THE UNIVERSITY OF CHICAGO

MODELING AND TACKLING TIMING BUGS IN MULTI-THREADED SYSTEMS AND DISTRIBUTED SYSTEMS

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

DEPARTMENT OF COMPUTER SCIENCE

BY
HAOPENG LIU

CHICAGO, ILLINOIS
AUGUST 2019

Copyright © 2019 by Haopeng Liu

All Rights Reserved

Dedicated to my family, for their endless love.

"If debugging is the process of removing software bugs, then programming must be the process of putting them in." — Edsger W. Dijkstra

TABLE OF CONTENTS

LIST OF FIGURES viii

LIST OF TABLES ix

ACKNOWLEDGMENTS x

ABSTRACT xii

1 INTRODUCTION 1
1.1 Motivation 1
1.1.1 Timing Bugs 1
1.1.2 LCbugs and DCbugs are Difficult to Tackle 3
1.2 Approaches to Tackle Timing Bugs 4
1.2.1 Modeling is important 4
1.2.2 State-of-the-art in Timing Bug Modeling, Detecting and Fixing . . . 5
1.3 Contributions 6
1.4 Dissertation Organization 9

2 BACKGROUND AND RELATED WORK 11
2.1 LCbug Detection 11
2.1.1 Data Race Detection 11
2.1.2 Atomicity Violation and Order Violation Detection 12
2.1.3 Effect-oriented Detection 13
2.2 DCbugs and Distributed System Model Checkers 13
2.2.1 Distributed Concurrency Bugs (DCbugs) 13
2.2.2 Distributed System Model Checkers 14
2.3 LCbug Fixing 14
2.4 Empirical Studies of Timing Bugs 15

3 EMPIRICAL STUDY OF TIMING BUGS 16
3.1 Timing Bugs in Production-run Cloud Service Systems 16
3.1.1 Methodology 16
3.1.2 Characteristic Study and the Findings 17
3.2 Manual Patch Study for Timing Bugs in Multi-threaded Systems 19
3.2.1 Methodology 19
3.2.2 What are Manual Patches Like? 20
3.2.3 How about Existing Techniques? 23
3.2.4 How about the Future? 24
3.3 Conclusion 25

4 DCATCH: AUTOMATICALLY DETECTING MESSAGE TIMING BUGS IN CLOUD SYSTEMS . . . 26
4.1 Introduction . . . 27
4.1.1 Motivation . . . 27
4.1.2 Contributions . . . 30
4.2 DCatch Happens-Before (HB) Model . . . 31
4.2.1 Inter-node Concurrency and Communication . . . 34
4.2.2 Intra-node Concurrency and Communication . . . 36
4.2.3 Summary . . . 37
4.3 DCatch Tracing and Trace Analysis . . . 38
4.3.1 DCatch Tracing . . . 38
4.3.2 DCatch Trace Analysis . . . 41
4.4 Static Pruning . . . 43
4.4.1 Identifying Failure Instructions . . . 43
4.4.2 Impact Estimation . . . 44
4.5 Message Timing Bugs Triggering and Validation . . . 45
4.5.1 Enable Timing Manipulation . . . 45
4.5.2 Design Timing Manipulation Strategy . . . 46
4.6 Implementation . . . 47
4.7 Evaluation . . . 49
4.7.1 Methodology . . . 49
4.7.2 Bug Detection Results . . . 51
4.7.3 Performance Results . . . 55
4.7.4 Comparison with Alternative Designs . . . 57
4.8 Conclusion . . . 59

5 FCATCH: AUTOMATICALLY DETECTING FAULT TIMING BUGS IN CLOUD SYSTEMS . . . 60
5.1 Introduction . . . 60
5.1.1 Motivation . . . 60
5.1.2 Contributions . . . 62
5.2 Modeling Fault Timing Bugs . . . 65
5.2.1 Crash-regular Fault Timing Bugs . . . 66
5.2.2 Crash-recovery Fault Timing Bugs . . . 68
5.2.3 Discussion . . . 70
5.3 Tracing Correct Runs . . . 70
5.3.1 Which Correct Runs to Observe? . . . 70
5.3.2 What to Trace? . . . 72
5.4 Predicting Fault Timing Bugs . . . 73
5.4.1 Causal and Blocking Relationship Analysis . . . 73
5.4.2 Predicting Crash-Regular Fault Timing Bugs . . . 76
5.4.3 Predicting Crash-Recovery Fault Timing Bugs . . . 78
5.5 Fault Timing Bug Triggering . . . 82

5.6 Implementation Details . . . 83
5.7 Methodology . . . 85
5.8 Experimental Results . . . 86
5.8.1 Bug Detection Results . . . 86
5.8.2 Performance Evaluation . . . 90
5.8.3 Random Crash Injection . . . 91
5.8.4 Pruning and Triggering . . . 92
5.9 Conclusion . . . 93

6 HFIX: GENERATING HIGH QUALITY PATCHES FOR TIMING BUGS IN MULTI-THREADED SYSTEMS . . . 95
6.1 Introduction . . . 95
6.1.1 Motivation . . . 95
6.1.2 Contributions . . . 96
6.2 Bug Fix Background and Overview . . . 97
6.3 HFix$_{join}$ . . . 98
6.3.1 Patch Suitability Checking . . . 99
6.3.2 Patch Generation . . . 100
6.4 HFix$_{move}$ . . . 101
6.4.1 Overview of HFix$_{move}$ . . . 101
6.4.2 Identifying Move Opportunities . . . 102
6.4.3 Control Flow Checking . . . 104
6.4.4 Data Flow Checking . . . 106
6.4.5 Deadlock and Performance Checking . . . 107
6.5 Patch Merging . . . 107
6.5.1 Merge Add$_{\text{join}}$ Patches . . . 108
6.5.2 Merge Move$_{\text{join}}$ and Move$_{\text{create}}$ Patches . . . 109
6.6 Experimental Evaluation . . . 110
6.6.1 Methodology . . . 110
6.6.2 Experimental Results . . . 111
6.7 Conclusion . . . 114

7 CONCLUSIONS AND FUTURE WORK . . . 115
7.1 Contributions . . . 115
7.2 Limitation and Future Work . . . 116

REFERENCES . . . 119

LIST OF FIGURES

1.1 A LCbug example simplified from a real bug in PBZIP2. 2
1.2 A DCbug example in Hadoop MapReduce. 2
1.3 A general flow to tackle software bugs. 4

3.1 Timing-bug incident resolve strategy 18
3.2 A bug in Tranmission, with '+' and '-' denoting its manual/HFix patch. 22

4.1 Root cause of the message timing bug shown in Figure 1.2 28
4.2 A wide variety of causality relationships in HBase. 32
4.3 Concurrency and communication in MapReduce 34

5.1 An example of fault timing bugs from MapReduce 61
5.2 The flow of FCatch 64
5.3 Conflicting operations in a crash-regular bug. 65
5.4 Conflicting operations in a crash-recovery bug. 67
5.5 Observe two runs to detect a crash-recovery bug 71
5.6 An example of crash-regular bugs from HBase 76
5.7 Crashing (white) and recovery operations (black, gray). 79
5.8 An example of sanity check in Zookeeper 81
5.9 Our snapshot-and-crash wrapper 83

6.1 A simplified concurrency bug in PBZIP2. 96
6.2 General bug fixing process 97
6.3 Fixing OV timing bugs and AV bugs through Move 100
6.4 Control flow challenge for code movement. 103

LIST OF TABLES

3.1 Applications and bugs in study 19
3.2 Synchronization primitives in patches 21
3.3 LCbug manual fix strategies 21
3.4 Semantic-changing patches (B: Bypass strategy; T: Tolerance strategy). 25

4.1 Concurrency and Communication in Distributed Systems 33
4.2 HB-related Tracing in DCatch 40
4.3 DCatch Benchmark Bugs and Applications. 49
4.4 DCatch Bug Detection Results 51
4.5 # of message timing bugs reported by DCatch trace analysis 53
4.6 DCatch Performance Results 56
4.7 Break-downs of # of Major Types of Trace Records in DCatch 56
4.8 Full Memory Tracing Results. 57
4.9 False negatives (before '/') and false positives (after '/') of ignoring certain HB-related operations 58

5.1 FCatch Benchmarks. 85
5.2 fault timing bugs found by FCatch. Res.: resources in contention; H: heap object; ZK: ZKnode; GF: global file; LF: local file. 87
5.3 FCatch bug detection results. *: same bug; Old: benchmark bugs; - : benchmark bug is not of this category; Exp.: false positives that throw exceptions. 89
5.4 FCatch performance. 90
5.5 # false positives pruned by different analysis 92

6.1 Patch comparison for OV bugs 109
6.2 HFix patch comparison for AV bugs 113

ACKNOWLEDGMENTS

The past five years had been a wonderful time at the University of Chicago, a Hogwarts school in my heart. I would like to express my most sincere gratitude to so many people who have helped me through the years.

First and foremost, I would like to thank my advisor, Prof. Shan Lu, for being a great mentor in research and life. Her professional expertise and insights helped me diving into cutting-edge research topics; her invaluable suggestions guided me through challenges in research projects and daily life; and her passion about research and life had a profound impact on my critical thinking, open mindset, collaborative attitude, and communication. It's my extremely honor to have become Shan's first PhD student at UChicago. I do not think I can reach the destination without her continued support in this tough journey.

Thanks also go to the other members of my committee: Prof. Haryadi S. Gunawi, Prof. Ryan Huang, and Dr. Suman Nath for being on my doctoral dissertation committee, and Prof. Ravi Chugh for being on my master's committee. Their invaluable suggestions and comments helped me shape the research direction and improve my thesis. I am really honored to have them on my committee.

It is my truly fortunate to work with many excellent colleagues. I would like to thank my mentors/collaborators in MSR and every member of Shan's and Haryadi's group: Dr. Madan Musuvathi, Dr. Yingnong Dang, Dr. Chuanxiong Guo, Linhai Song, Yuxi Chen, Guangpu Li, Shu Wang, Junwen Yang, Chi Li, Lefan Zhang, Chengcheng Wan, Dr. Xu Wang, Xianglan Chen, Jiaxin Li, Wei Yuan, Yu Gao, Jeffrey Lukman, Tanakorn Leesatapornwongsa, and Riza Suminto for the opportunity of research collaborations and technical discussions. I also want to thank the infrastructure support from Bob Bartlett, Phil Kauffman, and Colin Hudler that help and escort my research projects. I specially appreciate Margaret P. Jaffey and Sandy Quarles for their kind and professional administration help.

Besides, I would like to thank my fantastic friends: Min Xu, Yuanwei (Kevin) Fang, Zhixuan Zhou, Liwen Zhang, Xi Liang, Dixin Tang, Yuan Li, Yun Li, Shiqin Yan, Kai Li, Aiman Fang, Qingqin Zheng, Rida Assaf, Steven Basart, and many more that I cannot list them all here for many happy parties and dinners.

Finally, specially thanks to my family. I would like to thank my parents for their unconditional and greatest love. Thank you, Dad and Mom! I also would like to thank Teng for the support and love in heart. Thank you, Teng!

ABSTRACT

Multi-threaded software and distributed cloud software are prevalent as a dominant backbone for modern applications. Although it is extremely important, their reliability is severely threatened by software bugs. Among all types of software bugs, timing bugs are among the most troublesome due to their inherent non-deterministic nature and the huge interleaving space. Timing bugs are caused by unexpected timing among local events in multi-threaded systems (local concurrent bugs or LCbugs) or distributed events, such as message or faults, in distributed systems (distributed concurrency bugs or DCbugs).

A timing bug model is critical to guide the design of automated tackling tools, which includes three parts: concurrent source, synchronization mechanisms, and sharing resources. Existing timing bug models mainly focus on the thread interleaving concurrent source, the lock-related synchronization mechanism, and shared global memory resources for LCbugs.

