When missing unlock bugs happen, we observe multiple threads blocked and they all wait for the lock with the same lock ID. The safest way to fix this kind of bug is to release the lock, while terminating the blocked threads cannot fix the bug completely.

Race-induced infinite loop or blocking: The root causes of these bugs are race conditions, which further causes infinite loop or blocking. HangFix cannot ensure to fix such bug completely as currently we do not target hang bugs with race conditions as their root causes. For example, in HBase-16211 bug, if clearing JMX cache and injecting the data sink are done at the same time, the two operations access the cache simultaneously, causing the race condition. The consequence is that the injected sink lost, further leading to an reading data operation hanging. HangFix's patch jumps out of blocking reading operation, without fixing the data loss or the race condition nevertheless. The correct fix for the bug is to add a lock for cache writing operations.

Deadlock: Deadlock bugs are easy to observe through stack traces. Two threads are in BLOCKED states and they are holding different locks while they are waiting for the lock held by the other thread. HangFix's patch can terminate the deadlock but it cannot fix the deadlock.

3.5.3 Experimental Results

To the best of our abilities, we reproduce 42 bugs falling into HangFix's four patterns, as shown in Table 3.7. We list the buggy system version and the detailed description. Through the empirical study, 40 out of the 42 bugs can be completely fixed. We further validate them in our experiments. Additionally, we reproduce two partially fixed bugs and present the experimental results to illustrate why HangFix cannot fix them completely.

As shown in Table 3.8, HangFix successfully fixes 40 out of 42 hang bugs completely in our benchmarks, including 15 bugs in Pattern #1, 13 bugs in Pattern #2, six bugs in Pattern #3, and eight bugs in Pattern #4.

Our experiments show that for the 40 fixed bugs, the hang bug localization tool and existing bug detection tools do not raise alarms and the patched program executes successfully without hanging or crashing using the regression test suites. For the two partially bugs, HangFix cannot restore the system state or corrupted data. In contrast, 14 out of the 42 bugs are fixed by developers with manual patches, while the remaining bugs are still open or pending for developers' check to merge patches.

To further evaluate the patches generated by HangFix, we compare them with the manual patches for the 14 fixed bugs by both approaches. We find that HangFix's patches are similar as the

Table 3.7 42 reproducible hang bug benchmarks. Although some bugs have the same description, they happen in different functions or classes.

Bug name	Version	Description
Cassandra-7330	v2.0.8	Skipping on a corrupted <code>InputStream</code> returns error code, affecting loop stride.
Cassandra-9881	v2.0.8	Improper exception handling skips loop index-forwarding API.
Compress-87	v1.0	Reading on a truncated zip file returns error code, affecting loop stride.
Compress-451	v1.0	Misconfigured variable <code>bufferSize</code> indirectly affects loop index.
Hadoop-8614	v0.23.0	Skipping after EOF returns error code, affecting loop stride.
Hadoop-15088	v2.5.0	Skipping on a corrupted <code>InputStream</code> returns error code, affecting loop stride.
Hadoop-15415	v2.5.0	Misconfigured variable <code>buffSize</code> indirectly affects loop index.
Hadoop-15417	v2.5.0	Misconfigured variable <code>bufferSize</code> indirectly affects loop index.
Hadoop-15424	v2.5.0	Misconfigured variable <code>buff</code> causes loop stride be 0.
Hadoop-15425	v2.5.0	Misconfigured variable <code>sizeBuf</code> indirectly affects loop index.
Hadoop-15429	v2.5.0	Unsynchronized index is set and reset periodically, causing <code>DataInputByteBuffer</code> hangs.
HDFS-4882	v0.23.0	Corruption handling causes loop index update operation skipped.
HDFS-5438	v0.23.0	Incorrect block report processing causes corrupted replicas to be accepted during commit.
HDFS-10223	v2.7.0	<code>TcpPeerServer</code> endlessly waits for a response from an unresponsive <code>DataNode</code> .
HDFS-13513	v2.5.0	Misconfigured variable <code>BUFFER_SIZE</code> indirectly affects loop index.
HDFS-13514	v2.5.0	Misconfigured variable <code>BUFFER_SIZE</code> indirectly affects loop index.
HDFS-14481	v2.5.0	Misconfigured variable <code>BUFFER_SIZE</code> causes loop stride be 0.
HDFS-14501	v2.5.0	Misconfigured variable <code>BUFFER_SIZE</code> causes loop stride be 0.
HDFS-14540	v0.23.0	Block deletion failure causes an infinite polling.
Mapreduce-2185	v0.23.0	Improper error handling causes the loop index updating operation skipped.
Mapreduce-5066	v2.0.3	<code>JobEndNotifier</code> endlessly waits for a response from an unresponsive Hadoop job.
Mapreduce-6990	v0.23.0	Skipping on a corrupted <code>InputStream</code> returns error code, affecting loop stride.
Mapreduce-6991	v2.5.0	File creation failure and improper exception handling skips loop index-forwarding API.
Mapreduce-7088	v2.5.0	Misconfigured variable <code>bufferSize</code> causes loop stride be 0.
Mapreduce-7089	v2.5.0	Misconfigured variable <code>bufferSize</code> causes loop stride be 0.
Yarn-163	v0.23.0	Skipping on a corrupted <code>FileReader</code> returns error code, affecting loop stride.
Yarn-1630	v2.2.0	<code>YarnClient</code> endlessly polls the state of an asynchronous application.
Yarn-2905	v2.5.0	Skipping on a corrupted aggregated log file returns error code, affecting loop stride.
HBase-8389	v0.94.3	HBase endlessly sends lease recovery requests to HDFS but HDFS fails on recovery.
Hive-5235	v1.0.0	Uncompressing a corrupted ORC file blocks the Hive task.
Hive-13397	v1.0.0	Reading on a corrupted ORC file returns error code, affecting loop stride.
Hive-18142	v1.0.0	Reading on a corrupted ORC file returns error code, affecting loop stride.
Hive-18216	v2.3.2	<code>bytesToPoint</code> function returns error code and skips loop index-forwarding API.
Hive-18217	v2.3.2	<code>bytesToPoint</code> function returns error code and skips loop index-forwarding API.
Hive-18219	v2.3.2	Skipping on a corrupted <code>InputStream</code> returns error code, affecting loop stride.
Hive-19391	v1.0.0	<code>RowContainer</code> endlessly retries to create a file but failed.
Hive-19392	v1.0.0	Unsynchronized index is set and reset periodically, causing <code>DataInputByteBuffer</code> hangs.
Hive-19395	v1.0.0	Misconfigured variable <code>bufferSize</code> causes loop stride be 0.
Hive-19406	v2.3.2	<code>HiveKVResultCache</code> endlessly retries to create a file but failed.
Kafka-6271	v0.10.0	Skipping on a corrupted file returns error code, affecting loop stride.
Lucene-772	v2.1.0	Index corruption causes Lucene stuck on uncompression task.
Lucene-8294	v2.1.0	Misconfigured variable <code>bufferSize</code> causes loop stride be 0.

manual ones except for the Compress-451 bug. Its manual patch throws runtime exceptions after identifying the `bufferSize` variable is misconfigured to be non-positive, while HangFix throws an `IOException` which can be properly handled by the Compress system without interrupting the program's execution. We applied the manual patch on the buggy Compress program, ran our bug-triggering test case, and found that the program with the manual patch crashed after we triggered the bug.

HangFix fixed the 40 bugs completely in seconds. HangFix supports inter-procedural analysis using either iterative `BodyTransformer` or `SceneTransformer` from Soot, which decides the fixing time of the bugs. `BodyTransformer` is faster but can fail sometimes while `SceneTransformer` is relatively slower but always succeeds. HangFix first generates the patch using iterative `BodyTransformer` and evaluates the patch. If the patched code cannot fix the bug, HangFix then uses the `SceneTran`

sformer approach. For example, HangFix generates the patch for the Hadoop-15417 bug using SceneTransformer approach, which results in a longer fixing time (22 seconds) than other bugs. Note that it usually takes several weeks or even longer for developers to manually fix the bugs. It might be unfair to directly compare HangFix's patch generation time with the bug's manual resolve time because HangFix is an automatic fixing tool while manual patches involve human efforts. However, we believe that HangFix can help developers to efficiently fix hang bugs in massive cloud systems.

After adopting HangFix's patch, we measure the additional performance overhead by running the same workload again. We observe that the overhead of HangFix's patch is within 1%. Compared with the original root cause function, HangFix's patch only adds a checker (e.g., return value checker or configuration parameter checker) to the function, which imposes little overhead.

3.5.3.1 Two Partially Fixed Bugs

We discuss the two bugs partially fixed by HangFix, i.e., HDFS-4882 and HDFS-5438, in detail. HangFix cannot fix them completely because HangFix cannot restore corrupted data, which can further cause other hang problems.

```

//LeaseManager.java                                HDFS-4882(v0.23.0)
369 public void run() {
370     for(; fsnamesystem.isRunning(); ) {
        ...
374     checkLeases();
        ...
388 }}

393 private synchronized void checkLeases() {
        ...
395 for(; sortedLeases.size() > 0; ) {
396     final Lease oldest = sortedLeases.first();
        ... //p is a file's lease path
+ int index = oldest.getPaths().size();
412 if(fsnamesystem.internalReleaseLease(p, ...)) {
413     LOG.info("...");
414     removing.add(p); //remove p from sortedLeases
+     index -= 1;
416 } else {
417     LOG.info("...block recovery for file " + p);
418 }
+ if(index == 0) removing.add(p); //restore stride
        ...
429 }}

```

Figure 3.16 Example of a pattern #3 hang bug which cannot be fixed by HangFix. A corrupted file *f* associated with the lease path *p* makes the `internalReleaseLease` function fail for recovering the lease for *f*. When it happens, *p* is not removed from `sortedLeases` (skip updating loop index), LeaseManager keeps recovering lease for the file *f* endlessly. “→” represents the function call invocation, while “- ->” represents the control flow. “+” means added code, representing the patch generated by HangFix.

HDFS-4882 (Pattern 3): As shown by Figure 3.16, HangFix inserts the loop index updating operations for stride restoration in the patch after line #418. HangFix’s patch enables HDFS system to jump out of the infinite loop inside the hanging function `checkLeases()`. However, the outer loop between line #370 and line #388 is processing the data. Since HangFix cannot restore the corrupted data, HDFS falls into another infinite loop to process the corrupted file block endlessly.

HDFS-5438 (Pattern 4): HangFix successfully inserts the timeout settings to break an infinite polling loop. However, this patching strategy can only move forward the hanging function `completeFile()`. It cannot restore the corrupted blocks in the pipeline recovery. As a result, corrupted replicas are accepted causing another missing unlock problem in the `DFSInputStream.fetchBlockByteRange()` function.

3.6 Summary

In this chapter, we have presented TFix and HangFix, an automatic timeout bug fixing system and a new hang bug fixing framework. TFix employs a new drill-down analysis framework for narrowing

down the root cause of the misused timeout bug and recommending bug fix. The drill-down analysis of TFix consists of four major steps: 1) checking whether the detected bug is a misused timeout bug by matching common timeout related functions in different server systems; 2) identifying abnormal functions which are affected by the timeout bug using application performance tracing; 3) pinpointing the root cause timeout variable using static taint analysis; and 4) recommending proper timeout values based on the performance tracing results during normal runs. We have implemented a prototype of TFix and evaluated it using 13 real world timeout bugs in a set of commonly used server systems (e.g., Hadoop, HBase, Flume). The experimental results show that TFix can produce effective fix for all the tested misused timeout bugs. TFix is lightweight and imposes less than 1% runtime overhead, which makes it practical for automatically fixing timeout bugs in production systems.

HangFix is a new hang bug fixing framework for automatically patching a hang bug that is detected in production cloud environments. HangFix leverages both dynamic and static analysis techniques to localize hang functions and identify likely root cause patterns. HangFix then generates corresponding patches to fix the hang bug automatically. We have implemented a prototype of HangFix and evaluated it on 42 real-world software hang bugs in 10 commonly used cloud server systems. HangFix successfully fixes 40 out of 42 hang bugs within seconds. We have also conducted an empirical study over 237 real world hang bugs and found that our likely root cause patterns cover 76% of the 237 bugs and the rest 24% bugs are either synchronization or concurrency bugs. HangFix does not require application source or any application-specific knowledge, which makes it practical for production systems.