To fight timing bugs and improve the concurrent software reliability in multi-threaded systems and distributed systems, this dissertation works on these three parts and makes the following contributions:

First, this dissertation conducts an empirical study of timing bugs in multi-threaded systems and distributed cloud service systems to understand how common are timing bugs, what's the resource being competed, and how were they resolved or fixed. Our empirical study includes two parts. (1) we conduct a comprehensive characteristic study on real-world incidents in Microsoft Azure production-run cloud services. The study reveals several main findings: (a) about 15% software bug incidents in our study set are caused by timing bugs; (b) 60% timing bugs in our study set are DCbugs (message timing bugs or fault timing bugs); (c) half of the timing bugs in our study set are racing on persistent data instead of shared global memory variables; (d) mitigation strategy, especially running-environment mitigation, is widely used to resolve timing bug incidents in the cloud. (2) we conduct an empirical study of manual patches for real-world LCbugs in multi-threaded systems to understand the gap between automatically generated patches and manually generated patches. The study finds that (a) lock is the dominant synchronization primitive for enforcing atomicity; lock-related signals/waits are not the dominant primitive for enforcing pairwise ordering in patches. (b) leveraging existing synchronization in software is as common as adding extra primitives. These findings provide many motivation and guidelines for the design of timing bug tackling tools in this field.

Second, guided by the empirical study, this thesis proposesnew models and detection tools for message timing bugs and fault timing bugs in distributed systems. Our new model captures two new concurrent sources (message interleavings and random faults), the new synchronization mechanisms introduced by them, and new sharing resources, persistent data. Guided by the proposed model, detection tools are designed to predict message timing bugs and fault timing bugs from correct runs. Each step of our detection tool is carefully customized to address the unique challenges for DCbugs in distributed systems. The evaluation result shows that our tool can effectively and efficiently detect message timing bugs and fault timing bugs with low false positive rates.

Third, motivated by the findings of LCbug fixing strategies, we design a fixing tool to model and enforce timing relationship by leveraging existing non-lock synchronization primitives. Evaluation using real-world bugs shows that our tool can automatically generate patches that have matching quality with manual patches and are much simpler than those generated by the previous state of the art techniques.

CHAPTER 1
INTRODUCTION

In the multi-core era, timing bugs or concurrency bugs [1] widely exist and severely hurt software reliability in multi-threaded systems. Their unique non-deterministic nature makes them very difficult to tackle. Although great progress has been made in automated concurrency bug fixing techniques recently, patches generated by existing tools are mostly different from patches designed by developers. In the cloud computing era, the timing bug problem is getting even worse. New types of non-determinism, inter-node message/event interleavings and random faults, are introduced in distributed systems, which go beyond traditional intra-node thread interleavings.

Facing the challenge of timing bugs in multi-threaded systems and distributed systems, this dissertation conducts an empirical study of timing bugs in these systems, proposes novel timing bug models, and designs fixing and detecting tools guided by the model we proposed.

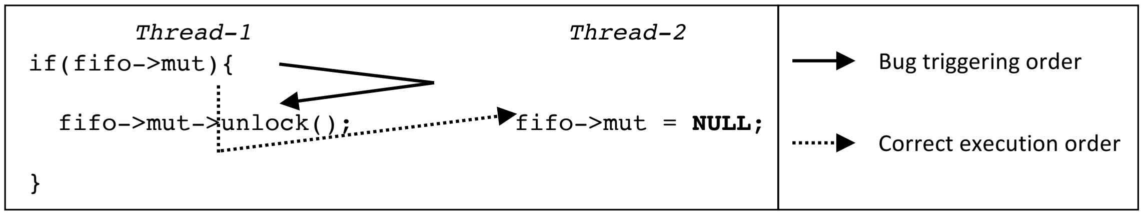
1.1 Motivation

1.1.1 Timing Bugs

Software systems play a critical role in modern society, but their reliability is severely threatened by software bugs, flaws in a program that cause the program to produce incorrect or unexpected results. Recent reports show that software failures impact half of the world's population (3.7 billion people) and cause USD 1.7 trillion in financial losses in 2017 [13].

Among all types of software bugs, timing bugs are one of the most troublesome. Timing bugs are non-deterministic bugs triggered by unexpected timing among events in software systems. An event can be a local computation, message arrival/sending, fault (e.g., node crashes, message drops, timeouts), and reboot. Local concurrency bugs (LCbugs) are timing

1. In this dissertation, the terms timing bug and concurrency bug are used interchangeably.



```
if(fifo->mut){
   fifo->mut->unlock();              fifo->mut = NULL;
}
```

Figure 1.1: A LCbug example simplified from a real bug in PBZIP2.

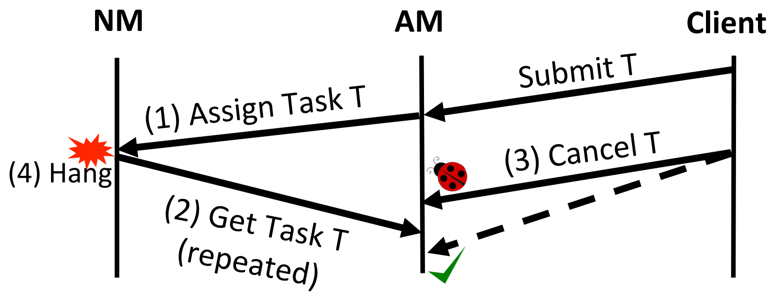


Figure 1.2: A DCbug example in Hadoop MapReduce.

bugs caused by local computation events in single-machine multi-threaded systems; distributed concurrency bugs (DCbugs) are timing bugs caused by distributed events (message, fault, and reboot) in cloud-scale distributed systems.

Figure 1.1 shows a LCbug example simplified from a real bug in PBZIP2. There are two concurrent threads, `Thread-1` and `Thread-2`, and a shared variable `fifo->mut`. The code is being carefully designed here that `Thread-1` first checks the shared variable `fifo->mut` in a `if` statements and access this shared variable only if it is not `NULL`. If we consider these two threads separately, this piece of code looks correct. However, developers easily forget that other threads (`Thread-2` in this example) could concurrently access the shared variable `fifo->mut`. This unexpected timing order of local computation events eventually triggers the bug. In this example, when `Thread-2` nullifies `fifo->mut` in the middle, the program crashes due to a null pointer dereference in `Thread-1`.

Figure 1.2 illustrates a real-world DCbug example from Hadoop MapReduce. There are three nodes, Node-Manager (NM), Application-Manager (AM), and the client. The program is also being carefully designed that after AM assigns a task `T` to a container in NM (#1), this NM container tries to retrieve the content of task `T` from AM (#2) and retries this retrieval request until it gets a valid return from AM. Meanwhile, the client can cancel a task `T` (#3) after the submitting. If we consider the client and NM separately, each piece of code looks logically correct. However, developers easily forget that the cancel message (#3) can arrive AM before the retrieval request from the NM container (#2). If these two messages are ordered in this timing, when the retrieval request is delivered to AM, task `T` has already been canceled upon the client's request (#3). Not anticipating this timing scenario, the NM container hangs (#4), waiting forever for AM to return task `T`.

1.1.2 LCbugs and DCbugs are Difficult to Tackle

As it is shown in Figure 1.1 and Figure 1.2, writing correct concurrent programs is not easy. To leverage more than one processing core in local multi-threaded systems or more than one node in distributed cloud systems, software needs to have multiple components (e.g., computation tasks, messages) that can be executed on different cores or nodes concurrently. This inevitable trend introduces three challenges while developers write concurrent programs or tackle timing bugs in these systems.

First, most developers think and program sequentially, instead of concurrently. They are used to thinking about one component or one event at a time, such as `Thread-1` in Figure 1.1 and the NM container in Figure 1.2. This sequential thinking habit can easily make mistakes when multiple components or events in a program execute concurrently.

Second, the interleaving space of timing bugs introduced by this trend is huge. With more concurrency mechanisms provided by the underlying system, the size of a program's interleaving space is exponentially growing with the execution length of the program. For example, these three statements in Figure 1.1 have three possible interleavings overall (only one interleaving is the buggy order). If there are four statements, the number of possible

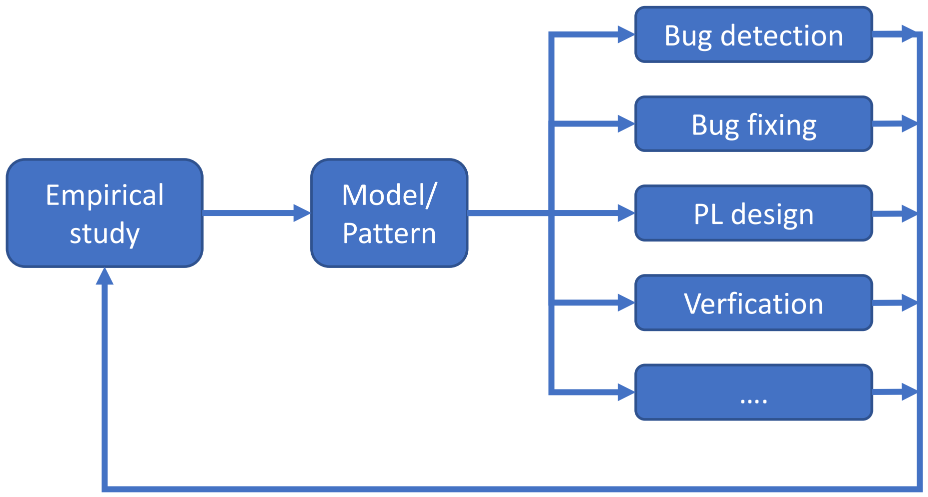


Figure 1.3: A general flow to tackle software bugs.

interleavings is six. A similar exponential growth can be seen in DCbugs.

Finally, the manifestation of a timing bug is non-deterministic. The execution result of concurrent or distributed programs depends not only on the input but also the interleaving. Even if a timing bug manifests once, it is very likely that this bug would disappear in the next run with the same input. The timing bug in Figure 1.1 and Figure 1.2 may or may not manifest depending on which interleaving it takes (the solid arrows or the dotted arrows in the figure). This non-determinism property makes timing bugs challenging to manifest and debug in software systems.

1.2 Approaches to Tackle Timing Bugs

1.2.1 Modeling is important

A software bug is a flaw in a computer program that causes the program to produce incorrect or unexpected results. Figure 1.3 shows a general flow to tackle software bugs. Empirical studies have always been crucial in motivating and guiding the fight against software bugs to improve software reliability. Following new directions or findings pinpointed by empirical studies, modeling software bugs provides a general and concrete pattern to accurately define and formalize software bugs. These models and patterns could be further adopted into software bug detection, software bug fixing, program language design and so on. Finally, this process can repeatedly iterate to evolve software systems and achieve high reliability. Overall, modeling is an important step to tackle software bugs.

1.2.2 State-of-the-art in Timing Bug Modeling, Detecting and Fixing

A model of timing bugs generally includes three parts: concurrent source, synchronization mechanisms, and sharing resources.

Concurrent source defines the underlying concurrency mechanism. For example, the concurrent source in Figure 1.1 is thread interleavings, but in Figure 1.2, the concurrent source is message interleavings. *Synchronization mechanisms* capture the timing relationship enforcing mechanisms among operations, such as `pthread_mutex_lock` primitive in the pthread library or the partial order introduced by causality relationship. *Sharing resources* define resources on which timing bugs could be competing. For example, the previous LCbug and DCbug examples are racing on shared global memory variables.

Existing empirical study Empirical study is crucial in motivating and guiding the improvement of software availability. Thorough studies have been conducted for LCbugs and DCbugs [64, 26, 51] with many follow-up work to date. Previous studies review bugs mainly in open-source bug databases. Although useful, these studies have not and cannot, due to the limitation of their data sources, provide in-depth understanding about production-run cloud service incidents, answering fundamental questions like: what are timing bugs in the cloud escaped in-house testing and how were they resolved. Answers to these questions would be crucial to improving the availability of distributed cloud systems.

Traditional LCbug detection approaches Many concurrency bug detection tools have been proposed to predict LCbugs in different bug patterns: data race, atomicity violation, order violation, and effect-oriented detection. Although amazing results have been achieved, these works are all about shared global memory accessing issues caused by thread interleavings. These tools cannot adopt to DCbug detection as their model cannot capture new concurrent sources, such as message interleavings and random faults, and new synchronization mechanisms introduced by distributed systems. Effective techniques to detect DCbugs are desired.

Traditional LCbug fixing approaches Many automated fixing techniques dedicated to LCbugs have been proposed to automatically generate patches leveraging a unique property of LCbugs— since concurrency bugs manifest non-deterministically, the correct computation semantics already exists in software. These tools work not by changing computation semantics, but by adding synchronization operations , including locks and condition variable signals/waits, into software. However, patches generated by existing tools are mostly different from patches manually designed by developers. Leveraging existing non-lock synchronization to enforce timing relationship is a common fix strategy in manual patches. How to generate patches as simple and well-performing as manual patches is still a challenging research problem.

1.3 Contributions

This dissertation works on three components to address the timing bug problem: understanding real-world timing bugs in production-run cloud service systems and multi-threaded systems through empirical studies, modeling and detecting timing bugs in distributed systems (DCbugs), modeling and fixing timing bugs in multi-threaded systems (LCbugs). The contribution of these three components interacts and complements each other to extend timing bug models from different aspects. Specifically, in the concurrent source aspect, our work extends the model to capture message/events interleavings and random faults;

in the synchronization mechanism aspect, we extend the model to cover non-lock thread-, message/events-related synchronization and fault handling. In the sharing resource aspect, this dissertation extends the model to include persistent data.

(1) Understanding timing bugs in multi-threaded systems and production-run cloud service systems To answer questions how common are timing bugs, what's the resource being competed, and how were they resolved or fixed, this dissertation conduct an empirical study of timing bugs in multi-threaded systems and production-run cloud service systems. Our empirical study includes two parts.

(a) Timing bugs in production-run cloud service systems Cloud service incidents caused by timing bugs adversely affect the expected service operations, and they are extremely costly in terms of user impacts and engineering efforts required to resolve them. Unfortunately, there is limited understanding about timing bugs in cloud services that actually happen during production runs. We conducts an empirical study on a large number real-world incidents in Microsoft Azure services to provide an in-depth understanding and answer four fundamental questions: how common are timing bugs, how common are message timing bugs and fault timing bugs, what is the sharing resource being competed, and how were they resolved in the cloud. Our study finds that: (i) about 15% software-bug incidents in our study set are caused by timing bugs; (ii) 60% timing bugs are message timing bugs or fault timing bugs; (iii) half of timing bugs in our study are racing on persistent data instead of shared global memory variables; (iv) mitigation, especially running-environment mitigation, is a widely used strategy to resolve timing-bug incidents in the cloud.

(b) Understand LCbug manual patches in multi-threaded systems Although indispensable, timing bug fixing is time-consuming and error-prone. Patches automatically generated by existing fixing techniques mostly insert locks and lock-related synchronization primitives into software. However, there is a big gap between automatically generated patches and manually generated patches; less than one third of real-world local concurrency bugs are manually fixed through adding or changing lock primitives. Facing this gap, we conduct a characteristic study of manual patches for real-world local concurrency bugs. This study of manual patches for real-world LCbugs reveals many interesting findings: (i) lock is the dominant synchronization primitive for enforcing atomicity; lock-related signals/waits are *not* the dominant primitive for enforcing pairwise ordering in patches. (ii) leveraging existing synchronization, such as thread creation, in software is as common as adding extra new primitives.

(2) Modeling and detecting timing bugs in distributed systems Different from previous work on LCbug detection, DCbugs in distributed systems have their unique non-determinism mechanisms, causality relationships, and synchronization properties. Without a precise DCbug model, detection techniques cannot predict them effectively and efficiently. Guided by the empirical study, this dissertation focuses on message timing bugs and fault timing bugs, two of the most common types of DCbugs. For each type of DCbugs, a new model is proposed to capture their unique property and a detection tool guided by the proposed model is designed as follows.

(a) DCatch: a message timing bug model and detection technique Message timing bugs are caused by untimely message arrivals in distributed systems. The biggest challenge in detecting them is the more complicated timing relationship in distributed systems than LCbugs in multi-threaded systems. To address this challenge, DCatch first proposes a new model to capture a wide variety of causality and synchronization mechanisms in real-world distributed systems. Guided by this new model, we build the DCatch tool to detect message timing bugs by monitoring correct executions with four steps: tracing, trace analysis, static pruning, and triggering. Each step is carefully customized to address unique challenges for DCbugs. The evaluation on four representative open-source distributed systems shows that DCatch can detect message timing bugs effectively and efficently.

(b) FCatch: a fault timing bug model and detection technique Fault timing bugs are caused by untimely faults in distributed systems. The biggest challenge is that previous work relies on manual specification or domain knowledge to hit fault timing bugs, which is daunting and inefficient. To address this challenge, we treat fault timing bugs as a type of timing bugs rather than semantic bugs. We first build a new model that regards faults as a type of non-determinism and captures new causality relationship introduced by hardware faults. Following this new model, we design the FCatch tool to detect fault timing bugs in four steps: tracing, identifying conflicting operations, identifying fault-intolerant operations, and triggering. The evaluation on six common workloads in four widely used open-source distributed systems shows that FCatch is effective to find severe fault timing bugs with low false-positive rates by only observing correct execution.

(3) Modeling and fixing timing bugs in multi-threaded systems with high-quality patches Guided by the empirical study of manual patches, this dissertation designs a fixing tool, HFix, to automatically generate high-quality patches for LCbugs. The first, HFix$_{join}$, enforces ordering relationship by adding thread-join primitives, instead of signals/waits. The second, HFix$_{move}$, enforces ordering and atomicity relationship by leveraging synchronization primitives that already exist in software. The evaluation on real-world LCbugs shows that it can automatically generate patches that have matching quality with manual patches and are much simpler than those generated by previous state of the art technique.

1.4 Dissertation Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces previous work on local concurrency bug detection and fixing, distributed concurrency bug study, and other related topics. Chapter 3 discusses our empirical study of timing bugs in multi-threaded systems and production-run cloud service systems. Chapter 4 and 5 present our DCbug models and detection tools, DCatch and FCatch, for message timing bugs and fault timing bugs. Chapter 6 presents our automated fixing fool, HFix, for local concurrency bugs. Chapter 7 concludes this thesis and discusses future research work.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter presents the background and the related work of this dissertation focusing on the timing bug research literature. Chapter 2.1 discusses existing approaches on detecting local concurrency bugs. Chapter 2.2 discusses distributed concurrency bugs and distributed system model checkers. These approaches are closely related to this dissertation's Chapter 4 and 5. Chapter 2.3 discusses previous work on local concurrency bug automated fixing, which is relevant to this dissertation's Chapter 6. Chapter 2.4 discusses the related work of this dissertation's Chapter 3 on empirical studies of timing bugs.

2.1 LCbug Detection

Many techniques have been proposed to detect LCbugs, conflicting accesses on shared global memory caused by thread interleavings. They can be categorized into three categories based on the different types of bug patterns in they detection: data race detection, atomicity violation and order violation detection, and effect-oriented detection.

2.1.1 Data Race Detection

Data race is a common type of concurrency bugs. A data race occurs when two memory accesses from different threads access the same shared data without being properly synchronized and at least one access is a write operation [14].

Three major types of detection techniques have been proposed: lock-set [82] based detection, happens-before [48] based detection and hybrid detection combining both lock-set and happens before approaches. The lock-set based approach tracks lock sets that are used to protect each shared memory location. It reports data race bugs when the lock set of a memory location becomes empty. According to Lamport's happens-before relation [48], happens-before detection traces synchronization and causality operations. It calculates the happens-before order among memory accesses and reports data race bugs if there is no happens-before order between two conflicting accesses. These two approaches have their own advantages; the lock-set approach can report data race bugs that do not covered during the monitoring runs, while the happens-before approach is more accurate with lower false-positive rates. Hybrid approaches [98, 73] have been proposed to combine their advantages. It can detect more bugs than the happens-before approach with lower false positive rates than the lock-set approach.

2.1.2 Atomicity Violation and Order Violation Detection

Different from data races that are semantic-unrelated, atomicity violation and order violation associate with program semantics. Specifically, atomicity violation bugs are caused by concurrent execution unexpectedly violating the atomicity of a certain code region assumed by programmers; order violation bugs are caused by concurrent execution unexpectedly violating the order relationship of two (groups of) operations assumed by programmers [64].

Many techniques have been proposed that require programmers to specify expected atomic code regions or order relationships. They provide a new programming language feature, such as a new type system, and an associated verification system to detect potential atomicity violation bugs or order violation bugs. A limitation of these techniques is that it is impractical to rely on programmers' annotations to specify all semantic information. To address this limitation, several detection tools have been proposed and do not require manual annotations. SVD [94] uses data dependency and control dependency to infer atomic regions. AVIO [65] automatically extracts access invariants and atomic regions from correct runs. MUVI [63] automatically infer multi-variable correlations through static source code analysis and data mining.

2.1.3 Effect-oriented Detection

Apart from above cause-oriented approaches, an effect-orient detection technique is proposed recently. It starts from potential failure/error sites of a program and detect concurrency bugs in a backward fashion to find possible interleavings that can trigger the failure. ConMem [102] handles memory errors caused by concurrency bugs starting from potential memory error sites (e.g., pointer dereference) and conducting dynamic analysis to identify the buggy interleaving that leads to the memory errors, such as null pointer dereferences, dangling pointer, and uninitialized reads. ConSeq [101] handles semantic errors caused by concurrency bugs starting from potential failure sites (e.g., assertion failures) and conducting intra-thread slicing to locate potential critical reads and buggy interleavings that leads to the failures, such as assertion failures, error message prints.

2.2 DCbugs and Distributed System Model Checkers

2.2.1 Distributed Concurrency Bugs (DCbugs)

Distributed concurrency bugs (DCbugs) are caused by non-deterministic orders of distributed events in distributed systems. Distributed events could be message arrivals, hardware crashes/reboots, network timeout, etc.

A comprehensive real-world DCbug study, TaxDC [51], focusing on bug triggering, bug symptom, and bug fixing has been conducted. TaxDC studies 104 DCbugs collected from widely-deployed cloud-scale datacenter distributed systems including Cassandra, Hadoop MapReduce, HBase, and ZooKeeper. For bug manifestation conditions, the study shows that 64% of DCbugs are triggered by untimely delivery of messages, and 32% of DCbugs are triggered by untimely faults or reboots, and occasionally by a combination of both (4%). The findings of this study have driven DCbug research along many directions, including but not limited to bug detection and fixing. Our work, DCatch in Chapter 4 and FCatch in

Chapter 5, is also directly motivated by the findings from this study.

2.2.2 Distributed System Model Checkers

Distributed system model checkers (or *dmck* in short) is an implementation-level model checker. It works by intercepting non-deterministic distributed events and permuting their ordering and hereby pushing the target system into corner-cases and unearthing hard-to-find bugs. However, the more events it included, the more scalability issues will arise due to the state-space explosion.

To address the state-space explosion issue, distributed system model checkers have many variance proposed in literature recently. Existing dmcks adopt a random walk or basic reduction techniques. MACEMC [46] combines DFS and random walk biased with weighted events manually labeled by testers. A popular reduction technique widely adopted in state-of-the-art dmcks is dynamic partial order reduction (DPOR [25]) which exploits the independence of events to reduce the state explosion. DPOR-based dmcks can be categorized into black-box manner or white-box manner. Black-box approaches (CrystalBall [95], dBug [88], and MoDist [96]) treat target systems as a complete black box and permute events without any domain-specific knowledge. White-box approaches (SAMC [50] and FlyMC [67]) further reduce redundant event orderings by leveraging simple semantic knowledge of target systems.