Table 3.8 The comparison of HangFix and manual fixing.

Bug name	Manual	HangFix		
	Fixed?	Fixed?	Root cause pattern type	Fixing time (sec)
Cassandra-7330	✓	✓	#1	1.2±0.2
Compress-87	✓	✓	#1	1.1±0.1
Hadoop-8614	✓	✓	#1	0.7±0.1
Hadoop-15088	✗	✓	#1	1.0±0.1
Hadoop-15424	✗	✓	#1	0.9±0.1
Hadoop-15425	✗	✓	#1	1.1±0.1
Mapreduce-6990	✗	✓	#1	0.8±0.1
Yarn-163	✗	✓	#1	0.9±0.0
Yarn-1630	✓	✓	#1	1.1±0.1
Yarn-2905	✓	✓	#1	0.8±0.0
HBase-8389	✓	✓	#1	0.9±0.0
Hive-13397	✓	✓	#1	0.8±0.1
Hive-18142	✗	✓	#1	0.9±0.1
Hive-18219	✗	✓	#1	1.0±0.1
Kafka-6271	✗	✓	#1	0.9±0.0
Compress-451	✓	✓	#2	0.8±0.1
Hadoop-15415	✗	✓	#2	0.9±0.1
Hadoop-15417	✗	✓	#2	22±1.0
Hadoop-15429	✗	✓	#2	0.8±0.1
HDFS-13513	✗	✓	#2	0.9±0.1
HDFS-13514	✗	✓	#2	1.1±0.1
HDFS-14481	✗	✓	#2	0.7±0.0
HDFS-14501	✗	✓	#2	0.8±0.1
Mapreduce-7088	✗	✓	#2	1.0±0.1
Mapreduce-7089	✗	✓	#2	0.8±0.0
Hive-19392	✗	✓	#2	0.9±0.1
Hive-19395	✗	✓	#2	1.0±0.0
Lucene-8294	✓	✓	#2	1.0±0.1
Cassandra-9881	✗	✓	#3	0.9±0.1
HDFS-4882	✓	✗	#3	-
Mapreduce-2185	✓	✓	#3	1.3±0.2
Mapreduce-6991	✗	✓	#3	1.2±0.1
Hive-18216	✗	✓	#3	1.2±0.2
Hive-18217	✗	✓	#3	1.1±0.2
HDFS-10223	✓	✓	#4	1.0±0.1
HDFS-5438	✓	✗	#4	-
HDFS-14540	✗	✓	#4	1.1±0.1
Mapreduce-5066	✓	✓	#4	0.9±0.1
Hive-5235	✗	✓	#4	0.8±0.1
Hive-19391	✗	✓	#4	1.2±0.2
Hive-19406	✗	✓	#4	0.8±0.1
Lucene-772	✗	✓	#4	0.7±0.0

CHAPTER

4

EXTRACTING PERFORMANCE BUG SIGNATURES VIA MULTI-MODALITY ANALYSIS

In this chapter, we present the performance bug signature extraction framework called PerfSig. We introduce the motivation followed by the detailed design and experimental evaluation.

4.1 Introduction

Cloud systems are becoming increasingly complex, which dramatically increase the occurrence chance of various software bugs. We use performance bugs to refer to those software bugs which cause cloud systems to get stuck in a hang state or experience performance slow down. Performance bugs triggered in production cloud environments are notoriously difficult to diagnose and fix due to the lack of diagnostic information. When a performance bug occurs in production cloud environments, system operators and developers often need to put a lot of manual efforts to diagnose and fix the problem under time pressure. For example, it took more than 12 hours for Amazon to recover its membership service outage caused by a performance bug [Awsa]. The bug was triggered by a limit on the allowable thread count, that is, the server hung when the number of server threads exceeded its pre-defined limit.

During our empirical bug study using popular bug repositories such as Jira and Bugzilla [Jir; Bug], we observe that many performance bugs repeatedly occur in different versions of open source

```

//RPC class
341 public static <T> ProtocolProxy<T>
    waitForProtocolProxy(...) throws IOException {
-   return waitForProtocolProxy(...,0,...);
+   return waitForProtocolProxy(...
+       ,getRpcTimeout(conf),...);
346 }

+   public static int getRpcTimeout
        (Configuration conf) {
+   return conf.getInt(CommonConfigurationKeys
+       .IPC_CLIENT_RPC_TIMEOUT_KEY,
+       CommonConfigurationKeys
+       .IPC_CLIENT_RPC_TIMEOUT_DEFAULT);
+   }

```

Figure 4.1 The code snippet of the Hadoop-11252 Bug. The buggy code is invoked at the DataNode.

systems, which causes the community to perform redundant debugging over the same bug. Moreover, micro-services using containers [Doc] make the bug replication easier than ever – the same bug occurs in multiple containers that are created from the same container image. To this end, developing an automatic tool to extract signatures for different performance bugs can help operators identify repeatedly occurring bugs and provide useful diagnostic information for developers.

Previous work [Coh05; Coh04; Dea14; Aru13] mainly focuses on depicting performance bugs via analysis over single data type such as system metrics, system calls, system logs, or performance counters. However, a performance bug may manifest as anomalies in different data types. For example, an infinite loop bug may cause a persistently high CPU usage while a timeout bugs can cause abnormal log sequences. Thus, it is likely that we may fail to extract bug signatures for some performance bugs if we only focus on analyzing one data type. Moreover, extracting anomalies alone often cannot uniquely characterize a performance bug because different performance bugs may exhibit similar anomaly patterns in one data modality. For example, different infinite loop bugs can all show increased CPU consumption. So it is necessary to perform multi-modality analysis to extract representative signatures for different performance bugs.

Previous work [Du17; Zha14b; Yu16b] has several limitations in depicting performance bugs occurred in production cloud systems. First, the existing tools often require domain knowledge extracted from the source code or binary code. However, such information is not easily accessible in production systems. Second, the existing tools are not application-agnostic. Third, the existing tools can incur high overhead to cloud systems. Fourth, limited contribution has been done in investigating distributed bugs occurred in cloud systems. Therefore, it is essential to design an application-agnostic, light-weight performance bug signature extraction tool that can be applied to different distributed cloud systems without requiring domain knowledge.

```
... 12:04:53 INFO ipc.Server: Starting Socket Reader #1 for port 43423
... 12:04:54 INFO ipc.Server: IPC Server Responder: starting
... 12:04:54 INFO ipc.Server: IPC Server listener on 43423: starting
... 12:04:56 INFO ipc.Server: Stopping server on 43423
... 12:04:56 INFO ipc.Server: Stopping IPC Server listener on 43423
... 12:04:56 INFO ipc.Server: Stopping IPC Server Responder
```

} Missing in buggy run!

Figure 4.2 Logs generated by Hadoop-11252 bug. The logs are produced at the NameNode.

4.1.1 A Motivating Example

We use Hadoop-11252 [Hadb] bug to illustrate how a performance bug happens and how it manifests in different machine data types. The root cause of Hadoop-11252 bug is that the DataNode does not properly timeout the connection with the NameNode. Timeout is a commonly used failover mechanism to close the broken connection. As shown in Figure 4.1, the root cause function is `waitForProtocolProxy` function which passes 0 timeout value (0 means never timeout) to the timeout configuration incorrectly at the DataNode side. When the NameNode experiences some unexpected problems such as network outage, the DataNode hangs on waiting for the response from the remote server without producing any error information. As shown in Figure 4.2, we observe that the log entries that are typically produced by server stopping are missing at the NameNode side. Even if developers can discover the missing log anomaly at the NameNode side, it is still difficult for them to pinpoint the root cause function which is actually located at the DataNode side.

4.1.2 Contribution

In this chapter, we present PerfSig, an automatic performance bug signature extraction tool which performs multi-modality analysis across different machine data including system metrics, system logs, and function call traces. When a performance alert or service level objective (SLO) violation is detected, PerfSig is triggered to analyze a time window of recent machine data. PerfSig first employs signal processing techniques and unsupervised machine learning methods to identify fine-grained anomaly patterns in various machine data. For example, for system metrics such as CPU usage time series, we employ fast Fourier transform (FFT) and time series discord mining to identify anomaly patterns such as fluctuation pattern changes, persistent increase, and cycle period changes. For system logs, we identify abnormal log sequences such as missing log entries in a certain common sequence or overly long time span for certain sequences. Next, PerfSig performs causal analysis between abnormal metric/log patterns and function call traces using information theory method mutual information (MI) [Liz14]. The goal is to identify the root cause function which is the top contributor to the metric or log anomaly. PerfSig outputs the performance bug signature as the combination of the detected anomaly pattern and the pinpointed root cause function.

Specifically, this chapter makes the following contributions.

- We present a new multi-modality performance bug signature extraction framework which

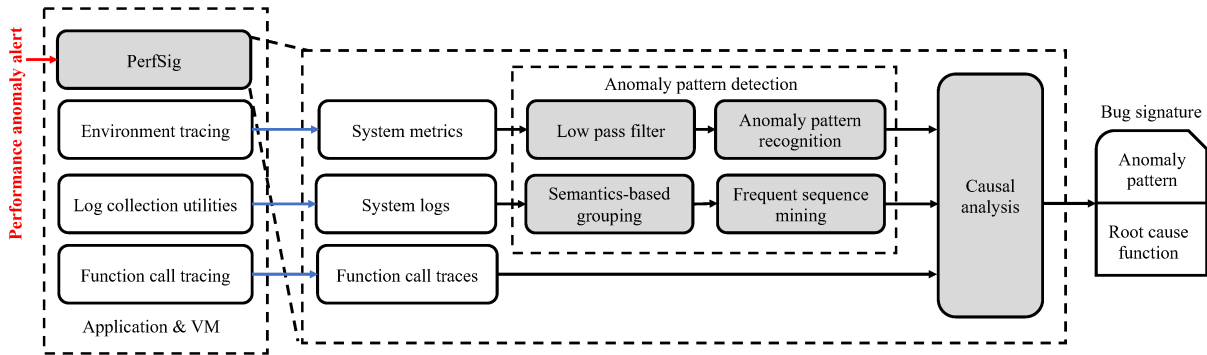


Figure 4.3 The architecture overview of PerfSig.

can precisely depict a performance bug using both fine-grained anomaly patterns and root cause functions.

- We describe a set of fine-grained anomaly detection methods to capture specific manifestation of a performance bug in system metrics or logs.
- We introduce an information theory based causal analysis approach to pinpointing root cause functions by discovering the causal relationship between function call traces and anomaly patterns of system metrics or logs.
- We have implemented a prototype of PerfSig and evaluated it over 20 real-world bugs that are discovered in six commonly used cloud systems. The results show that PerfSig can produce precise signatures for 19 out of 20 performance bugs.

The rest of the chapter is organized as follows. Section 4.2 discusses the system design details. Section 4.3 presents the experimental evaluation. Section 4.4 concludes the chapter.

4.2 System Design

In this section, we describe the system design of the PerfSig system in details. We first give an overview of the system. Next, we present how we identify various fine-grained anomaly patterns in system metric and system log data, respectively. Finally, we discuss how we perform causal analysis between system metric/log anomalies and function call traces to identify the root cause function which contributes to the anomaly.

4.2.1 Approach Overview

PerfSig adopts a two-phase approach to extracting signatures for a performance bug, shown by Figure 4.3. When a performance alert is generated or a service level objective (SLO) violation is detected, PerfSig is triggered to extract performance bug signatures on-the-fly by analyzing a recent time window of system metrics, system logs, and function call traces. During the first phase (Phase

I), PerfSig employs various signal processing and machine learning techniques to extract fine-grained anomaly patterns to capture the manifestation of the performance bug in either system metrics or system logs. Specifically, for system metrics, PerfSig uses time series analysis schemes to extract principal features such as fluctuation pattern changes, persistent increases, and cycle period changes (Section 4.2.2). For system logs, we leverage classification and frequent sequence mining to extract anomalous log patterns such as missing certain log entries or overly long time span of certain log sequences (Section 4.2.3). During the second phase (Phase II), PerfSig uses information theory approach to performing causal analysis between function call traces and detected abnormal system metric or system log patterns to pinpoint root cause functions (Section 4.2.4). Combining the phase I and Phase II results, PerfSig outputs the performance bug signature as the combination of the fine-grained anomaly pattern and the pinpointed root cause function.