2.3 LCbug Fixing

Concurrency bugs are the most difficult to fix correctly among common bug types [64, 97] with many incorrect patches released. Recently, several automated fixing techniques dedicated to LCbugs in multi-thread systems have been proposed.

Gadara [90] is the first automated approach to systematically address deadlock bugs. It adds new lock to disallow the buggy interleaving that expose deadlock bugs. AFix [40] is the first automated atomicity violation fixing approach proposed by adding new lock primitives to synchronize the interleaving statements between atomic regions. Axis [60] models a program's concurrent property as Petri nets and uses the SBPI control theory as the theoretical foundation to fix atomicity violations. Grail [58] builds a contextual analysis model to further distinguish different aliasing contexts to achieve a better efficiency. CFix [42] is the first automated order violation bug fixing technique proposed by inserting signal and wait primitives to enforce order timing.

2.4 Empirical Studies of Timing Bugs

Empirical studies have always been crucial to guide research effort towards the improvement of software reliability. Several dedicated studies have been conducted focusing on timing bugs in multi-threaded systems and distributed systems [64, 26, 51], and synchronization-related code changes [29, 74, 80, 92]. Many other studies about cloud systems also cover timing bugs in their root cause reviewing [31, 32].

These studies use two types of data sources: (1) open-source bug databases, which contain detailed information about bugs found during both in-house code review/testing and production runs [31, 51, 99, 64, 26, 29, 74, 80, 92]. They could not check which issue actually cause production incidents and how they were resolved during production (all issues ended up with code patches). (2) news reports about software failures or cloud outages, which contain detailed failure-impact information [32]. Most (76%) public reports do not discuss details about how outages were resolved, and many (60%) do not explain outage root causes. Consequently, this study mainly focuses on cloud system outage duration and coarse-granularity cause breakdowns.

CHAPTER 3
EMPIRICAL STUDY OF TIMING BUGS

Cloud services have become the backbone of today's computing world. Runtime *incidents*, which adversely affect the expected service operations, are extremely costly in terms of user impacts and engineering efforts required to resolve them. Hence, such incidents are the target of much research effort. Unfortunately, there is limited understanding about timing-bug incidents that actually happen during production runs: what cause them and how were they resolved in the cloud.

Although great progress has been made, patches generated by existing tools for LCbugs are mostly *different* from patches manually designed by developers. Existing auto-patches mostly insert locks and lock-related synchronization operations into software [40, 58, 60, 90]; yet, less than one third of real-world concurrency bugs are fixed by developers through adding or changing lock operations [64]. Clearly, we need a better understanding of the gap between automatically generated patches and manually generated patches, so that we can eventually design auto-fixing tools that generate not only correct but also simple and well-performing patches, appealing to developers.

This chapter presents our empirical study of timing bugs in production-run cloud service systems and multi-threaded systems.

3.1 Timing Bugs in Production-run Cloud Service Systems

3.1.1 Methodology

Microsoft Azure production incidents can be reported by (Microsoft internal or external) users or by system watchdogs that keep monitoring if certain system metric goes beyond a pre-configured threshold. Every incident is recorded in the incident database, associated with information such as user description or watchdog report, developers' discussion, severity-level tag, root cause description, work items issued to developer teams (if any), resolving strategy, the incident-impact duration, etc.

In this study, we focus on a set of 112 incidents. They are *all* the incidents that satisfy the following four conditions in a 6-month period (March 5th, 2018 – September 5th, 2018):

(1) the incident is not a false alarm and its severity level indicates that new features cannot commit into production environment until this incident is resolved;

(2) the incident led to changes in the cloud service, such as bug-fixing patches, testing enhancement, etc;

(3) the incident report contains enough information for us to judge the root cause;

(4) the root cause of the incident are software bugs.

Note that not all the 112 incidents we studied affected Microsoft's external customers. Many incidents affected Microsoft's internal users and many others were detected by internal users and automated watchdogs and mitigated before external customers reported them.

Our study aims to answer four questions: (1) how common are timing bugs; (2) how common are message timing bugs and fault timing bugs; (3) what's the resource being competed; (4) how were they resolved in the cloud.

3.1.2 Characteristic Study and the Findings

Q1: how common are timing bugs? Overall, there are about 15% software-bug incidents in our study set are caused by timing bugs. Timing bugs are the third biggest root cause category in our study.

Q2: how common are message timing bugs and fault timing bugs? Among all timing bugs in our study set, 30% cases are message timing bugs; 30% are fault timing bugs.

Q3: what's the resource being competed? half of these bugs are racing on persistent data like cached firewall rules, configuration entries, znodes in Zookeeper, database data, and others, instead of shared memory variables that traditional timing bugs race on. For

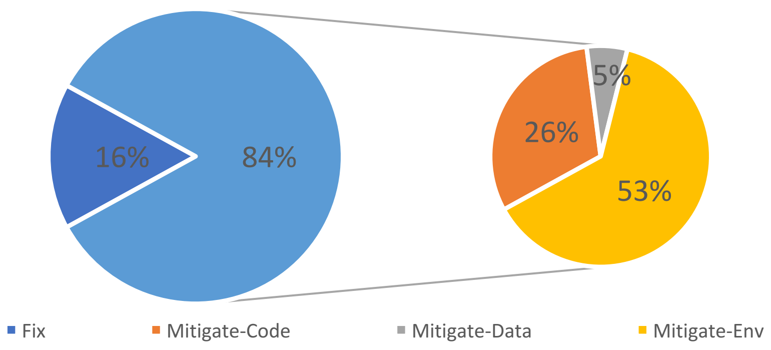


Figure 3.1: Timing-bug incident resolve strategy

example, in one case, two system processes read and write the same entry in the machine's configuration file. Races between these two processes' reads and writes eventually led to repeated machine restarts.

Q4: how were they resolved? Facing tight time pressure, more often than not, timing-bug incidents were resolved through a variety of *mitigation* techniques (84%) without patching the buggy code (16%), providing quick solutions to users and maximizing service availability. Note that, it is possible that an incident first got resolved by a mitigation technique and later led to a software patch that was not tracked by the incident report.

We categorize all mitigation techniques into three categories: code mitigation, data mitigation, and running-environment mitigation. As shown in Figure 3.1, these three strategies are all widely used, with environment mitigation the most common in our study.

Code mitigation mainly involves rolling back the software to an older version, or disabling certain code snippets such as an unnecessary/outdated sanity check that failed users' requests and caused severe incidents.

Data mitigation involves manually restoring, cleaning up, or deleting data in a file, a cloud table, etc.

Running-environment mitigation cleans up dynamic environment through killing/restarting processes, migrating workloads, adding fail-over resources, etc.

App.	Description	# Bugs	
		AV	OV
Apache	Web Server	8	5
Mozilla	Browser Suite	26	15
MySQL	Database Server	10	2
OpenOffice	Office Suite	3	2
Misc.	Cherokee web server, Click router, FFT benchmark, PBZIP2 compressor, Transmission bittorrent client, and X264 encoder	1	5
Total		48	29

Table 3.1: Applications and bugs in study

3.2 Manual Patch Study for Timing Bugs in Multi-threaded Systems

3.2.1 Methodology

Our study aims to answer three sets of questions.

What are manual patches like? What are the fix strategies and synchronization primitives used by manual patches? Are all concurrency bugs fixed by constraining the timing? Does any patch change sequential semantics? How do patches vary for different types of concurrency bugs?

How are existing techniques? How do existing tools work, particularly compared with patches manually developed by developers?

How about the future? How might future tools generate patches that match the quality of manual patches?

To answer the above questions, we review the manual patches of 77 bugs. These bugs come from two sources. The first is the real-world LCbug benchmark suite created by previous work [64]. Among the 74 non-deadlock bugs in that suite[1], a couple of them are not completely fixed by developers and hence are excluded from our study. The remaining

1. Our study focuses on non-deadlock concurrency bugs.

71 are shown in the top half of Table 3.1. The second part includes all the bugs evaluated by recent concurrency bug detection and fixing papers [40, 42, 101] that have available manual patches and are not included in the first source, shown in the "Misc." row of Table 3.1.

These bugs come from a broad set of C/C++ open-source applications, that include both big server applications (e.g., MySQL database server and Apache HTTPD web server) and client applications (e.g., Mozilla web-browser suite, and many others). These applications are all widely used, with a long software development history.

Among these bugs, 48 of them are atomicity violation (**AV**) bugs and 29 of them are order violation (**OV**) bugs. We carefully study the final patch of each bug. We also read developers' discussion on the on-line forums [1, 2, 3, 4] and software source code to obtain a deep understanding of each patch. Every bug was reviewed by all authors, with the patch categorization cross-checked by all authors.

Threats to Validity Like all empirical studies, our study cannot cover all concurrency bugs. Our study only looks at C/C++ user-level client and server applications, and does not cover Java programs, operating systems software, or high-performance computing software. Our study does not look at deadlock bugs, and also does not cover bugs that are related to the newly minted C++11 concurrency constructs. Our study does not and cannot cover all concurrency bugs in Apache, MySQL, Mozilla, and other software in our study. Our main bug source, the benchmark from previous work [64], is based on fixed concurrency bugs randomly sampled from the above applications' bug databases. All our findings should be interpreted with our methodology in mind.

3.2.2 What are Manual Patches Like?

Synchronization Primitives As shown in Table 3.2, a big portion of bugs are fixed *without* using any synchronization primitives (about half). Most of these bugs are fixed without disabling the buggy timing, which will be explained later.

	Lock	Con.Var.	Create	Join	Misc.	None
AV	18	1	0	0	2	27
OV	4	3	6	4	5	7
Total	22	4	6	4	7	34

Table 3.2: Synchronization primitives in patches

	Prevention				
	Timing		Instruction	Data	Tolerance
	Add_S	$Move_S$	Bypass	Private	
AV	15	6	13	8	6
OV	10	14	1	0	4
Ap	2	5	3	0	3
My	1	3	2	3	2
Mo	14	10	8	5	5
Op	4	0	1	0	0
Misc.	4	2	0	0	0
Total	25	20	14	8	10

Table 3.3: LCbug manual fix strategies

Among patches that leverage synchronization primitives, there is a clear distinction between atomicity violation and order violation patches. In AV patches, lock is the single dominant synchronization primitive; rarely, condition variables, interrupt disabling, and atomic instructions are used. In OV patches, condition variable signal-waits, thread creates, and thread joins are about equally common. Occasionally, customized synchronizations like spin loops are used.

Fix Strategies Concurrency bugs are caused by instructions that access shared variables under unexpected timing. Patches can prevent these bugs in three ways: (1) change the timing among those instructions (*Timing* in Table 3.3), which can be achieved by either adding new synchronization (Add_S) or moving around existing synchronization ($Move_S$); (2) bypass some instructions under the original buggy context (*Instruction Bypass*); (3) make some shared variables private under the original buggy context (*Data Private*). Patches

```
//child thread
assert(h->band); //B
```

```
 //parent thread
 void tr_sessionInit(...){
   h = malloc(...);
+  h->band = bdNew(h);
   tr_eventInit(...);
   ...
-  h->band = bdNew(h); //A
 }

 void tr_eventInit(...){
   pthread_create(...);
 }
```

Figure 3.2: A bug in Tranmission, with '+' and '-' denoting its manual/HFix patch.

could also tolerate the effect of concurrency bugs, instead of preventing them (*Tolerance*). The break-down of these strategies is shown in Table 3.3.

Overall, as shown in Table 3.3, constraining the timing through new or existing synchronization is the most common fix strategy, applied to almost 60% of bugs in study. Other fix strategies are not as common, but still non-negligible, each applied to at least 10% of studied bugs.

Among patches that use the *Timing* fix strategy, about half add new synchronization operations into the software, and the other half leverage existing synchronization operations. For the latter type, the patch is always done by code movement. For example, the real-world bug illustrated in Figure 3.2 is fixed by moving variable initialization (A) before child-thread creation in the parent thread, so that the child thread is guaranteed to read an already-initialized value (B).

8 bugs are fixed by making some instructions involved in the bug access local, instead of shared, variables. We will discuss them in more details in Chapter 3.2.3.

Patches with instruction-bypassing and bug-tolerance strategies change the sequential computation semantics (24 out of 77). Note that all previous concurrency-bug fixing work [19, 40, 42, 45, 58, 59, 60] uses an *opposite* assumption and only produce patches that preserve sequential semantics.

3.2.3 How about Existing Techniques?

Adding locks and condition variables Recently, several tools have been built to automatically fix concurrency bugs by adding locks, such as AFix, Grail, and Axis [40, 58, 60], and condition variables, such as CFix [42]. These techniques provide general fixing capability that applies for a wide variety of concurrency bugs.

Our empirical study shows that these techniques indeed emulate the most common type of manual fix strategies – add new synchronization (Add_S), as shown in Table 3.3.

However, there are many bugs that are **not** fixed through Add_S by developers (>40% in our study). In many cases, other fix strategies can produce much simpler patches and introduce fewer synchronization operations into software than Add_S, such as in Figure 3.2.

Another limitation for this series of tools is that they only look at two types of synchronization primitives: locks and condition variables. Locks are indeed the most dominant primitive for fixing AV bugs. However, condition variables are **not** the most dominant primitive for fixing OV bugs, as shown in Table 3.2. In fact, among the 10 OV bugs that are fixed by adding new synchronizations, only 3 of them are fixed by adding condition variable signal/waits. Most of them are in fact fixed by adding thread-join operations.

Data privatization Another fix strategy automated by recent research is data privatization [37]. Previous technique targets two types of AV bugs, where the atomicity of read-after-write (RAW) accesses or read-after-read (RAR) accesses can be violated. Its patch creates a temporary variable to buffer the result of an earlier write access (in case of RAW) or read access (in case of RAR) to or from shared variables, and let a later read from the same thread to directly access this temporary variable, instead of the shared variable.

Our empirical study shows that data privatization is indeed a common fix strategy for AV bugs in practice, taken by developers to fix 8 out of 48 AV bugs in our study.

However, our study also found that the data privatization scheme used by developers goes beyond what used by existing research. First, some write-after-write (WAW) atomicity violations are also fixed by data privatization by developers. For example, Mozilla-52111 and Mozilla-201134 are both fixed by making the first write outputs to a temporary local variable, so that an interleaving read will not see the intermediate value. Second, in several cases, the bugs are fixed not by introducing a temporary local variable, but by changing the declaration of the original shared variable to make it a local variable. For example, in MySQL-7209, Mozilla-253786 and MySQL-142651, developers realize there are in fact no need to make the buggy variables shared. Only 3 bugs are fixed by developers following exactly the same way as existing research proposes.

3.2.4 How about the Future?

Our study points out directions for future research in automated concurrency-bug fixing. Specifically, future work can further refine existing auto-fix strategies, such as data privatization, following our study above; future research can also try to automate manual fix strategies that have not been well explored before, which we will discuss below.

Automating Add$_{\mathbf{join}}$ for OV bugs Although many recent research tools apply Add$_S$ to fix concurrency bugs [40, 42, 58, 60], they only add lock-related synchronization into software, including locks and condition variables. This is particularly problematic for OV bugs, as many manual OV patches are unrelated to locks or condition variables. Our work along this direction will be presented in Chapter 6.

Automating Move$_S$ for concurrency bugs Move$_S$ leverages existing synchronization in software to fix concurrency bugs. It is one of the most common manual fix strategies for both AV (6 out of 48) and OV bugs (14 out of 29). Unfortunately, it has never been automated by previous research to fix real-world bugs in large applications. Our work along this direction will be presented in Chapter 6.

Semantic changing fix for concurrency bugs *Bypass* and *Tolerance* are two intriguing concurrency-bug fix strategies, as they change the sequential semantics and were never

	Patch Location					Patch Structure		
	AV$_c$	AV$_r$	OV$_A$	OV$_B$	Misc	Skip	UnSkip	Misc
B	2	11	1	0	0	13	0	1
T	2	3	1	3	1	6	4	0

Table 3.4: Semantic-changing patches (B: Bypass strategy; T: Tolerance strategy).

explored by previous research. They are common enough to deserve attention – together, they are chosen for 24 out of 77 real-world bugs in our study. Their patches are often simple, mostly between 1–3 lines of code changes.

Our in-depth study shows that these patches are not ad-hoc. Instead, they follow common patterns that can be leveraged by automated tools, as shown in Table 3.2.4.

First, the patch location is almost always around key operations in the bug report, as shown in Table 3.2.4.

Second, the patch structure is mostly simple. Naturally, all bypass patches add condition checks to bypass code. Interestingly, almost all tolerance patches are also about control flow changes. Some add condition checks to bypass failure-inducing operations, such as a NULL-pointer dereference, after the unsynchronized accesses. Others change existing condition checking, so that some code that was originally skipped under the unsynchronized accesses would now get executed under the patch.

3.3 Conclusion

This chapter presented an in-depth study about timing-bug incidents in production-run cloud service systems, and timing bugs in multi-threaded systems. The findings revealed in our study provide motivation and guidance to future research in tackling timing bugs in cloud distributed systems and multi-threaded systems and improving the software reliability.

CHAPTER 4
DCATCH: AUTOMATICALLY DETECTING MESSAGE TIMING BUGS IN CLOUD SYSTEMS

In big data and cloud computing era, reliability of distributed systems is extremely important. Unfortunately, message timing bugs widely exist. They hide in the large state space of distributed cloud systems and manifest non-deterministically depending on the timing of distributed computation and communication. Due to their new non-determinism, message interleaving, which goes beyond traditional thread interleaving in multi-threaded systems, existing timing bug models and detection tools for multi-threaded concurrency bugs cannot precisely capture and accurately predict them. Effective models and techniques to detect message timing bugs are desired.

This chapter presents a pilot solution, DCatch, in the world of DCbug detection. DCatch predicts message timing bugs by analyzing correct execution of distributed systems. To build DCatch, we design a set of happens-before rules that model a wide variety of communication and concurrency mechanisms in real-world distributed cloud systems. We then build runtime tracing and trace analysis tools to effectively identify concurrent conflicting memory accesses in these systems. Finally, we design tools to help prune false positives and trigger message timing bugs.

We have evaluated DCatch on four representative open-source distributed cloud systems, Cassandra, Hadoop MapReduce, HBase, and ZooKeeper. By monitoring correct execution of seven workloads on these systems, DCatch reports 32 message timing bugs, with 20 of them being truly harmful.

4.1 Introduction

4.1.1 Motivation

Message timing bugs are non-deterministic and hide in the huge state space of a distributed system spreading across multiple nodes. They are difficult to avoid, detect, and debug.

There are only a few sets of approaches that tackle message timing bugs, to the best of our knowledge: software model checking, verification, verifiable language, record and replay debugging. Although this set of techniques are powerful, they suffer from inherent limitations. Distributed system model checkers [33, 46, 50, 87, 96] face state-space explosion problems, making them difficult to scale for many large real-world systems. Verification approaches [35, 91] require thousands of lines of proof to be written for every protocol. Verifiable language [22] is not deployed, as low-level imperative languages are still popular for performance reasons. Record and replay techniques [61] cannot help discover bugs until software has failed and are not yet effective for debugging message timing bugs due to the huge number of timing-related events in distributed systems.

In comparison, there is one approach that has been widely studied for combating LCbugs in single-machine software but has yet been explored for DCbugs — *dynamic bug detection* [24, 36, 44, 65, 66, 82]. In a nutshell, dynamic bug-detection techniques monitor and analyze memory accesses and synchronization operations, and identify conflicting and concurrent memory accesses as LCbug suspects. *Conflicting* means that multiple accesses are touching the same memory location with at least one write access. *Concurrent* means that there is no *happens-before* causality relationship between accesses, and hence accesses can happen one right after the other in any order [48]. These techniques do not guarantee finding all bugs and often report many false positives. However, they can usually work directly on large existing real-world systems implemented in popular languages, without much annotation or code changes from developers.

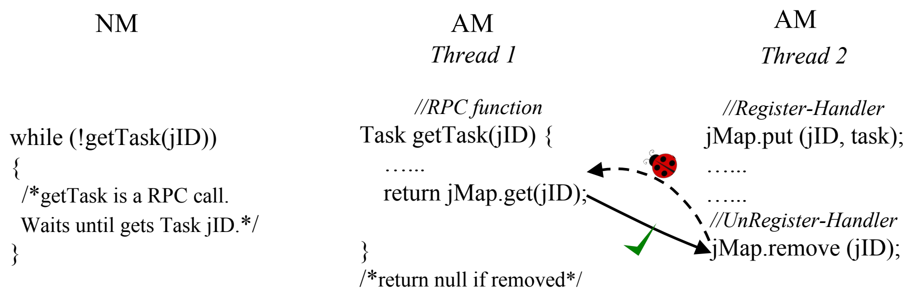


Figure 4.1: Root cause of the message timing bug shown in Figure 1.2

Despite its benefits, bug-detection approach has not permeated the literature of combating DCbugs. Thus, in this chapter, we present one the first attempts in building DCbug-detection tool to predict message timing bugs for distributed systems.

Our attempt of building a message timing bug detection tool is guided by our following understanding of message timing bugs.

Opportunities Message timing bugs have fundamentally similar root causes as LCbugs: unexpected timing among concurrent conflicting accesses to the *same* memory location inside *one* machine. Take the message timing bug in Figure 1.2 as an example. Although its triggering and error propagation involve communication among multiple nodes, its root cause is that event handler `UnRegister` could delete the `jID`-entry of `jMap` concurrently with a Remote Procedure Call (RPC) `getTask` reading the same entry, which is unexpected by developers (Figure 4.1).

This similarity provides opportunities for message timing bug detection to re-use the theoretical foundation (i.e., happens-before ordering) and work flow of LCbug detection. That is, we can abstract the causality relationship in distributed systems into a few happens-before (HB) rules; we can then follow these rules to build an HB graph representing the timing relationship among all memory accesses; finally, we can identify all pairs of concurrent conflicting memory accesse operations based on this HB graph and treat them as message timing bug candidates.

Challenges Message timing bugs and distributed systems also differ from LCbugs and single-machine systems in several aspects, which raise several challenges to message timing bug detection.

1. More complicated timing relationship: Although root-cause memory accesses of message timing bugs are inside one machine, reasoning about their timing relationship is complicated. Within each distributed system, concurrent accesses are conducted not only at thread level but also node level and event level, using a diverse set of communication and synchronization mechanisms like RPCs, queues, and many more (exemplified by Figure 4.1). Across different systems, there are different choices of communication and synchronization mechanisms, which are not always standardized. *Thus, designing HB rules for real-world distributed systems is not trivial. Wrong or incomplete HB modeling would significantly reduce the accuracy and the coverage of message timing bug detection.*

2. Larger scales of systems and bugs: Distributed systems naturally run at a larger scale than single-machine systems, containing more nodes and collectively more dynamic memory accesses. Message timing bugs also operate at a larger scale than LCbugs. For example, the message timing bug shown in Figure 1.2 involves three nodes (client, AM, and NM) in its triggering and error propagation. *The larger system scale poses scalability challenges to identify message timing bugs among huge numbers of memory accesses; the larger bug scale also demands new techniques in bug impact analysis and bug exposing.*

3. More subtle fault tolerance: Distributed systems contain inherent redundancy and aim to tolerate component failures. Their fault-tolerance design sometimes cures intermediate errors and sometimes amplifies errors, making it difficult to judge what are truly harmful bugs. For example, in Figure 4.1, the `jMap.get(jID)` in Thread-1 actually executes concurrently with two conflicting accesses in Thread-2: `jMap.put(jID,task)` from the `Register` handler, and `jMap.remove(jID)` from the `UnRegister` handler. The former is *not* a bug, due to the re-try while loop in NM; the latter is indeed a bug, as it causes the re-try while loop in NM to hang. *Thus, the subtle fault tolerance features pose challenges in maintaining accuracy of message timing bug detection.*

4.1.2 Contributions

Guided by the above opportunities and challenges, we built DCatch, to the best of our knowledge, a pilot solution in the world of message timing bug detection. The design of DCatch contains two important stages: (a) design the HB model for distributed systems and (b) design DCatch tool components.