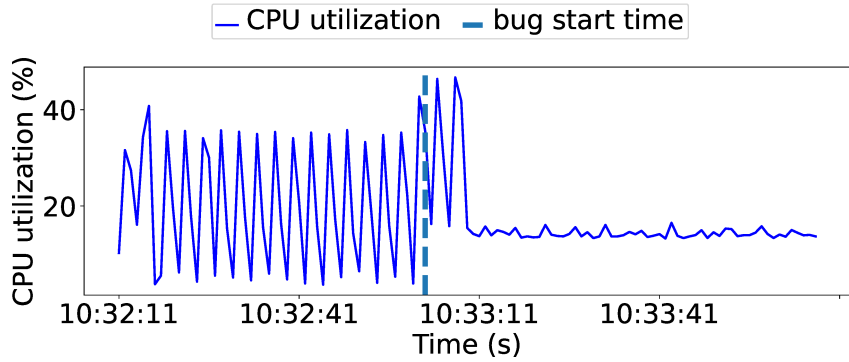
4.2.2 System Metric Anomaly Pattern Detection

Many performance bugs can manifest as changes in system metrics such as CPU utilization, memory utilization, and network traffic. However, different performance bugs can exhibit different anomaly patterns. For example, Figure 4.4 shows different CPU usage abnormal patterns for three real performance bugs. To extract distinctive signatures for different performance bugs, we need to not only detect anomalies but also extract fine-grained anomaly patterns.

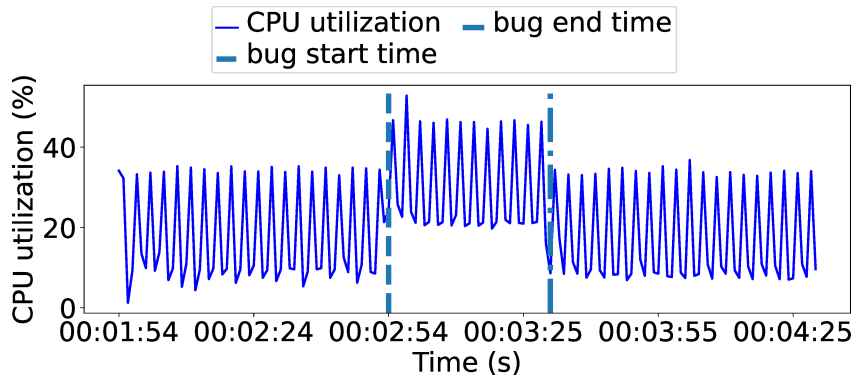
We observe that the system metrics such as CPU consumption are inherently fluctuating. In order to extract principal anomaly patterns, we first leverage low pass filters to remove random noises from the raw system metric time series. The low pass filter performs data denoising by filtering out high frequency signals in original system metric time series. The rationale is that random fluctuations usually manifest as the high frequency signals. We transform the time series to the signals in frequency domain and drop high frequency signals. After that, we transform the signals in the frequency domain back to the time series. Next, we employ signal processing methods over denoised time series to extract principal anomaly patterns.

Fluctuation Pattern changes. For dynamic data-intensive computing systems such as Hadoop, CPU utilization usually has periodical large fluctuations during normal run. It is because the application workload contains different types of interleaving jobs. For example, Figure 4.4a shows the CPU utilization's fluctuation change when Hadoop-15415 bug occurs. During the normal run (the first half of the figure), we observe large fluctuations. After the bug is triggered (the second half of the figure), the system hangs inside an infinite loop which fully consumes one CPU core and then CPU utilization stays at a steady value. We observe that many hang bugs in dynamic data-intensive systems often manifest as fluctuation pattern changes. To capture this anomaly pattern, PerfSig calculates the standard deviations of a moving window in the system metric time series and identify the time when the moving window standard deviation experiences significant changes (e.g., dropping from a large value to a small value).

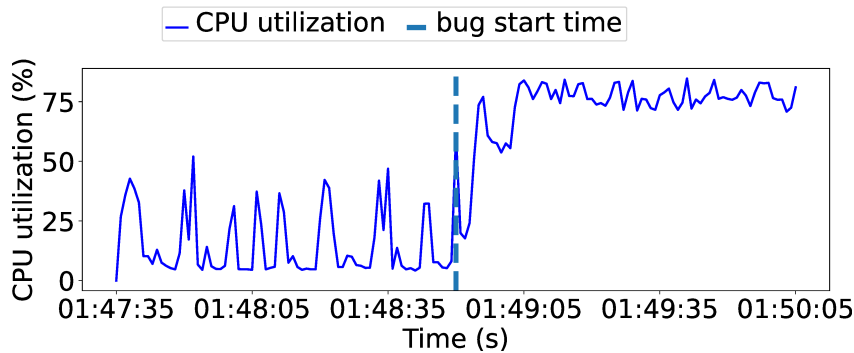
Persistent increases. Besides software hang bugs, slowdown bugs are another common category of performance bugs. We observe slowdown bugs caused by code inefficiency often consumes a



(a) Hadoop-15415 bug's fluctuation pattern change. The bug starts at 10:33:02.



(b) Hadoop-6133 bug's persistent increase. The bug starts at 00:02:54 and ends at 00:03:30.



(c) Cassandra-7330 bug's cyclic pattern change. The bug starts at 01:48:50.

Figure 4.4 Three commonly seen system metric anomaly patterns.

large amount of computational resources (e.g., CPU) during the abnormal period. For example, Figure 4.4b shows the CPU increase caused by the Hadoop-6133 bug. PerfSig detect such abnormal pattern using time series discord [Yeh16]. A sliding window is applied on the filtered system metric time series where the nearest neighbor distance [Yeh16] between the previous sliding windows and the current sliding window is computed. PerfSig detects the persistent increase pattern when the nearest neighbor distance shows significant increases.

Cyclic pattern changes. Many production server systems exhibit cyclic resource consumption patterns. We observe that when a performance bug is triggered, the cyclic pattern experiences changes. Figure 4.4c shows the CPU cyclic pattern change caused by the Cassandra-7330 bug. During normal run, CPU shows a cycle of nine seconds, while during buggy run, CPU does not show cyclic pattern. To detect such cyclic pattern changes, we employ fast Fourier transform (FFT) algorithm on a sliding window of system metric time series to extract the dominating frequencies whose magnitude values are in the top rank list. We detect the cyclic pattern change when the top frequency values experience changes.

4.2.3 System Log Anomaly Pattern Detection

We now describe how PerfSig extracts anomaly patterns from system logs. Much existing work focuses on detecting abnormal error logs that only appear in a buggy run. However, we observe that when a performance bug is triggered, the system usually does not produce any error log message. Instead, some log entries included in the normal run are missing during the buggy run, which are quite common among software hang bugs. In other cases like slowdown bugs, some log sequences could exhibit longer time span (i.e., the time duration from the start time of the first log entry in a sequence to the end time of the last log entry in a sequence) during the buggy run when compared with normal run.

Previous work in reconstructing execution path from the system logs contains three limitations: 1) focusing on sequential task execution [Du17]; 2) combined with domain knowledge extracted from binary code [Zha14b]; and 3) is not generic for all the systems [Yu16b]. Compared with the existing work, PerfSig considers concurrent task execution and only takes the log entries and vector timestamp as the input. PerfSig does not require any application-specific knowledge. To discover the log sequences from interleaving log entries, PerfSig first classifies the log entries generated by different tasks. After the task classification is done, we further separate the log entries based on the time gaps. Then we perform frequent sequence mining to extract the log sequences.

Semantics-based Grouping. After we collect logs from distributed hosts, PerfSig classifies logs generated by different tasks. The idea is that logs generated by the same task have similar semantic meanings. To achieve this goal, we use the word embedding vector [Mik13] to represent each word's contextual meaning. After that, we use the average of the word embedding vectors of the words' to represent each log entry, which is common for generate representation for natural language sentences [LM14]. Intuitively, log entries which have similar meanings have similar word embedding vectors. Therefore, we can apply clustering algorithm to grouping the similar log entries together. Figure 4.5 summarizes the classification procedures. Specifically, PerfSig pre-processes the logs to split the log entries into words, extracts word embedding vectors for each word, builds log entry representation by aggregating the word embeddings associated with each log entry, and classifies the entries generated by different tasks with the Self-Organizing Map (SOM) algorithm [Koh90].

First, we pre-process each log entry to extract the words for learning word embedding vectors. The first step is to split the log entries based on brackets and parentheses, because content inside a

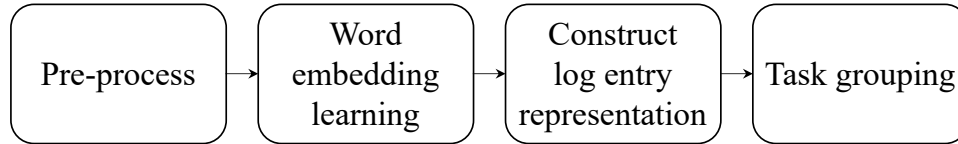


Figure 4.5 The log entry classification framework.

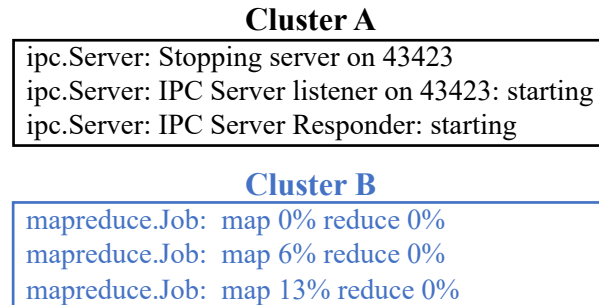


Figure 4.6 The log entries' embedding vectors forms two clusters, one for each task.

pair of brackets and parentheses often represents one command or operation, e.g., database query command. After that, we split the log entry based on comma, full stop, colon, and semi-colon. Then we further separate them according to the spaces between words.

Once we have extracted words from the log entries, we treat each entry as a sentence and learn the word embedding representation for each word. The major advantage of using word embedding is that it considers words' semantic meaning in the contexts. It is based on the hypothesis that words occurred in the similar contexts tend to be semantically similar. We choose to use word embedding as the feature vector because log entries generated by the same task typically use short sentence with similar terms. We use word2vec with Continuous Bag of Words (CBoW) [Mik13] to learn the word embedding vector to represent each word's meaning. Each word embedding is initialized as a n -dimensional vector. In each iteration, we train each word's embedding using the sum of m surrounding context words' embedding vectors. We update each word's embedding vector until the convergence is reached. In our experiment, n is set to 100 and m is set to five.

We construct each log entry's embedding vector by taking the average of all the words' embedding vectors in the log entry. After that, we apply the SOM clustering algorithm to clustering log entries that belong to different tasks. Compared with the traditional distance-based methods, SOM model has better performance to cluster high-dimensional vectors.

Figure 4.6 shows the an example of classification results. The log entries are split into two clusters and each cluster represents one type of task. For example, log cluster A represents Hadoop system establishes IPC connection between different processes. Log cluster B represents Hadoop system runs the map and reduce computational jobs.

Frequent Sequence Mining: After task classification, we successfully separate interleaving logs into log entry clusters, representing different tasks. The goal is this step is to extract frequent system log sequences with each log cluster, which often represent a set of execution such as client-server

connection or threads communication. Performance bugs manifest as missing log entries in the log sequence or the log sequence has abnormal time span. For example, Figure 4.2 shows a complete log sequence. When Hadoop-11252 bug happens, the last three log entries are missing. Our idea is to only extract the constants (log keys) from the log entries. Then we split the log entries based on the time gaps and extract frequently occurred log sequences.