HB Model: First, we build an HB model on which DCatch will operate, based on our study of representative open-source distributed cloud systems. This HB model is composed of a set of HB rules that cover inter-node communication, intra-node asynchronous event processing, and intra-node multi-threaded computation and synchronization. The details will be discussed in Chapter 4.2.

DCatch tool components: Next we build DCatch, our message timing bug detection tool. Although it follows the standard work flow of many LCbug detectors, our contribution includes customizing each step to address unique challenges for message timing bugs.

1. Run-time tracer traces memory accesses, event operations, inter-node RPCs, socket communication, and others as the system runs. The scope and granularity of this component is carefully designed to focus on inter-node communication and computation, which helps us to address the large-scale challenge in message timing bug detection and make DCatch scale to large real-world distributed cloud systems (Chapter 4.3.1).

2. Offline trace analysis processes run-time traces to construct an HB graph for all recorded memory accesses and reports all pairs of concurrent conflicting accesses as message timing bug candidates. Our contribution is the implementation of DCatch HB model for real-world distributed systems (Chapter 4.3.2).

3. Static pruning analyzes the program to figure out what might be the local and distributed impact of a message timing bug candidate. It estimates which message timing bug candidates are unlikely to cause failures, avoiding excessive false positives (Chapter 4.4).

4. Triggering re-runs the system and manipulates the timing of distributed execution according to the bug report, while considering the diverse concurrency and communication mechanisms in distributed systems. It helps trigger true bugs and further prunes false positives (Chapter 4.5).

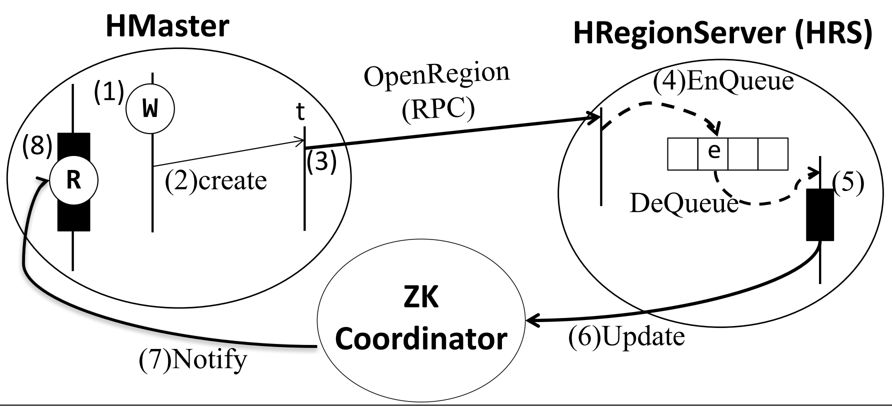
We evaluated DCatch on 4 varying real-world distributed systems, Cassandra, HBase, Hadoop MapReduce, and ZooKeeper. We tested 7 different workloads in total on these systems. Users have reported timing-related failures under these workloads. DCatch reports 32 message timing bugs. With the help of DCatch triggering component, we confirmed that 20 out of these 32 message timing bugs are indeed harmful: 12 of them explain the 7 failures we were aware of and the remaining 8 could lead to other failures we were unaware of. The detailed experimental results are presented in Chapter 4.7.

4.2 DCatch Happens-Before (HB) Model

Goals & Challenges Timing relationship is complicated in distributed systems. For example, to understand the timing between R and W in Figure 4.2, we need to consider thread (step 2 in Figure 4.2), RPC (step 3), event handling (step 4 & 5), ZooKeeper synchronization service (step 6 & 7), etc. Missing any of these steps will cause R and W to be incorrectly identified as concurrent with each other.

Our goal here is to abstract a set of HB rules by studying representative distributed cloud systems. Every rule R represents one type of causality relationship between a pair of operations[1], denoted as $o_1 \overset{R}{\Rightarrow} o_2$. These rules altogether allow reasoning about the timing between any two operations: if a set of HB rules chain o_1 and o_2 together $o_1 \overset{R^1}{\Longrightarrow} oo_1 \overset{R^2}{\Longrightarrow}$

1. An operation could be a memory access, a thread creation, etc.



W: `regionsToOpen.add(region)`
R: `if(regionsToOpen.isEmpty())`
(After this **R**, it has another **W** `(regionsToOpen.remove(region))`

(1) HMaster adds a region to `regionsToOpen` list (i.e., the W)
(2) A thread `t` is created to open the region
(3) `t` invokes an RPC call `OpenRegion`
(4) The RPC implementation in HRS puts a region-open event `e` into a queue
(5) `e` is handled
(6) A request is sent to ZooKeeper to update the corresponding region status
(7) ZooKeeper sends a notification to HMaster about this state change
(8) In the notification handling, HMaster reads `regionsToOpen` (i.e., the R)

Figure 4.2: A wide variety of causality relationships in HBase.

App	Inter-Node			Intra-Node	
	Synchronous RPC	Asynchronous Socket	Custom Protocol	Synchronous Threads	Asynchronous Events
Cassandra	-	✓	-	✓	✓
HBase	✓	-	✓	✓	✓
MapReduce	✓	-	✓	✓	✓
ZooKeeper	-	✓	-	✓	✓

Table 4.1: Concurrency and Communication in Distributed Systems

$oo_2...oo_{k-1} \overset{R^k}{\Longrightarrow} o_2$, o_1 must happen before o_2, denoted as $o_1 \Rightarrow o_2$. If neither $o_1 \Rightarrow o_2$ nor $o_2 \Rightarrow o_1$ holds, they are concurrent and hence can execute side by side in any order. This set of HB rules need to be comprehensive and precise in order for DCatch to achieve good bug detection accuracy and coverage.

Why do we need a new model? On one hand, HB models were thoroughly studied for single-machine systems, including both multi-threaded software [72] and event-driven applications [36, 69, 79]. However, these models do not contain all causality relationships in distributed systems, and may contain causality relationships not held in distributed systems. On the other hand, distributed-system debugging tools [68, 86] proposed meta-data propagation techniques to track coarse-granularity causality relationship between *user-specified* operations. However, without a formal HB model, they are unsuitable for message timing bug detection, where fine-granularity causality relationship needs to be computed among a huge number of memory accesses.

Below we present the concurrency and communication mechanisms that we learn from studying representative distributed systems (Table 4.1), from which we abstract HB rules.

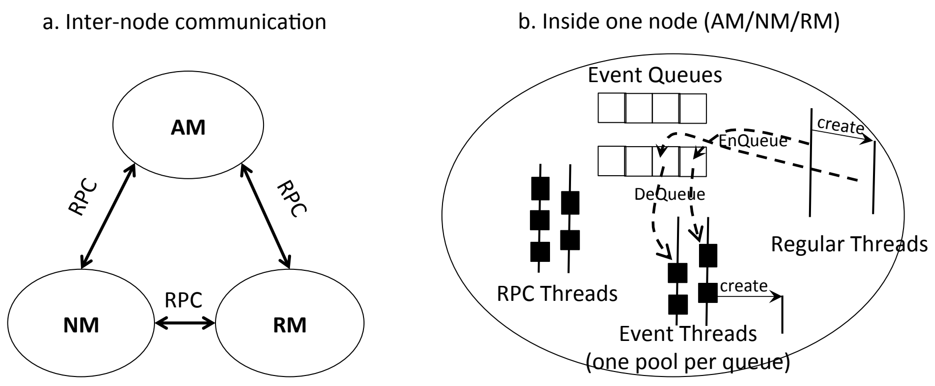


Figure 4.3: Concurrency and communication in MapReduce

4.2.1 Inter-node Concurrency and Communication

Every distributed system involves multiple parallel-executing nodes that communicate with each other through messages (Figure 4.3a). We abstract message-related HB rules, short as **Rule-M**, based on different communication patterns.

Synchronous RPC A thread in node n_1 could call an RPC function r implemented by node n_2, like step (3) in Figure 4.2. This thread will block until n_2 sends back the RPC result. RPC communication implies the following HB rules: making an RPC call r on n_1, denoted as *Create (r, n_1)*, happens before the beginning of the RPC execution on n_2, *Begin (r, n_2)*; the end of the RPC execution, *End (r, n_2)*, happens before the return from the RPC call r on n_1, *Join (r, n_1)*.

Rule-M$^{\mathbf{rpc}}$: $\textit{Create}\ (r, n_1) \xRightarrow{M^{\text{rpc}}} \textit{Begin}\ (r, n_2)$;

$\textit{End}\ (r, n_2) \xRightarrow{M^{\text{rpc}}} \textit{Join}\ (r, n_1)$.

Asynchronous Socket A thread in node n_1 sends a message m to node n_2 through network sockets. Unlike RPC, the sender does not block and continues its execution. The sending happens before the receiving.

Rule-M$^{\mathbf{soc}}$: $\textit{Send}\ (m, n_1) \xRightarrow{M^{\text{soc}}} \textit{Recv}\ (m, n_2)$.

In addition to the above two basic communication mechanisms, we also found the following common custom synchronization protocols, implemented using a combination of RPC/-socket communication and intra-node computation.

Custom Push-Based Synchronization Protocol Node n_1 updates a status s to a dedicated coordination node n_c, and n_c notifies all subscribed nodes, such as n_2, about this update. The update of s by n_1, *Update (s,n_1)*, happens before the notification delivered at n_2, *Pushed (s, n_2)*. For example, HBase nodes sometimes communicate through ZooKeeper: one node registers a *zknode* with a specific path in ZooKeeper; ZooKeeper will then notify this node of all changes to this *zknode* from other nodes, like steps (6) and (7) in Figure 4.2.

Rule-M$^{\mathbf{push}}$: *Update* $(s, n_1) \overset{M^{\text{push}}}{\Longrightarrow}$ *Pushed* (s, n_2).

Note that, this rule is **not** redundant given Rule-M$^{\text{rpc}}$ and Rule-M$^{\text{soc}}$. We can decompose this rule into three chains of causality relationship: (1) *Update*$(s, n_1) \Rightarrow$ *Recv*(s, n_c); (2) *Recv*$(s, n_c) \Rightarrow$ *Send*(s, n_c); (3) *Send*$(s, n_c) \Rightarrow$ *Pushed*(s, n_2). Chain (2) is very difficult to figure out, as it involves complicated intra-node computation and synchronization in n_c, which guarantees that every node interested in s gets a notification. Even for chain (1) and (3), there is no guarantee that Rule-M$^{\text{rpc}}$ and Rule-M$^{\text{soc}}$ can figure them out, because the communication between n_1/n_2 and n_c often contains more than one RPC or socket message.

Custom Pull-Based Synchronization Protocol Node n_2 keeps polling n_1 about status s in n_1, and does not proceed until it learns that s has been updated to a specific value. Clearly, the update of s in n_1 happens before the use of this status on n_2.

Rule-M$^{\mathbf{pull}}$: *Update*$(s, n_1) \overset{M^{\text{pull}}}{\Longrightarrow}$ *Pulled*(s, n_2).

This is similar with the distributed version of the while-loop custom synchronization in single-machine systems [89, 93]. Figure 4.1 shows an example of this HB relationship: `jMap.put (jID, task)` in AM happens before the exit of the `while`-loop in NM.

This rule is not redundant given other rules due to complicated semantics inside n_1: traditional HB rules cannot establish the causality between s being set and s being read by an RPC function or being serialized into a socket message.

4.2.2 Intra-node Concurrency and Communication

Synchronous multi-threaded concurrency Within each node, there are multiple processes and threads, as shown in Figure 4.3b[2]. The rules here are about classic fork/join causality: the creation of a thread t (or process) in the parent thread, denoted as *Create(t)*, happens before the execution of t starts, *Begin (t)*. The end of t's execution, *End (t)*, happens before a successful join of t *Join (t)*.