The log entries contain variables like socket reader number and port number, which are different in each IPC connection. We extract the constants (log keys) by adopting simple regular expressions. After we split the log entries into words, we keep the words that only contain alphabetic letters and replace other words with “*”. Then we concatenate the words with a space as the log key. For each log entry cluster, we separate it based on the time gaps. The rationale is based on the observation that the same log sequence occurs after long time gap compared with its time span. We calculate the time gaps between each two consecutive log entries and derive the average value and standard deviation of each cluster. If the time gap between two consecutive log entries exceeds the average value + $2.0 \times$ standard deviation, we separate the log cluster.

After we get separated log clusters and replace each log entry with the log key, we perform frequent sequence mining to extract log sequences from normal run data. We use PrefixSpan [Han01] to perform frequent sequence mining because it is more efficient compared with other methods. After we extract top frequent log sequences from normal run, we use them to detect anomalies during buggy run. Specifically, we detect two anomaly patterns, i.e., missing logs and abnormal log sequence time span.

Missing log entries anomaly pattern: During the anomaly detection phase, we first perform semantics-based grouping. In each cluster, we extract log keys and check whether the log keys belong to any extracted log sequence. If the answer is yes, we check whether other log keys of the log sequence also appear in the log cluster. For example, Figure 4.2 shows how we detect Hadoop-11252 bug. During normal run, we extract the complete log sequence from starting the socket reader to stopping the server responder. During buggy run, we perform semantic-based grouping to cluster log entries generated from IPC connection tasks together. For the extracted log sequence, we find the log keys to start the server but we cannot find the log keys to stop the server.

Excessive time span anomaly pattern: Similar to missing log pattern detection, we perform semantics-based grouping and log key extraction. If all the log keys of one particular log sequence appear in one cluster, PerSig extracts the log sequence time span which starts from the first log key’s occurring time to the last log key’s occurring time. If the time span is excessive long, PerSig reports the abnormal log sequence time span pattern. For example, Figure 4.7 shows how we detect HDFS-4301 bug. During normal run, we extract the log sequence and the log sequence spans six seconds. During buggy run, the log sequence spans 97 seconds and PerSig reports the abnormal log sequence time span pattern.

4.2.4 Root Cause Analysis via Multi-modality Causal Analysis

After we identify the anomaly patterns, we perform causal analysis to localize the root cause function.

Normal run time span: 6s

```
... 21:41:21 INFO mortbay.log: Started
HttpServer2$SelectChannelConnectorWithSafeStartup@localhost:...
... 21:41:27 INFO mortbay.log: Stopped
HttpServer2$SelectChannelConnectorWithSafeStartup@localhost:...
```

Buggy run time span: 97s

```
... 21:42:43 INFO mortbay.log: Started
HttpServer2$SelectChannelConnectorWithSafeStartup@localhost:...
... 21:44:20 INFO mortbay.log: Stopped
HttpServer2$SelectChannelConnectorWithSafeStartup@localhost:...
```

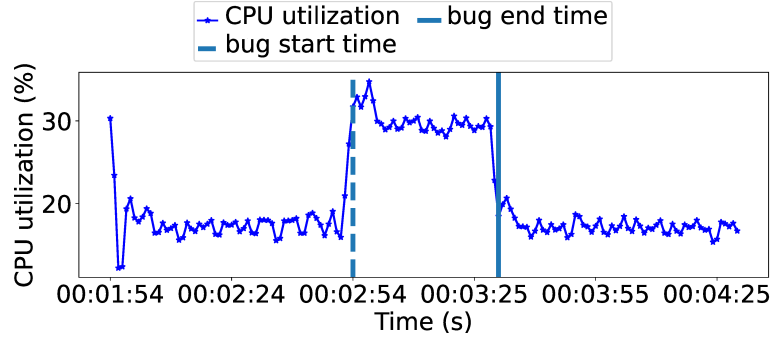
Figure 4.7 Logs generated by HDFS-4301 bug.

```
{ "i": "2960606995495142", "s": "ab3c06709e3bbab4",
  "b": 1622001774031, "e": 1622001774058,
  "d": "DFSInputStream#byteArrayRead",
  "r": "YarnChild", "p": [],
  "n": { "path": ".../job_1621397508582_0005/job.split" } }
...
{ "i": "37dada28edec1ea0", "s": "5cbe565f9c5a3c8b",
  "b": 1622001775036, "e": 1622001810393,
  "d": "conf.getClassByName", "r": "Test", "p": [] }
```

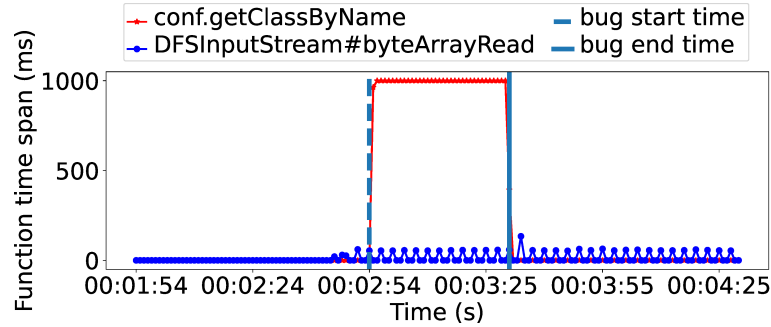
Figure 4.8 The extracted function call traces. “r” represents the function name, “b” represents function’s start timestamp and “e” represents function’s end timestamp. All the traces have the start time no earlier than 00:02:54 and end time no later than 00:03:30.

4.2.4.1 Causal Analysis between System Metrics and Function Call Traces

After we identify three anomaly patterns of the system metric time series, we retrieve a window of the filtered time series which contains the anomaly patterns. Suppose the bug happens between time $[t_1, t_2]$, then we retrieve the system metric between $[t_1 - w, t_2 + w]$, where w is the parameter to control the window size. Then we retrieve all the function call traces which occurs in the time window, i.e., $[t_1 - w, t_2 + w]$. For example, Figure 4.9a shows the CPU utilization time series. It contains the CPU spike occurred between 00:02:54 and 00:03:30. We set w to one minute, therefore, we include the normal run data from 00:01:54 to 00:02:54 and data from 00:03:30 to 00:04:30, because we want to include the sharp CPU changes at 00:02:54 and 00:03:30. Including normal run data can increase the accuracy of causal analysis. If we only consider the buggy run data, the frequently used functions also occur in the buggy run and may infer a high causality score. However, if we consider both normal run and buggy run, the frequent used functions always appear, which decreases the causal relationship between the system metrics and them. Figure 4.8 shows the retrieved function call trace snippet. All the function call traces are in the chronological order. We extract all the function call traces occurred between 00:02:54 and 00:03:30, i.e., their start times are no earlier than 00:02:54 and end time no later than 00:03:30. We extract three kinds of information, i.e, beginning timestamp,



(a) The CPU utilization time series after filtering. The bug starts at 00:02:54 and ends at 00:03:30. We retrieve one minute normal run data before and after bug is triggered.



(b) The function call time series in Hadoop-6133 bug. `conf.getClassByName` is the root cause function and `DFSInputStream#byteArrayRead` is a non-root cause function.

Figure 4.9 Hadoop-6133 bug’s time series.

ending timestamp and function name, to formulate the function call trace into time series.

Next, we formulate the function call traces into the time series. we extract the aggregated time span of each function between two consecutive time points of system metric time series. If CPU utilization are sampled at t_1, t_2, \dots, t_N , then the function time series can be formulated as $T[t_1, t_2), T[t_2, t_3), \dots, T[t_N, t_{N+1})$, where $T[t_i, t_{i+1})$ represents the function’s aggregated time span during $[t_i, t_{i+1})$ period. For example, Figure 4.9b shows the `conf.getClassByName` function time series. During the time $[00:02:25, 00:02:26)$, the bug happens and `conf.getClassByName` is invoked for all the one second period. Therefore, `conf.getClassByName` time series has the value of 1000 milliseconds at the time point 00:02:25. If M functions are invoked during the whole time window, then there are M function time series. In this way, the function time series is aligned with system metric time series, and we can apply causal analysis algorithm on them. We extract the time span as the function call’s feature because performance issues such as hang or slowdown, usually manifest as the changes in function time spans.

After we get all the function time series, we normalize each function time series and the system metric time series. After normalization, the sum of all the data points in one time series is equal to one. We adopt information theoretic method, i.e., mutual information (MI) [McC16; Liz14; Jid], to infer the causal relationship between each function’s time series and CPU time series. Mutual information measures how much knowing one of the two time series reduces uncertainty about the

other. Mutual information not only consider the absolute value of each data point, but also consider the data distribution across the whole time series. Therefore, we can filter out the frequently invoked functions. It is because the frequently invoked functions have long time span during both buggy run and normal run. Considering the data distribution, they cannot have a larger mutual information than those functions which are only frequently invoked during buggy run. In comparison, the frequent invoked functions can have high correlation scores with system metric time series because they have long time spans during buggy run.

Mutual information between two time series X and Y is defined as:

$$MI(X, Y) = \sum_{x,y} p(x, y) \log\left(\frac{p(x, y)}{p(x)p(y)}\right) \quad (4.1)$$

where $p(x)$ and $p(y)$ represents the probabilities of X and Y occurred at the sampling point. $p(x, y)$ represents the probability that X and Y occur at the same time.

Compared with co-variance based correlation methods such as Pearson and Spearman coefficient, mutual information can capture non-linear correlations between two time series. We calculate the mutual information between each function time series and the system metric time series. We then determine the function that contributes to the anomaly most as the one with the largest mutual information.

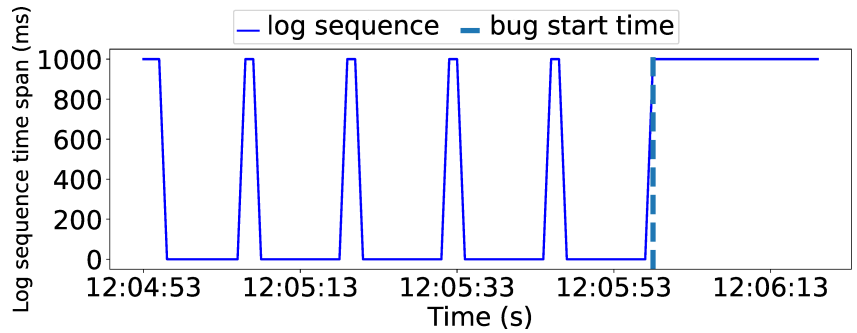
4.2.4.2 Causal Analysis between System Logs and Function Call Traces

We perform causal analysis on function time series and log sequence time series to localize the root cause function that contributes to the log anomaly. Even though system logs contain rich information, e.g., the class name that generates the logs, it is still essential to perform causal analysis, because the function that generates the log sequence is usually not the root cause function.

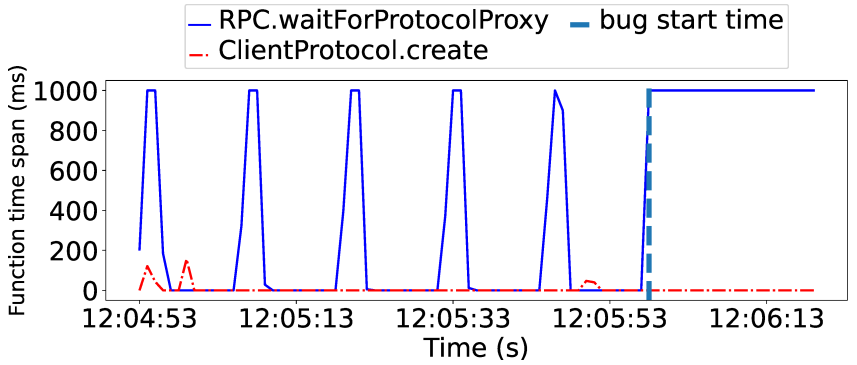
After we identify the missing log or longer execution patterns, we retrieve the log sequence's information, i.e., beginning time and end time, during both normal run and buggy run. The start time is the first log entry's invoking time and the end time is the last log entry's invoking time. For example, in Figure 4.2, the log sequence start time is 12:04:53 and end time is 12:04:56. If missing log pattern happens, we regard its time span as infinity because the log sequence never ends. Besides that, we retrieve all the function call trace within the same time window, i.e, during the period that the log sequences are produced.

We formulate the log sequences and the function call traces into time series. We sample them every one second and collect the aggregated time span during the one second interval. It is similar to how we formulate the function time series in Section 4.2.4.1. Note that the function time series still needs to be aligned with the log sequence time series. Figure 4.10a shows the log sequence time series and Figure 4.10b shows the function call time series. The root cause function has similar shape with the log sequence.

After generating log sequence and function time series, we calculate the mutual information between each function's time series and the log sequence to identify the root cause function, similar



(a) The log sequence time series. The bug is triggered at 12:05:58.



(b) The function call time series. `RPC.waitForProtocolProxy` is the root cause function and `ClientProtocol.create` is a non-root cause function.

Figure 4.10 Hadoop-11252 bug's time series.

to Section 4.2.4.1.

4.3 Experimental Evaluation

In this section, we first present the evaluation methodology followed by the experimental results. Next, we present several real bug examples including one negative case study where PerfSig fails to extract a signature for the bug.

4.3.1 Evaluation Methodology

Table 4.1 Bug benchmark. bug ID* represents it is a distributed system performance bug.

Bug ID	Version	Description
Cassandra-7330	2.0.8	The corrupted InputStream returns error code, causing an infinite loop. (Hang)
Cassandra-9881	2.0.8	Improper exception handling skips loop index-forwarding API, causing an infinite loop. (Hang)
Hadoop-11252*	2.5.0	the RPC connection timeout is missing, leading to system hanging. (Hang)
Hadoop-15415	2.5.0	Misconfigured parameter indirectly affects loop index, causing an infinite loop. (Hang)
Hadoop-5318	0.19.0	The AtomicLong operations cause contention with multiple threads. (Slowdown)
Hadoop-6133	0.20.0	Extra calls cause 80x performance slowdown. (Slowdown)
Hadoop-8614	0.23.0	Skipping after EOF returns error code, affecting loop stride. (Hang)
Hadoop-9106*	3.0.0-alpha1	IPC connection timeout is hard-coded, causing a much longer failure recovery time. (Slowdown)
MapReduce-5066*	2.0.3-alpha	Timeout is missing when JobTracker calls a URL, causing system hanging. (Hang)
MapReduce-4089	2.0.0-alpha	Task status updates cause hung task never timeout. (Hang)
MapReduce-3862	0.23.1	NodeManager hangs on shutdown due to struggling DeletionService threads. (Hang)
MapReduce-7089	2.5.0	Misconfigured variable causes loop stride to be set to 0. (Hang)
MapReduce-6990	2.5.0	Skipping on a corrupted InputStream returns error code, affecting loop stride. (Hang)
HDFS-1490*	2.0.2-alpha	Timeout is missing for image transfer operations, causing system hanging. (Hang)
HDFS-4301*	2.0.3-alpha	Timeout value on image transfer operation is large. (Slowdown)
HDFS-7005*	2.5.0	DFS input streams do not timeout. (Hang)
HBase-17341	1.3.0	Timeout is missing for terminating replication endpoint. (Slowdown)
Yarn-163	2.0.0-alpha	Skipping on a corrupted FileReader returns error code, affecting loop stride. (Hang)
Yarn-1630	2.2.0	YarnClient endlessly polls the state of an asynchronized application. (Hang)
Yarn-2905	2.5.0	Skipping on a corrupted aggregated log file returns error code, affecting loop stride. (Hang)

Cloud systems: We studied 20 real performance bugs from six commonly used open-source cloud systems: Hadoop common library, Hadoop MapReduce big data processing framework, Hadoop HDFS file system, Hadoop Yarn resource management service, HBase database system, and Cassandra database system. Four Hadoop systems are set up in distributed modes to evaluate PerfSig’s effectiveness over distributed system performance bugs.

Benchmarks: We use the “hang”, “stuck”, “block”, “log”, “CPU”, “performance” and “slowdown” keywords to search for performance bugs. We manually examine each bug to determine whether it is a real performance bug triggered in production environments and whether it is reproducible in deterministic ways. To the best of our efforts, we successfully reproduced 20 bugs in six cloud systems. Table 4.1 shows our collected bug benchmark.

Setup: All the experiments were conducted in our lab machine with an Intel i7-4790 Octa-core 3.6GHz CPU, 16GB memory, running 64-bit Ubuntu v16.04 with kernel v4.4.0.

4.3.2 Implementation

Function call tracing: The function call traces are collected using Google Dapper framework. Dapper has various implementations on different production systems. For example, an implementation of Dapper, HTrace [Htr] is integrated into Hadoop since version 2.7.0. Another implementation of Dapper, Zipkin [Zip] is integrated into Hadoop, HBase, and Cassandra. Those implementations collect traces for error-prone functions in cloud systems. We can configure the parameters for Dapper tracing in the configuration files directly and deploy the production systems to trace the function calls.

Anomaly pattern analysis: We implement the anomaly pattern detection in Python 3.9. We use scikit-learn [Ped11] package to implement FFT analysis and log classification. We use the Word2vec-CBoW implemented in Gensim [RS10] for embedding learning.

Hyperparameter in word embedding learning: We use the Word2vec-CBoW algorithm [Mik13] with the embedding vector size of 100 and the window size of 5 words. The SOM is set to be 5x5 grid map and the weight vector length in SOM is set to be 100, the same as the embedding vector size.

4.3.3 Alternative Approaches

4.3.3.1 Logs Anomaly Pattern Detection

For system log anomaly pattern detection, we implement four alternative approaches to compare with PerfSig.

DBScan-embedding: We use the same embedding representation. However, instead of the SOM clustering algorithm, we use DBScan [Sch17] to group the log entries generated by different tasks. DBScan is a popular non-parametric density-based clustering algorithm. It automatically determines the number of clusters when trained.

SOM-TFIDF: We use Term-Frequency-Inverse-Document-Frequency (TFIDF) [Jon72] to extract features from each log entry. Term frequency (TF) captures the word count frequency in a log entry.

Inverse document frequency (IDF) is used to weight down the frequently used meaningless words such as “the”, “an”, and “on”. The SOM algorithm is used to classify the TFIDF vectors associated with different log entries into different task groups.

DBScan-TFIDF: We use TFIDF to represent each log entry and DBScan [Sch17] to classify the log entries into different task groups.

Topic-LDA: Latent Dirichlet Allocation (LDA) [Ble03] is a popular technique to analyze the topics of natural language documents. When log entries are fed into the LDA algorithm, the algorithm estimates the probability of each log entry belonging to different topics. We choose the topic with the largest probability as its topic. Then we classify the log entries with the same topic into one task group.

4.3.3.2 Causal Analysis

We implement two alternative approaches for causal analysis for comparison with PerfSig. Both alternative approaches use correlation coefficients to predict causality. Specifically, we test the *Pearson Correlation Coefficient* [Guy08] and *Spearman's Rank Correlation Coefficient* [Guy08].

Pearson Correlation Coefficient: Given two time series x and y , Pearson coefficient is computed using Equation 4.2:

$$\frac{(x - \mu_x)(y - \mu_y)}{\sigma_x \sigma_y} \quad (4.2)$$

where μ_x is the mean of x , μ_y is the mean of y , σ_x is the standard deviation of x and σ_y is the standard deviation of y . In other words, it measures the correlation using normalised measurement of the co-variance.

Spearman's Rank Correlation Coefficient: Instead of computing the correlation using the raw values of x and y , both x and y are converted to rank values before computing the co-variance. For example, if the raw values for x is [0.3, 0.2, 0.5, 0.1], x is first converted to rank values [3, 2, 4, 1] before calculating the co-variance with the rank values of y . If the function for converting raw values to rank values is denoted as $\text{Rank}(\cdot)$, Spearman is computed with Equation 4.3:

$$\frac{(\text{Rank}(x) - \mu_{\text{Rank}(x)})(\text{Rank}(y) - \mu_{\text{Rank}(y)})}{\sigma_{\text{Rank}(x)} \sigma_{\text{Rank}(y)}} \quad (4.3)$$

We use the absolute value of the Spearman coefficient as the causal score. A larger causal score means the two time series are strongly correlated.

4.3.4 Results

Table 4.2 Signature extraction results for bugs which have system metric anomalies. Pearson^f is pearson method with filter. Similarly, Spearman^f is spearman method with filter. For each method, we present the root cause function's rank using causal analysis.

Bug ID	Anomaly Pattern	Root Cause Function	PerfSig	Pearson ^f	Spearman ^f	MI	Pearson	Spearman
Hadoop-8614	Fluctuation pattern change	IOUtils#skipFully	1	31	1	6	14	12
Hadoop-15415	Fluctuation pattern change	IOUtils #copyBytes	1	3	1	6	13	20
Yarn-163	Fluctuation pattern change	ContainerLogs Page#printLogs	1	24	2	8	20	81
Yarn-2905	Fluctuation pattern change	AggregatedLogs Block#read ContainerLogs	1	37	2	6	46	35
Yarn-1630	Fluctuation pattern change	YarnClientImpl #submit Application	1	2	1	12	10	21
MapReduce-7089	Fluctuation pattern change	ReadMapper #doIO	1	4	1	5	6	48
Mapreduce-6990	Fluctuation pattern change	TaskLog#Reader	1	41	1	2	58	20
Hadoop-5318	Persistent increase	FSDataOutput Stream#write	1	1	1	1	1	18
Hadoop-6133	Persistent increase	conf#get ClassByName	1	1	1	21	24	32
Cassandra-9881	Cyclic pattern change	Scrubber#scrub	1	1	1	1	1	1
Cassandra-7330	Cyclic pattern change	StreamReader #drain	1	1	1	1	1	1

Table 4.2 shows the results of signature extraction for bugs which manifest as system metric anomalies. PerfSig can identify three different anomaly patterns, i.e, fluctuation pattern changes, persistent increases and cyclic pattern changes, from all the 11 tested bugs. Moreover, PerfSig is capable of detecting the true root cause functions for all the tested bugs.

We also evaluated three different causal analysis methods (i.e., MI, Pearson and Spearman) under two different settings: with low-pass filter and without low-pass filter. Overall, the low-pass filter improves the quality of causal analysis result, no matter which causal analysis method is used. It is because smoothing the time series reduces irrelevant fluctuations brought by dynamic workloads and makes the anomaly pattern more salient. For example, in Hadoop-15415 bug, the `hflush` function ranks the highest when filtering is not employed. The `hflush` function is CPU-intensive, which is mis-detected as the root cause function due to a low causal score.

When comparing the three different causal analysis methods, we can see that the proposed MI scheme outperforms the alternative Pearson and Spearman methods. The experimental results show that good causal analysis techniques needs to consider data distribution across the whole time series. Moreover, entropy based method such as MI is better than co-variance based methods.

Table 4.3 Signature extraction results for bugs which have system log anomalies. X represents the anomaly pattern and the root cause function cannot be identified.

Bug ID	Signature (Anomaly Pattern, Root Cause Function)	PerfSig	DBScan -embedding	SOM -TFIDF	DBScan -TFIDF	Topic -LDA
Hadoop-11252	Infinite log sequence timespan due to missing closing server log, RPC.waitForProtocolProxy	1	1	1	1	X
MapReduce-5066	Infinite log sequence timespan due to missing closing server log, JobEndNotifier.localRunnerNotification	1	X	X	3	X
MapReduce-4089	Infinite log sequence timespan due to missing stopping service log, PingChecker.run	1	1	1	1	1
Mapreduce-3862	Infinite log sequence timespan due to missing stopping service log, DeletionService.delete	1	1	1	1	X
HDFS-7005	-	X	X	X	X	X
HDFS-1490	Infinite log sequence timespan due to missing closing server log, TransferFsImage.getFileClient	1	X	1	1	X
Hadoop-9106	Abnormal log sequence timespan, Client.call()	1	1	X	X	X
HBase-17341	Abnormal log sequence timespan, ReplicationSource.terminate()	1	1	X	X	X
HDFS-4301	Abnormal log sequence timespan, TransferFsImage.getFileClient()	1	X	1	1	X

Table 4.4 The runtime overhead and diagnosis time. Hadoop represents the four Hadoop system, i.e., Hadoop Common, Hadoop MapReduce, Hadoop HDFS and Hadoop Yarn.