Rule-T$^{\text{fork}}$: $\mathit{Create}\ (t) \overset{T^{\text{fork}}}{\Longrightarrow} \mathit{Begin}\ (t)$.

Rule-T$^{\text{join}}$: $\mathit{End}\ (t) \overset{T^{\text{join}}}{\Longrightarrow} \mathit{Join}\ (t)$.

Asynchronous event-driven concurrency All the systems in Table 4.1 conduct asynchronous event-driven processing, like steps (4)(5) in Figure 4.2, essentially creating concurrency inside a thread. Events could be enqueued by any thread, and then processed by pre-defined handlers in event-handling thread(s). The enqueue of an event e, *Create* (e), happens before the handler-function of e starts, denoted as *Begin* (e).

Rule-E$^{\text{enq}}$: $\mathit{Create}\ (e) \overset{E^{\text{enq}}}{\Longrightarrow} \mathit{Begin}\ (e)$.

For two events e_1 and e_2 from the same queue, the timing between their handling depends on several properties of the queue. For all the systems that we have studied, all the queues are FIFO and every queue has only one dispatching thread, one or multiple handling threads. Consequently, the handling of e_1 and e_2 is serialized when their queue is equipped with only one handling thread, and is concurrent otherwise. We refer to the former type of queues as *single-consumer queues.* All the queues in ZooKeeper and some queues in MapReduce are single-consumer queues.

Rule E$^{\text{serial}}$: $\mathit{End}\ (e_1) \overset{E^{\text{serial}}}{\Longrightarrow} \mathit{Begin}\ (e_2)$, if $\mathit{Create}\ (e_1) \Rightarrow \mathit{Create}\ (e_2)$; $e_1, e_2 \in Q$; Q

2. MapReduce contains multiple processes in one node not shown in figure.

is single-consumer FIFO queue.

Previous work has built HB rules for single-machine event-driven applications, particularly Android apps [36, 69, 79]. In comparison, some complicated queues (e.g., non-FIFO queues) and corresponding rules observed by previous work have not been observed in these distributed systems.

Sequential program ordering According to the classical HB model [48], the execution order within one thread is deterministic and hence has the following rule.

Rule $\mathbf{P^{reg}}$: $o_1 \xRightarrow{P^{\text{reg}}} o_2$, if o_1 occurs before o_2 during the execution of a regular thread.

We need to revise this rule for threads that are involved in asynchronous computing. Specifically, for two operations inside an event/RPC/message handling thread, sequential program ordering exists between them only when they belong to the same event/RPC/message handler function.

Rule $\mathbf{P^{nreg}}$: $o_1 \xRightarrow{P^{\text{nreg}}} o_2$, if o_1 occurs before o_2 during the execution of an event handler, a message handler, or an RPC function.

4.2.3 Summary

The above **MTEP** rules constitute the DCatch HB model. Our evaluation will show that every rule is crucial to the accuracy and coverage of message timing bug detection (Chapter 4.7.4). For the real-world example demonstrated in Figure 4.2, we can now infer $W \Rightarrow R$, because of the following chains of happens-before relationship: $W \xRightarrow{P^{\text{reg}}}$ *Create (t)* $\xRightarrow{T^{\text{fork}}}$ *Begin (t)* $\xRightarrow{P^{\text{reg}}}$ *Create (OpenRegion, HMaster)* $\xRightarrow{M^{\text{rpc}}}$ *Begin (OpenRegion, HRS)* $\xRightarrow{P^{\text{nreg}}}$*Create (e)* $\xRightarrow{E^{\text{enq}}}$*Begin (e)* $\xRightarrow{P^{\text{nreg}}}$*Update (RS...OPENED, HRS)* $\xRightarrow{M^{\text{push}}}$*Pushed (RS...OPENED, HMaster)* $\xRightarrow{P^{\text{nreg}}}$ *R.*

Note that, our model is not the only viable HB model for distributed systems. Our model abstracts away some low-level details in RPC and event libraries. For example, incoming

RPC calls are first put into queue(s) before assigned to RPC threads, but our Rule-M$^{\text{rpc}}$ abstracts away these queues inside RPC library; between the enqueue of an event and the beginning of the event handling, a dedicated thread would conduct event dispatching, which is also abstracted away in our Rule-E$^{\text{enq}}$.

Our model also intentionally ignores certain causality relationships that do not affect our message timing bug detection. For example, our model does not consider condition-variable notify-and-wait causality relationship, because it is almost never used in the inter-node communication and computation part of our studied distributed systems; we do not consider lock synchronization in this model, because lock provides mutual exclusions not strict ordering.

Our model could also miss some custom synchronization protocols in distributed systems. Next few chapters will describe the design of the four components of DCatch based on the model defined above.

4.3 DCatch Tracing and Trace Analysis

Given our HB model, we began building the DCatch tool. As first steps, we need to (1) trace the necessary operations and (2) build the HB graph and perform analysis on top. Below we describe how these work and how we address tracing and analysis challenges such as reducing memory access traces and applying the MTEP rules correctly.

4.3.1 DCatch Tracing

DCatch produces a trace file for every thread of a target distributed system at run time. These traces will then allow trace analyzer to identify message timing bug candidates. The detailed implementation is based on WALA, a static Java bytecode analysis framework, and Javassist, a dynamic Java bytecode transformation framework; more implementation details are discussed in Chapter 4.6.

Which operations to trace?

Memory-access tracing Naively, we want to record all accesses to program variables that could potentially be shared among threads or event handlers. However, this exhaustive approach would lead to huge traces that are expensive or even cannot be processed for many real-world distributed system workloads as we will see in Chapter 4.7.4.

Fortunately, such excessive logging is unnecessary for DCbug detection. DCbugs are triggered by inter-node interaction, with the root-cause memory accesses in code regions related to inter-node communication and corresponding computation, not everywhere.

Following this design principle, DCatch traces all accesses to heap objects and static variables in the following three types of functions and their callees: (1) RPC functions; (2) functions that conduct socket operations; and (3) event-handler functions. The third type is considered because they conduct many pre- and post-processing of socket sending/receiving and RPC calls.

HB-related operation tracing DCatch traces operations that allow its trace analysis to apply the **MTEP** rules, as shown in Table 4.2. DCatch automatically identifies these operations at run time using the Javassist infrastructure. The implementation details are discussed in Chapter 4.6.

For push-based synchronization, the current prototype of DCatch focuses on the synchronization service provided by ZooKeeper, as discussed in Chapter 4.2.1. DCatch traces ZooKeeper APIs `ZooKeeper::create`, `ZooKeeper::delete`, and `ZooKeeper::setData` as *Update* operations, and ZooKeeper `Watcher` events with event types `NoteCreated`, `NodeDeleted`, and `NodeDataChanged` as *Push* operations. The parameters, event types, and timestamps help DCatch trace analysis to group corresponding *Update* and *Push* together. For pull-based synchronization, the *Update* and *Pull* operations involve memory accesses, RPC calls, and loops, which are already traced. We will explain how to put them together to construct

	M-Rule	T-Rule	E-Rule	P-Rule
Creat (t), Join (t)		✓		
Begin (t), End (t)		✓		
Begin (e), End (e)			✓	✓
Create (e)			✓	
Begin (r, n_2), End (r, n_2)	✓			✓
Create (r, n_1), Join (r, n_1)	✓			
Send (m, n_1)	✓			
Recv (m, n_2)	✓			✓
Update (s, n_1)	✓			
Pushed (s, n_2)	✓			✓
Pull (s, n_2)	✓			

Table 4.2: HB-related Tracing in DCatch

pull-based HB relationship in Chapter 4.3.2.

Other tracing DCatch does not need to trace lock and unlock operations to *detect* message timing bugs, because lock and unlock operations are not part of the DCatch HB model. However, as we will see in Chapter 4.5.2, DCatch needs to know about lock/unlock operations to *trigger* some message timing bug candidates. Such information sometimes can help avoid hangs when DCatch tries to manipulate the timing and *trigger* a message timing bug candidate. Therefore, DCatch also traces lock and unlock operations, including both implicit lock operations (i.e., `synchronized` methods and `synchronized` statements) and explicit lock operations.

What to record for each traced operation?

Each trace record contains three pieces of information: (1) type of the recorded operation; (2) callstack of the recorded operation; and (3) ID. The first two are straightforward. The IDs help DCatch trace analyzer to find related trace records.

For a memory access, ID uniquely identifies the accessed variable or object. The ID of an object field is the field-offset and the object hashcode. The ID of a static variable is the variable name and its corresponding namespace.

For HB-related operations, the IDs will allow DCatch trace analysis to correctly apply HB rules. For every thread- or event- related operation, the ID is the object hashcode of the corresponding thread or event object. For each RPC-related and socket-related operation, DCatch tags each RPC call and each socket message with a random number generated at run time (details in Chapter 4.6).

For lock/unlock operations, the IDs uniquely identify the lock objects, allowing DCatch's triggering module to identify all lock critical sections and perturb the timing at appropriate places (details in Chapter 4.5.2).

4.3.2 DCatch Trace Analysis

DCatch trace analyzer identifies every pair of memory accesses (s, t), where s and t access the same variable with at least one write and are concurrent with each other (i.e., no HB-relationship between them), and considers (s, t) as a message timing bug candidate.

HB-graph construction

An HB graph is a DAG graph. Every vertex v represents an operation $o(v)$ recorded in DCatch trace, including both memory accesses and HB-related operations. The edges in the graph are arranged in a way that v_1 can reach v_2 if and only if $o(v_1)$ happens before $o(v_2)$.

To build such a graph, DCatch first goes through all trace files collected from all threads of all processes in all nodes, and makes every record a vertex in the graph.

Next, DCatch adds edges following our MTEP rules. We discuss how to apply **Rule $\mathbf{E}^{\mathbf{serial}}$** and **Rule $\mathbf{M}^{\mathbf{pull}}$** below. We omit the details of applying other rules, as they are straightforward and can be applied in any order — the ID of each trace record allows DCatch to easily group related operations.

DCatch applies **Rule $\mathbf{E}^{\mathbf{serial}}$** as the last HB rule. For every thread that handles a single-consumer event queue, DCatch checks every pair of *End (e_i)* and *Begin (e_j)* recorded in its trace, and adds an edge from the former to the latter, if DCatch finds *Create* $(e_i) \Rightarrow$ *Create* (e_j) based on those HB edges already added so far. DCatch repeats this step until reaching a fixed point.

Applying **Rule M$^{\mathbf{pull}}$** requires program analysis. The algorithm here is inspired by how loop-based custom synchronization is handled in LCbug detection [89, 93]. For every pair of conflicting concurrent read and write $\{r, w\}$, we consider r to be potentially part of a pull-based synchronization protocol if (1) r is executed inside an RPC function; (2) the return value of of this RPC function depends on r; (3) in another node that requests this RPC, the return value of this RPC is part of the exit condition of a loop l. We will then run the targeted software again, tracing only such rs and all writes that touch the same object based on the original trace. The new trace will tell us which write $w*$ provides value for the last instance of r before l exits. If $w*$ and r are from different threads, we will then conclude that $w*$ in one node happens before the exit of the remote loop l in another node. Due to space constraints, we omit the analysis details here. This part of the analysis is done together with intra-node while-loop synchronization analysis. Although requiring running the software for a second time, it incurs little tracing or trace analysis overhead, because it focuses on loop-related memory accesses.

Message timing bug candidate report

The HB graph is huge, containing thousands to millions of vertices in our experiments. Naively computing and comparing the vector-timestamps of every pair of vertices would be too slow. Note that each vector time-stamp will have a huge number of dimensions, with each event handler and RPC function contributing one dimension.

To speed up this analysis, DCatch uses the algorithm proposed by previous asynchronous race detection work [79]. The algorithm computes a reachable set for every vertex in HB graph, and then turns HB-relationship checking into a constant-time array lookup.