System	Workload	Tracing Overhead	Metric Anomaly Detection Time	Log Anomaly Detection Time	Causal Analysis Time
Hadoop	π calculation	0.18%	0.13±0.01s	0.22±0.26s	0.28±0.16s
HBase	database query	0.67%	0.13±0.01s	0.17±0.01s	0.02±0.002s
Cassandra	database query	1.7%	0.13±0.01s	0.36±0.03s	0.15±0.01s

Table 4.3 shows the results of bug signature extraction results for bugs which manifest as log anomalies. Specifically, We compare PerfSig (i.e., SOM-embedding) with four other alternative designs: DBScan-embedding, SOM-TFIDE, DBScan-TFIDE, and Topic-LDA. Overall, PerfSig is capable of detecting the anomaly pattern for eight out of nine bugs which show log anomalies.

First, we compare PerfSig with DBScan-embedding where the workload classification method, SOM, is replaced with DBScan. PerfSig outperforms DBScan-embedding because the word embedding vector is 100-dimensional. DBScan has poor performance on high-dimensional vectors due to the curse of dimensionality [Kop00].

If we replace the embedding representation with the TFIDF representation, we can see PerfSig outperforms both alternative approaches with TFIDE, i.e., SOM-TFIDE and DBScan-TFIDE. It is because TFIDF only considers individual word occurrence and fails to capture the semantics of each word. The embedding representation conversely captures the semantic information by modeling the contextual surrounding words. As system logs are written in a human readable format for developers to diagnose problems of the system, the advantage of embedding representation over TFIDF is clear and also is observed from the experimental results. SOM and DBScan clustering have similar performance on TFIDF representation, because number of TFIDF dimensions is not large. Log entries typically are short sentences with limited key words.

Next, we replace the classification framework with topic extraction model (i.e., Topic-LDA), which is another kind of semantic analysis. We observe a much worse performance compared with the other methods. Topic-LDA has poor performance because the log entries are usually very short. According to a study on text documents from microblogging platform Twitter [Twi; HD10], the performance of Topic-LDA degrades when the input text documents are short. According to [HD10], it is hard for Topic-LDA to extract semantics from short documents as Topic-LDA cannot obtain sufficient statistics from short documents.

```

//ReflectionUtils class
104 public static <T> T newInstance(...) {
117     setConf(result, conf);
119 }

59 public static void setConf(...) {
64     setJobConf(theObject, conf);
66 }

74 private static void setJobConf(...) {
80     Class<?> jobConfClass =
81     conf.getClassByName(
        "org.apache.hadoop.mapred.JobConf");
95 }

//Configuration class
761 public Class <?> getClassByName(...)
        throws ClassNotFoundException {
762     return Class.forName(name, true, classLoader);
        /* duplicated costly operations to search for
        the same configuration class */
763 }

```

Figure 4.11 The code snippet of the Hadoop-6133 Bug.

4.3.5 Overhead and Diagnosis Time

Table 4.4 shows the runtime overhead and diagnosis time. The runtime tracing overhead is below 2%. The diagnosis time are all less than one second. Note that although log anomaly detection uses the deep learning model, i.e., word embedding model, the diagnosis time is still very short because log entries are typically short sentences with repeated words.

4.3.6 Case Studies

PerfSig extracts the log sequence time series during normal run and regard the log sequence's time span as infinity due to missing log during buggy run. PerfSig performs causal analysis on all the function call time series and the log sequence time series. The time series are shown in Figure 4.10. PerfSig determines the root cause function as `RPC.waitForProtoProxy` because it has the largest MI value.

Hadoop-6133 bug is caused by duplicated costly operations. As shown in Figure 4.11, when we use `ReflectionUtils.newInstance` function to initialize a new reflection instance at line 104, `setConf` function is invoked at line 117 to set the job configuration. `setConf` calls `setJobConf` function at line 64, then calls the `conf.getClassByName` function at line 81 to find a particular configuration class. However, the JDK function `Class.forName` invoked by `conf.getClassByName` is costly. When there are multiple threads which initialize the instances, `Class.forName` is frequently invoked to search for the same configuration class, which is unnecessary. When the bug is triggered, we observe 80x slowdown in the system. PerfSig identifies the bug anomaly pattern as the persistent increases, because `Class.forName` consumes a lot of CPU resources, as shown in Figure 4.9a. PerfSig then performs causal analysis on the CPU utilization time series and all the function time series.

```

... hdfs.MinidFSCluster: starting cluster: numNameNodes=1, numDataNodes=1
...
... hdfs.MinidFSCluster: dnInfo.length != numDataNodes → (Cluster A)
...
... hdfs.MinidFSCluster: Cluster is active → (Cluster B)
... hdfs.MinidFSCluster: Shutting down the Mini HDFS Cluster
... hdfs.MinidFSCluster: Shutting down DataNode 0

```

} Missing in buggy run!

Figure 4.12 Logs generated by HDFS-7005 bug.

As shown in Figure 4.9b, PerfSig determines the root cause function as `conf.getClassByName` because it has the largest MI value.

Negative Case Study: HDFS-7005 bug is caused by missing timeout for `DFSClient#newConnectedPeer`. When this bug happens, the Resource Manager hangs on waiting response from the HDFS cluster. We observe that the logs of shutting down HDFS cluster is missing as shown in Figure 4.12. PerfSig fails to classify the log entries to the same cluster. For example, the first two log entries are grouped in cluster A which represents DataNode’s tasks. Since the log entries of the log sequence are classified to different tasks, we cannot extract the right log sequence from the log clusters.

4.4 Summary

In this chapter, we present PerfSig, an automatic performance bug signature extraction tool. PerfSig can analyze various kinds of machine data including system metric, system logs, and function call traces to identify principal anomaly patterns and root cause functions as unique signature patterns for representing performance bugs. We have implemented a prototype of PerfSig and conducted extensive evaluations using 20 real world performance bugs on six commonly used cloud systems. Our results show that PerfSig can successfully extract unique signatures for 19 out of 20 tested performance bugs. PerfSig imposes low overhead to the cloud system, which makes it practical for production environments.

CHAPTER

5

RELATED WORK

5.1 Performance Bug Detection and Diagnosis

Performance bug is notoriously difficult to detect and diagnose. Previous work has developed both static and dynamic analysis tools to address the challenge. For example, Fournier et al. [FD10] proposed to analyze dependencies among processes to understand how the total elapsed time is distributed among different processes. X-ray [Att12] presented performance summarization techniques to attribute performance costs to fine-grained events for diagnosing performance problems. PerfScope [Dea14] used system call tracing and frequent episode mining to localize root cause functions for performance anomalies. PerfCompass [Dea16] presented a tool to differentiate external faults from internal faults based on the impact factor analysis over detected performance anomaly faults. BLeak [VB18] automatically debugged the memory leak problem which increases garbage collection frequency and overhead, further degrading responsiveness. Different from those generic performance bug detection and diagnosis tools, our work (TScope and TFix) focuses on identifying the root causes of timeout bugs.

Previous work has been done to detect software hang bugs. Hang doctor [BW18] detected soft hangs at runtime to address the limitations of offline detection. PerfChecker [Liu14] and HangWiz [Wan08] automatically detected soft hang bugs by searching the application code for known blocking APIs. Cotroneo et al. [Cot09] proposed to detect hangs in software by monitoring the response time of user actions. DScope [Dai18a] focused on detecting data corruption induced software hang problems.

Work has also been done to detect hang bugs caused by inefficient loops. Jolt [Car11] dynamically detected infinite loops by checking each loop iteration's run-time state. Carburizer [Kad09] statically

analyzed device driver code and identifies infinite driver-polling problems.

In comparison to previous hang bug detection schemes, our work (HangFix) focuses on auto-fixing a detected hang bug that is triggered in production cloud environments. HangFix can leverage those hang bug detection schemes for both patch generation triggering and patch validation.

5.2 Static Bug Detection and Diagnosis

Existing work has been done in developing static bug detection tools. Jin et al. [Jin12a] conducted a comprehensive study on performance bugs and propose efficiency rules to detect unknown bugs statically. DCatch [Liu17] designed a set of happen-before rules to model concurrency mechanisms in distributed cloud systems. CLARITY [Oli15] applied static analysis to identify redundant traversal bugs, which cause serious performance problems, e.g., system hanging. Other tools focused on detecting database’s performance anomalies [Che14], sequential errors [Zha11] and inefficient nested loops [Nis13]. Compared with the existing work, our work combines static analysis and dynamic analysis to achieve a higher accuracy and low overhead.

5.3 Dynamic Bug Detection and Diagnosis

Previous work has been done extensively to detect and diagnose bugs using dynamic analysis. For example, X-ray [Att12] diagnosed performance bugs by tracing the inputs and outputs of different components using dynamic binary instrumentation and inferencing the traces. Chopstix [Bha08] collected low-level OS events including scheduling, CPU utilization, I/O operations, etc. online and reconstructed these events offline for troubleshooting standalone bugs. Fournier et al. [FD10] proposed to analyze dependencies among processes and how the total elapsed time is distributed using kernel-level tracing. REPT [Cui18] utilized hardware trace to reconstruct the program’s execution and employed record-and-replay techniques for debugging. Magpie [Bar04] instrumented middleware and packet transfer points to record fine-grained system events and correlated these events to capture the control-flow and resource consumption of each request for debugging. SafeTimer [MW18] checked whether a timeout bug is caused by network delay or packet delay at the OS level. Faddegon et al. [FC16] presented a computation tree generation method that requires only a simple tracing library, for debugging any Haskell programs. In comparison, our work leverages various kinds of data such as system log, system call traces and function call traces, which complements the existing work.

Existing work has leveraged machine learning techniques to detect performance problems. Cohen et al. [Coh04] presented a tool based on tree-augmented Bayesian networks to correlate system-level metrics with high-level performance states. Lee et al. [LF16] introduced a fuzzy-prediction based self-tuning approach to improve the Hadoop system performance. Votke et al. [Vot17] designed an analytical model to estimate performance under various interference conditions. EntomModel [Ste10] applied decision tree classification to depicting the workloads and management

policies under which potential performance anomalies are likely to. UBL [Dea12] leveraged Self-Organizing Maps to capture emergent system behaviors and performs anomaly prediction for cloud systems. FChain [Ngu13a] localized faulty components based on the abnormal change propagation patterns and inter-component dependency relationships. Compared with the existing work, our work leverages various unsupervised machine learning techniques to perform fine-grained anomaly detection.

5.4 Configuration Bug Detection and Diagnosis

Previous work has been done to study the configuration bugs. Yin et al. [Yin11] and Xu et al. [XZ15] gave comprehensive studies of configuration bugs and pointed out configuration errors are hard to detect and diagnose. SPEX [Xu13] studied configuration constraints and exposed potential configuration errors by injecting errors that violate the constraints. ConfValley [Hua15b] introduced a new language to define system validation rules and check configurations against those rules before the application is deployed in production. PCheck [Xu16] analyzed the application source code and automatically emulated the late execution that uses configuration values to detect latent configuration errors. CODE [Yua11] detected configuration bugs by identifying the abnormal program executions using invariant configuration access rules. ConfAid [AF10] adopted dynamic taint tracking method to instrument the binary application and analyze the information flow in order to pinpoint the root causes of configuration errors. ConfDiagnoser [ZE13] extracted the control flow of configuration options, instrumented the application code for profiling and analyzed the configuration deviation to detect the erroneous configuration option. EnCore [Zha14a] applied machine learning techniques to model the correlation between the configuration settings and the executing environment and correlations between configuration entries, in order to learn and detect configuration bugs. However, the existing approaches detect configuration bugs by checking whether the system violates predefined rules. They cannot be readily applied to fix performance bugs which are triggered during system runtime due to unexpected input data or computing environment conditions.

5.5 Automatic Bug Fix

Work has also been done for automatic bug fixes. AFix [Jin11] and CFix [Jin12b] proposed automatic patching strategies for concurrency bugs. ClearView [Per09] identified violated invariants from erroneous executions and generated candidate repair patches to change the invariants. Tian et al. [Tia12] presented an automatic bug fixing patch identification tool to maintain older stable versions. DFix [Li19] adopts the rollback and fast-forward strategy to fix distributed timing bugs.

Tufano et al. [Tuf18] applied an encoder/decoder model to mine the existing patches and automatically generate new patches. Genprog [Le 12] is a search-based genetic programming approach for automated program repairs. SemFix [Ngu13b] is a semantic-based program repair tool, which derives repair constraints from a set of tests and solves the repair constraints to generate a valid

repair.

Assure [Sid09] fixed runtime faults by restoring program execution to a rescue point where error-handling is performed to recover the program execution. Ares [Gu16] recovered the program from runtime unexpected errors with the program’s existing error-handlers. Ares synthesized a number of error-handlers and selects the most promising one via virtual testing techniques. Gulwani et al. [Gul18] proposed an automated program repair algorithm to use the existing correct student solutions to provide feedback for incorrect ones in programming education. Remix [Eiz16] leveraged Intel’s performance counter techniques to detect and repair false sharing bugs in multi-threaded programs. Huron [Kha19] presented a hybrid false sharing and repair framework with a low overhead.

The existing work focuses on functional bug fixing. However, our work focuses on fixing two types of performance bugs (timeout bugs and hang bugs).

5.6 Single-modality Data Analysis

Previous work has been done on bug diagnosis by performing single modality data analysis. Cohen et al. [Coh05; Coh04] leveraged Tree-Augmented Naive Bayes models and clustering methods to extract signatures from system metrics. PerfScope [Dea14] and PerfCompass [Dea16] diagnosed performance bugs by performing unsupervised machine learning on system call traces. Stitch [Zha16] and lprof [Zha14b] reconstructed the domain knowledge and system model from the logs. CloudSeer [Yu16b] reconstructed the execution workflow entirely from interleaving OpenStack logs. Deeplog [Du17] applied a deep neural network model, i.e., Long Short-Term Memory (LSTM), to detect anomalies from the system logs. The mystery machine [Cho14] analyzed logs from Internet service to diagnose Facebook request latency. CSight [Bes14] modeled the system behavior in the form of CFSM from system logs. Lou et al. [Lou10] constructed program workflow from event traces to understand system behaviors and verify system executions. PBI [Aru13] and REPT [Cui18] leveraged hardware traces, i.e., performance counters and Intel Processor Trace, to understand the software bugs. Compared with existing work, our work (PerfSig) performs multi-modality analysis to address the limitation that performance bugs manifest in different data types.

5.7 Causal Analysis

Causal analysis has been done on identifying Granger-causal relationship among different time series. McCracken [McC16] presented a comprehensive review to perform exploratory causal analysis. The causal analysis techniques could be divided into two categories, i.e., regression-based and information theory-based methods. For the regression-based method, Arnold et al. [Arn07] adopted linear lasso regression for identifying Granger-causal relationship among time series. Tank et al. [Tan21] explored the possibility of detecting non-linear Granger-causal relationship among time series by training deep learning models (i.e., multi-layer perceptrons and recurrent neural networks) with sparsity constrain. For the techniques using information theory, the Granger-causal

relationship was detected by entropy-based measures [AM11; Vic11]. Existing work leveraged different kinds of measures like directed information theory [AM11; AM13] or transfer entropy [Vic11]. Compared with existing work, our work (PerfSig) makes the first step to apply the information theory method mutual information to performance bug signature extraction.

CHAPTER

6

CONCLUSION AND FUTURE WORK

This dissertation focuses on developing a framework of automatically fixing performance bugs and extracting bug signatures. Specifically, we present three key techniques: a performance bug detection module, a performance bug fixing module and a performance bug signature extraction module.

The rest of this chapter is organized as follows. We first summarize the main contributions of the dissertation. We then discuss the future research directions.

6.1 Contributions

The contributions of this dissertation are in developing effective and practical bug detection techniques, bug fixing strategies, and bug signature extraction solutions. Specifically, we make the following contributions:

- We present TScope, an automatic runtime timeout bug identification system. TScope combines a unique system call selection scheme with unsupervised machine learning based anomaly detection to achieve higher detection accuracy and precise timeout bug identification. TScope is light-weight and does not require application instrumentation. We have implemented a prototype of TScope and conducted experiments on 19 real-world performance bugs, including 12 timeout bugs and 7 non-timeout performance bugs. The experimental results show that TScope correctly classifies 18 out of 19 bugs. Compared to existing runtime bug detection schemes, TScope reduces the average false positive rate from 47.24% to 0.8%.
- We present TFix, a tool that automatically fixes timeout bugs with proper configurations in

cloud systems. TFix adopts a *drill-down bug analysis protocol* that can narrow down the root cause of a timeout bug and produce configuration recommendations for correcting the root cause. TFix employs a system call frequent episode mining scheme to check whether a timeout bug is caused by a misused timeout variable. TFix combines dynamic application tracing and static taint analysis to pinpoint the misused timeout variable. TFix performs timeout variable value recommendation based on the profiled execution time of the pinpointed function during normal runs. We have implemented a prototype of TFix and conducted extensive evaluation using 13 real world timeout bugs. Our results show that TFix can correctly classify all tested misused timeout bugs and pinpoint the exact timeout variable that has caused the timeout bug. The timeout values suggested by TFix can effectively correct all the tested misused timeout bugs.

- We present HangFix, a domain-agnostic, byte-code-based software hang bug fixing tool. HangFix first leverages stack trace analysis to localize the hang function and then performs root cause pattern matching to classify hang bugs into different types based on likely root causes. Next, HangFix generates effective code patches based on the identified root cause patterns. Our empirical bug study shows that our likely root cause patterns cover 76% of 237 hang bugs. HangFix can correct the hang symptom for all of those matched bugs and completely fix the root cause for 75% of them. We have implemented a prototype of HangFix and evaluated our system using 42 reproduced real-world software hang bugs in 10 commonly used cloud systems. HangFix successfully fixes 40 of them in seconds.
- We present PerSig, a new multi-modality performance bug signature extraction framework. PerSig first employs signal processing techniques and unsupervised machine learning methods to identify fine-grained anomaly patterns in various machine data. Next, PerSig performs causal analysis between abnormal metric/log patterns and function call traces using information theory method mutual information (MI) [Liz14]. The goal is to identify the root cause function which is the top contributor to the metric or log anomaly. We have implemented a prototype of PerSig and evaluated it over 20 real-world bugs that are discovered in six commonly used cloud systems. The results show that PerSig can produce precise signatures for 19 out of 20 performance bugs.

6.2 Future Work

Cloud systems become increasingly complex and performance bugs are inevitable. We have presented a framework of performance bug detection, fixing, and signature extraction. However, there is still much work to be done.

- **Hot patching.** Our current design for TFix and HangFix requires the application system to be re-deployed and restarted. In comparison, hot patching techniques can patch the application system on-the-fly, which further reduces the system downtime and minimizes the financial

cost. We have explored to modify the system call execution by inserting delays to fix the timeout bugs caused by too small timeout values. However, this method only applies to a small category of performance bugs, and it cannot be generalized to fix all the performance bugs. One of the promising future direction is to insert kernel modules or modify the interpreter (e.g., Java Virtual Machine) to fix performance bugs on-the-fly.

- **Heterogeneous configuration for heterogeneous nodes.** Our current implementation for TFix provides configuration recommendation based on normal run tracing data to determine proper configurations under current workloads and system environment. The configuration recommendation applies to all the nodes in the system. However, modern large-scale cloud system composes of thousands of heterogeneous nodes. In this case, heterogeneous nodes may require heterogeneous settings due to different environments. Therefore, configuration recommendation scheme can be extended to recommend heterogeneous configurations to adapt to each node's environment, which however can introduce system inconsistency and code inefficiency. We will investigate how to address the problem of performance degradation caused by homogeneous configuration for heterogeneous nodes. Moreover, we will investigate how to improve our fixing system by recommending heterogeneous configurations for heterogeneous nodes.

Performance bugs are serious issues in the cloud today and the initial work done here by no means makes it a solved problem. A broader challenges researchers need to address is how to integrate the state-of-art techniques to provide an integrated framework to automatically fix performance bugs and extract bug signatures in large-scale production systems. Our work presented in the dissertation attempts to deliver an effective framework to fix performance bugs and extract bug signatures. However, there are still a lot of open research problems to be investigated in the future.

BIBLIOGRAPHY

- [AM11] Amblard, P.-O. & Michel, O. J. “On directed information theory and Granger causality graphs”. *Journal of computational neuroscience* **30.1** (2011), pp. 7–16.
- [AM13] Amblard, P.-O. & Michel, O. J. “The relation between Granger causality and directed information theory: A review”. *Entropy* **15.1** (2013), pp. 113–143.
- [Hba] *Apache HBase*. <http://hbase.apache.org>. 2014.
- [Jir] *Apache JIRA*. <https://issues.apache.org/jira>.
- [Arn07] Arnold, A. et al. “Temporal causal modeling with graphical granger methods”. *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007, pp. 66–75.
- [Aru13] Arulraj, J. et al. “Production-run software failure diagnosis via hardware performance counters”. *Acm Sigplan Notices* **48.4** (2013), pp. 101–112.
- [Att12] Attariyan, M. et al. “X-ray: Automating root-cause diagnosis of performance anomalies in production software”. *OSDI*. 2012.
- [AF10] Attariyan, M. & Flinn, J. “Automating Configuration Troubleshooting with Dynamic Information Flow Analysis.” *OSDI*. Vol. 10. 2010. 2010, pp. 1–14.
- [Awsa] *AWS outage*. <https://www.techrepublic.com/article/amazon-reveals-reason-for-last-weeks-major-aws-outage/>. 2020.
- [Awsb] *AWS outage knocks Amazon, Netflix, Tinder and IMDb in MEGA data collapse*. https://www.theregister.co.uk/2015/09/20/aws_database_outage/. 2015.
- [Bar04] Barham, P. et al. “Using Magpie for request extraction and workload modelling”. *OSDI*. 2004.
- [Bes14] Beschastnikh, I. et al. “Inferring models of concurrent systems from logs of their behavior with CSight”. *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 468–479.
- [Bha08] Bhatia, S. et al. “Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems.” *OSDI*. 2008, pp. 103–116.
- [Ble03] Blei, D. M. et al. “Latent dirichlet allocation”. *the Journal of machine Learning research* **3** (2003), pp. 993–1022.
- [BW18] Brocanelli, M. & Wang, X. “Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps”. *Proceedings of the Thirteenth EuroSys Conference*. EuroSys. 2018.
- [Bug] *Bugzilla*. <https://www.bugzilla.org>.

- [Car11] Carbin, M. et al. “Detecting and Escaping Infinite Loops with Jolt”. *Proceedings of the European Conference on Programming Languages*. ECOOP. 2011.
- [Cas] *Cassandra*. <https://cassandra.apache.org/>.
- [Che] *Checker Framework*. <https://checkerframework.org>.
- [Che14] Chen, T.-H. et al. “Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping”. *ICSE*. 2014.
- [Cho14] Chow, M. et al. “The mystery machine: End-to-end performance analysis of large-scale internet services”. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 217–231.
- [Coh04] Cohen, I. et al. “Correlating instrumentation data to system states: A building block for automated diagnosis and control”. *OSDI*. 2004.
- [Coh05] Cohen, I. et al. “Capturing, indexing, clustering, and retrieving system history”. *ACM SIGOPS Operating Systems Review* **39.5** (2005), pp. 105–118.
- [Cot09] Cotroneo, D. et al. “Assessment and Improvement of Hang Detection in the Linux Operating System”. *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*. SRDS. 2009.
- [Cui18] Cui, W. et al. “REPT: Reverse Debugging of Failures in Deployed Software”. *OSDI*. 2018, pp. 17–32.
- [Dai18a] Dai, T. et al. “DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems”. *Proceedings of the ACM Symposium on Cloud Computing*. SoCC. 2018.
- [Dai18b] Dai, T. et al. “Understanding Real World Timeout Problems in Cloud Server Systems”. *IC2E*. 2018.
- [Dea12] Dean, D. J. et al. “UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems”. *ICAC*. 2012.
- [Dea14] Dean, D. J. et al. “PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures”. *SOCC*. 2014.
- [Dea15a] Dean, D. J. et al. “Automatic Server Hang Bug Diagnosis: Feasible Reality or Pipe Dream?” *ICAC*. 2015.
- [Dea16] Dean, D. J. et al. “PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds”. *TPDS* **27.6** (2016), pp. 1742–1755.
- [Dea15b] Dean, D. et al. “Automatic Server Hang Bug Diagnosis: Feasible Reality or Pipe Dream?” *ICAC*. 2015.
- [DD06] Desnoyers, M. & Dagenais, M. R. “The lttng tracer: A low impact performance and behavior monitor for gnu/linux”. *Linux Symposium*. 2006.

- [Doc] *Docker*. <https://www.docker.com/>. 2021.
- [Du17] Du, M. et al. “Deeplog: Anomaly detection and diagnosis from system logs through deep learning”. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1285–1298.
- [Eiz16] Eizenberg, A. et al. “Remix: online detection and repair of cache contention for the JVM”. *ACM SIGPLAN Notices*. Vol. 51. 6. ACM. 2016, pp. 251–265.
- [FC16] Faddegon, M. & Chitil, O. “Lightweight computation tree tracing for lazy functional languages”. *ACM SIGPLAN Notices*. Vol. 51. 6. ACM. 2016, pp. 114–128.
- [Flu] *Flume*. <https://flume.apache.org>.
- [FD10] Fournier, P. & Dagenais, M. R. “Analyzing blocking to debug performance problems on multi-core systems”. *SIGOPS*. 2010.
- [Gu16] Gu, T. et al. “Automatic Runtime Recovery via Error Handler Synthesis”. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE. 2016.
- [Gul18] Gulwani, S. et al. “Automated clustering and program repair for introductory programming assignments”. *ACM SIGPLAN Notices*. Vol. 53. 4. ACM. 2018, pp. 465–480.
- [Gun14] Gunawi, H. S. et al. “What Bugs Live in the Cloud?: A Study of 3000+ Issues in Cloud Systems”. *SOCC*. 2014.
- [Guy08] Guyon, I. et al. “Practical feature selection: from correlation to causality”. *Mining massive data sets for security: advances in data mining, search, social networks and text mining, and their applications to security* (2008), pp. 27–43.
- [Hada] *Hadoop*. <https://hadoop.apache.org/>.
- [Hadb] *Hadoop-11252*. <https://issues.apache.org/jira/browse/HADOOP-11252>.
- [Han01] Han, J. et al. “Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth”. *proceedings of the 17th international conference on data engineering*. Citeseer. 2001, pp. 215–224.
- [Hdf] *HDFS*. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
- [He18] He, J. et al. “TScope: Automatic Timeout Bug Identification for Server Systems”. *ICAC*. 2018.
- [He19] He, J. et al. “TFix: Automatic Timeout Bug Fixing in Production Server Systems”. *ICDCS*. 2019.
- [HD10] Hong, L. & Davison, B. D. “Empirical study of topic modeling in twitter”. *Proceedings of the first workshop on social media analytics*. 2010, pp. 80–88.
- [Hpr] *HProf*. <https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html>.

- [Htr] *HTrace*. <http://htrace.incubator.apache.org/>.
- [Hua15a] Huang, J. et al. “Understanding issue correlations: a case study of the Hadoop system”. *SOCC*. 2015.
- [Hua15b] Huang, P. et al. “ConfValley: a systematic configuration validation framework for cloud services”. *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 19.
- [Dyn] *Irreversible Failures: Lessons from the DynamoDB Outage*. <http://blog.scalyr.com/2015/09/irreversible-failures-lessons-from-the-dynamodb-outage/>.
- [Jid] *Java Information Dynamics Toolkit (JIDT)*. <https://github.com/jlizier/jidt>.
- [Jin12a] Jin, G. et al. “Understanding and detecting real-world performance bugs”. *PLDI*. 2012.
- [Jin11] Jin, G. et al. “Automated atomicity-violation fixing”. *PLDI*. 2011.
- [Jin12b] Jin, G. et al. “Automated concurrency-bug fixing”. *OSDI*. 2012.
- [Jon72] Jones, K. S. “A statistical interpretation of term specificity and its application in retrieval”. *Journal of documentation* (1972).
- [Kad09] Kadav, A. et al. “Tolerating Hardware Device Failures in Software”. *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. SOSP. 2009.
- [KR09] Kaufman, L. & Rousseeuw, P. J. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons, 2009.
- [Kha19] Khan, T. A. et al. “Huron: hybrid false sharing detection and repair”. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2019, pp. 453–468.
- [Koh90] Kohonen, T. “The self-organizing map”. *Proceedings of the IEEE* **78.9** (1990), pp. 1464–1480.
- [Kop00] Koppen, M. “The curse of dimensionality”. *5th Online World Conference on Soft Computing in Industrial Applications (WSC5)*. Vol. 1. 2000, pp. 4–8.
- [Kpr] *KProbes*. <https://lwn.net/Articles/132196/>.
- [LM14] Le, Q. & Mikolov, T. “Distributed representations of sentences and documents”. *International conference on machine learning*. PMLR. 2014, pp. 1188–1196.
- [Le 12] Le Goues, C. et al. “GenProg: A Generic Method for Automatic Software Repair”. *IEEE Trans. Software Eng.* **38.1** (2012), pp. 54–72.
- [LF16] Lee, G. & Fortes, J. “Hadoop Performance Self-Tuning Using a Fuzzy-Prediction Approach”. *ICAC*. 2016.

- [Li19] Li, G. et al. “DFix: Automatically Fixing Timing Bugs in Distributed Systems”. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2019.
- [Sysa] *Linux system calls*. <http://man7.org/linux/man-pages/man2/syscalls.2.html>.
- [Liu17] Liu, H. et al. “DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems”. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2017, pp. 677–691.
- [Liu14] Liu, Y. et al. “Characterizing and Detecting Performance Bugs for Smartphone Applications”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE. 2014.
- [Liz14] Lizier, J. T. “JIDT: An information-theoretic toolkit for studying the dynamics of complex systems”. *Frontiers in Robotics and AI* **1** (2014), p. 11.
- [Lou10] Lou, J.-G. et al. “Mining program workflow from interleaved traces”. *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2010, pp. 613–622.
- [MW18] Ma, S. & Wang, Y. “Accurate timeout detection despite arbitrary processing delays”. *2018 USENIX Annual Technical Conference (ATC)*. 2018, pp. 467–480.
- [McC16] McCracken, J. M. “Exploratory Causal Analysis with Time Series Data”. *Synthesis Lectures on Data Mining and Knowledge Discovery* **8.1** (2016), pp. 1–147.
- [Mik13] Mikolov, T. et al. “Efficient estimation of word representations in vector space”. *arXiv preprint arXiv:1301.3781* (2013).
- [Moo01] Moore, R. J. “A Universal Dynamic Trace for Linux and Other Operating Systems.” *ATC*. 2001, pp. 297–308.
- [Ngu13a] Nguyen, H. et al. “FChain: Toward Black-box Online Fault Localization for Cloud Systems”. *ICDCS*. 2013.
- [Ngu13b] Nguyen, H. D. T. et al. “SemFix: Program Repair via Semantic Analysis”. *Proceedings of the 2013 International Conference on Software Engineering*. ICSE. 2013.
- [Nis13] Nistor, A. et al. “Toddler: Detecting Performance Problems via Similar Memory-access Patterns”. *ICSE*. 2013.
- [Nis15] Nistor, A. et al. “Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes”. *ICSE*. 2015.
- [Oli15] Olivo, O. et al. “Static detection of asymptotic performance bugs in collection traversals”. *ACM SIGPLAN Notices*. Vol. 50. 6. ACM. 2015, pp. 369–378.
- [Jst] *Oracle jstack*. <https://docs.oracle.com/en/java/javase/13/docs/specs/man/jstack.html>.

- [Ped11] Pedregosa, F. et al. “Scikit-learn: Machine learning in Python”. *the Journal of machine Learning research* **12** (2011), pp. 2825–2830.
- [Per09] Perkins, J. H. et al. “Automatically patching errors in deployed software”. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 87–102.
- [RS10] Rehurek, R. & Sojka, P. “Software framework for topic modelling with large corpora”. *In Proceedings of the LREC 2010 workshop on new challenges for NLP frameworks*. Citeseer. 2010.
- [Sch17] Schubert, E. et al. “DBSCAN revisited, revisited: why and how you should (still) use DBSCAN”. *ACM Transactions on Database Systems (TODS)* **42.3** (2017), pp. 1–21.
- [Sid09] Sidiroglou, S. et al. “ASSURE: Automatic Software Self-healing Using Rescue Points”. *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. 2009.
- [Sig10] Sigelman, B. H. et al. *Dapper, a large-scale distributed systems tracing infrastructure*. Tech. rep. Technical report, Google, Inc, 2010.
- [Soo] *Soot: A Framework for Analyzing and Transforming Java and Android Applications*. <https://sable.github.io/soot/>. 2019.
- [Spa] *Spark*. <https://spark.apache.org/>.
- [Ste10] Stewart, C. et al. “EntomoModel: Understanding and Avoiding Performance Anomaly Manifestations”. *MASCOTS*. 2010.
- [Sysb] *SystemTap*. <https://sourceware.org/systemtap/>.
- [Tan12] Tan, Y. et al. “PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems”. *ICDCS*. 2012.
- [Tan21] Tank, A. et al. “Neural Granger Causality”. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), 1–11.
- [Tia12] Tian, Y. et al. “Identifying linux bug fixing patches”. *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 386–396.
- [Tuf18] Tufano, M. et al. “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation”. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. 2018, pp. 832–837.
- [Twi] *Twitter*. <https://twitter.com>. 2021.
- [Vic11] Vicente, R. et al. “Transfer entropy—A model-free measure of effective connectivity for the neurosciences”. *Journal of computational neuroscience* **30.1** (2011), pp. 45–67.
- [VB18] Vilks, J. & Berger, E. D. “Bleak: automatically debugging memory leaks in web applications”. *ACM SIGPLAN Notices*. Vol. 53. 4. ACM. 2018, pp. 15–29.

- [Vot17] Votke, S. et al. “Modeling and Analysis of Performance under Interference in the Cloud”. *MASCOTS*. 2017.
- [Wan08] Wang, X. et al. “Hang Analysis: Fighting Responsiveness Bugs”. *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. EuroSys. 2008.
- [XZ15] Xu, T. & Zhou, Y. “Systems approaches to tackling configuration errors: A survey”. *ACM Computing Surveys (CSUR)* 47.4 (2015), p. 70.
- [Xu13] Xu, T. et al. “Do not blame users for misconfigurations”. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 244–259.
- [Xu16] Xu, T. et al. “Early Detection of Configuration Errors to Reduce Failure Damage”. *OSDI*. 2016.
- [Yeh16] Yeh, C.-C. M. et al. “Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets”. *2016 IEEE 16th international conference on data mining (ICDM)*. Ieee. 2016, pp. 1317–1322.
- [Yin11] Yin, Z. et al. “An empirical study on configuration errors in commercial and open source systems”. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 159–172.
- [Yu16a] Yu, X. et al. “CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs”. *ASPLOS*. 2016.
- [Yu16b] Yu, X. et al. “Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs”. *ACM SIGARCH Computer Architecture News* 44.2 (2016), pp. 489–502.
- [Yua11] Yuan, D. et al. “Context-based online configuration-error detection”. *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIX Association. 2011, pp. 28–28.
- [Zha14a] Zhang, J. et al. “Encore: Exploiting system environment and correlation information for misconfiguration detection”. *ACM SIGARCH Computer Architecture News* 42.1 (2014), pp. 687–700.
- [ZE13] Zhang, S. & Ernst, M. D. “Automated diagnosis of software configuration errors”. *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 312–321.
- [Zha11] Zhang, W. et al. “ConSeq: detecting concurrency bugs through sequential errors”. *ACM Sigplan Notices*. Vol. 46. 3. ACM. 2011, pp. 251–264.
- [Zha14b] Zhao, X. et al. “lprof: A non-intrusive request flow profiler for distributed systems”. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 629–644.

- [Zha16] Zhao, X. et al. “Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle”. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 603–618.
- [Zha17] Zhao, X. et al. “Log20: Fully automated optimal placement of log printing statements under specified overhead threshold”. *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 565–581.
- [Zip] *Zipkin*. <https://zipkin.io/>.