4.4 Static Pruning

Not all message timing bug candidates reported by trace analysis can cause failures. This is particularly true in distributed systems, which inherently contain more redundancy and failure tolerance than single-machine systems. The high-level idea of pruning false positives by estimating failure impacts has been used by previous LCbug detection tools [101, 102]. However, previous work only analyzes intra-procedural failure impacts. Thus, the challenge is to conduct inter-procedural *and* inter-node impact analysis to better suit the failure-propagation nature of message timing bugs in distributed systems.

To avoid excessive false positives, we first configure DCatch to treat certain instructions in software as *failure instructions*, which represent the (potential) occurrence of severe failures. Then, given a bug candidate (s, t), DCatch statically analyzes related Java bytecode of the target system to see if s or t may have local (i.e., within one node) or distributed (i.e., beyond one node) impact towards the execution of any failure instruction identified above.

4.4.1 Identifying Failure Instructions

The current prototype of DCatch considers the following failures and identifies *failure instructions* accordingly: (1) system aborts and exits, whose corresponding failure instructions are invocations of abort and exit functions (e.g., `System.exit`); (2) severe errors that are printed out, whose corresponding failure instructions are invocations of `Log::fatal` and `Log::error` functions in studied systems; (3) throwing uncatchable exceptions, such as `RuntimeException`; (4) infinite loops, where we consider every loop-exit instruction as a potential failure instruction. Finally, if a failure instruction is inside a `catch` block, we also consider the corresponding exception throw instruction, if available, as a failure instruction. This list is configurable, allowing future DCatch extension to detect message timing bugs with different failures.

4.4.2 Impact Estimation

For a message timing bug candidate (s, t), if DCatch fails to find any failure impact for s and t through the analysis described below, this message timing bug candidate will be pruned out from the DCatch bug list. All the implementation below is done in WALA code analysis framework, leveraging WALA APIs that build program dependency graphs.

Local impact analysis We conduct both intra-procedural and inter-procedural analysis for local impact analysis. Given a memory-access statement s located in method M, we first check whether any failure instruction in M has control- or data- dependence on s. We apply similar checking for t.

We then check whether s could affect failure instructions inside the callers of M through either the return value of M or heap/global objects. For the latter, DCatch only applies the analysis to one-level caller of M, not further up the call chain for accuracy concerns. Note that, since DCatch tracer and trace analysis report call-stack information, our inter-procedural analysis follows the reported call-stack of s. Finally, we check whether s could affect failure sites in the callee functions of M through either function-call parameters or heap/global variables. This analysis is also only applied to the one-level callee of M. We skip our algorithm details due to space constraints.

Distributed impact analysis As shown in Figure 4.1, an access in one node could lead to a failure in a different node. Therefore, DCatch also analyzes RPC functions to understand the remote impact of a memory access.

Specifically, if we find an RPC function R along the callstack of the memory access s, we check whether the return value of R depends on s. If so, we then locate the function M_r on a different node that invokes the RPC call R. Inside M_r, we check whether any failure instruction depends on the return value of R. Note that locating M_r is straightforward given the HB chains already established by DCatch trace analysis.

DCatch does not analyze inter-node impact through sockets, as socket communication is not as structured as RPCs.

4.5 Message Timing Bugs Triggering and Validation

A DCatch bug report (s, t) still may not be harmful for two reasons. First, s and t may not be truly concurrent with each other due to custom synchronization unidentified by DCatch. Second, the concurrent execution of s and t may not lead to any failures, as the impact analysis conducted in Chapter 4.4 only provides a static estimation. Furthermore, even for those truly harmful message timing bug candidates, triggering them could be very challenging in distributed systems.

To address this, we do not stop with just reporting potential message timing bugs, but rather we also build this last component of DCatch to help assess message timing bug reports and reliably expose truly harmful message timing bugs, hence an end-to-end analysis-to-testing tool. This phase includes two parts: (1) an infrastructure that enables easy timing manipulation in distributed systems; and (2) an analysis tool that suggests how to use the infrastructure to trigger a message timing bug candidate. These two features are unique to triggering message timing bugs.

4.5.1 Enable Timing Manipulation

Naively, we could perturb the execution timing by inserting `sleep` into the program, like how LCbugs are triggered in some previous work [78]. However, this naive approach does not work for complicated bugs in complicated systems, because it is hard to know how long the `sleep` needs to be. More sophisticated LCbug exposing approach [71, 84] runs the whole program in one core and controls the timing through thread scheduler. This approach does not work for message timing bugs, which may require manipulating the timing among operations from different nodes, which are impractical to run on one core.

Our infrastructure includes two components: client-side APIs for sending coordination-request messages and a message-controller server (we refer to the distributed system under testing as client here).

Imagine we are given a pair of operations A and B, and we want to explore executing A right before B and also B right before A. We will simply put a `_request` API call before A and a `_confirm` API call right after A, and the same for B. At run time, the `_request` API will send a message to the controller server to ask for the permission to continue execution. At the controller side, it will wait for the request-message to arrive from both parties, and then grant the permission to one party, wait for the confirm-message sent by the `_confirm` API, and finally grant the permission for the remaining party. The controller will keep a record of what ordering has been explored and will re-start the system several times, until all ordering permutations among all the request parties (just two in this example) are explored.

4.5.2 Design Timing Manipulation Strategy

With the above infrastructure, the remaining question is where to put the `_request` and `_confirm` APIs given a message timing bug report (s, t). The `_confirm` APIs can be simply inserted right after the heap access in the bug report. Therefore, our discussion below focuses on the placement of `_request` APIs.

The naive solution is to put `_request` right before s and t. However, this naive approach may lead to hangs or too many `_request` messages sent to the controller server due to the huge number of dynamic instances of s or t. DCatch provides the following analysis to help solve this problem, both are unique to triggering message timing bugs.

First, DCatch warns about potential hangs caused by poor placements of `_request` in the following three cases and suggests non-hang placements. (1) If s and t are both inside event handlers and their event handlers correspond to a single-consumer queue, DCatch warns about hangs and suggests putting `_request` in corresponding event enqueue functions. (2)

If s and t are both inside RPC handlers and their RPC functions are executed by the same handling thread in the same node, DCatch suggests putting `_request` in corresponding RPC callers. (3) If s and t are inside critical sections guarded by the same lock, DCatch suggests putting `_request` right before the corresponding critical sections. DCatch gets the critical section information based on lock-related records in its trace, as discussed in Chapter 4.3.1.

Second, DCatch warns about large number of dynamic instances of s and t and suggest better placements. The message timing bug report will contain call-stacks for s and t. When DCatch checks the run-time trace and finds a large number of dynamic instances of the corresponding call-stack for s (same for t), DCatch will check its happens-before graph to find an operation o in a different node that causes s, and checks whether o is a better place for `_request`. This analysis is very effective: many event handlers and RPC functions are always executed under the same call stack, and hence could make bug triggering very complicated without this support from DCatch.

4.6 Implementation

DCatch is implemented using WALA v1.3.5 and Javassist v3.20.0 for a total of 12 KLOC. Below are more details.

HB-related operation tracing DCatch traces HB-related operations using Javassist, a dynamic Java bytecode re-writing tool, which allows us to analyze and instrument Java bytecode whenever a class is loaded.

All thread-related operations can be easily identified following the `java.lang.Thread` interface. Event handling is implemented using `org.apache.hadoop.yarn.event.EventHandler` and `org.apache.hadoop.hbase.executor.EventHandler` interface in Hadoop and HBase. The prototype of an handler function is `EventHandler::handle (Event e)`. Cassandra and ZooKeeper use their own event interfaces. The way handler functions are implemented and invoked are similar as that in Hadoop/HBase.

For RPC, HBase and early versions of Hadoop share the same RPC library interface, `VersionedProtocol`. All methods declared under classes instantiated from this interface are RPC functions, and hence can be easily identified. Later versions of Hadoop use a slightly different interface, `ProtoBase`, but the way to identify its RPC functions is similar.

For socket, Cassandra has a superclass `IVerbHandler` to handle socket communication and every message sending is conducted by `IVerbHandler::sendOneWay (Message, EndPoint)`. DCatch can easily identify all such function calls, as well as the message object. ZooKeeper uses a super-class `Record` for all socket messages. DCatch identifies socket sending and receiving based on how `Record` objects are used.

Memory access tracing DCatch first uses WALA, a static Java bytecode analysis framework, to statically analyze the target software, identifies all RPC/socket/event related functions, and stores the result. DCatch then uses Javassist to insert tracing functions before every heap access (`getfield/putfield` instruction) or static variable access (`getstatic/putstatic` instruction) in functions identified above.

Tagging RPC DCatch statically transforms the target software, adding one extra parameter for every RPC function and one extra field in socket-message object, and inserting the code to generate a random value for each such parameter/field at the invocation of every RPC/socket-sending function. DCatch tracing module will record this random number at both the sending side and the receiving side, allowing trace analysis to pair message sending and receiving together.

Portability of DCatch As described above, applying DCatch to a distributed software project would require the following information about that software: (1) what is the RPC interface; (2) what are socket messaging APIs; (3) what are event enqueue/handler APIs; (4)

BugID	LoC	Workload	Symptom	Error	Root
CA-1011	61K	startup	Data backup failure	DE	AV
HB-4539	188K	split table & alter table	System Master Crash	DE	OV
HB-4729	213K	enable table & expire server	System Master Crash	DE	AV
MR-3274	1,266K	startup + wordcount	Hang	DH	OV
MR-4637	1,388K	startup + wordcount	Job Master Crash	LE	OV
ZK-1144	102K	startup	Service unavailable	LH	OV
ZK-1270	110K	startup	Service unavailable	LH	OV

Table 4.3: DCatch Benchmark Bugs and Applications.

whether the event queues are FIFO and whether they have one or multiple handler threads.

In our experience, providing the above specifications is straightforward and reasonably easy, because we only need to identify a small number of (RPC/event/socket) interfaces or prototypes, instead of a large number of instance functions. We also believe that the above specifications are necessary for accurate message timing bug detection in existing distributed systems, just like specifying pthread functions for LCbug detection and specifying event related APIs for asynchronous-race detection.

4.7 Evaluation

4.7.1 Methodology

Benchmarks We evaluate DCatch on seven timing-related problems reported by real-world users in four widely used open-source distributed systems: Cassandra distributed key-value stores (CA); HBase distributed key-value stores (HB); Hadoop MapReduce distributed computing framework (MR); ZooKeeper distributed synchronization service (ZK). These systems range from about 61 thousand lines of code to more than three million lines of code, as shown in Table 4.3.

We obtain these benchmarks from TaxDC benchmark suite [51]. They are all triggered by untimely communication across nodes. As shown in Table 4.3, they cover all common types of failure symptoms: job-master node crash, system-master node crash, hang, etc. They cover different patterns of errors: local explicit error (LE), local hang (LH), distributed explicit error (DE), distributed hang (DH). Here, local means on the same machine as the root-cause memory accesses; distributed means on a different machine from the root-cause accesses. They also cover different root causes: order violations (OV) and atomicity violations (AV).

Experiment settings We use failure-triggering workloads described in the original user reports, as shown in Table 4.3. They are actually common workloads: system startups in Cassandra and ZooKeeper; alter a table and then split it in HBase; enable a table and then crash a region-server in HBase; run WordCount (or any MapReduce job) and kill the job before it finishes in MapReduce. Note that, due to the non-determinism of message timing bugs, failures rarely occur under these workload. DCatch detects message timing bugs by monitoring **correct** runs of these workload.

We run each node of a distributed system in one virtual machine, and run all VMs in one physical machine (M1), except for HB-4539, which requires two physical machines (M1 & M2). Both machines use Ubuntu 14.04 and JVM v1.7. M1 has Intel® Xeon® CPU E5-2620 and 64GB of RAM. M2 has Intel® Core$^{\text{TM}}$i7-3770 and 8GB of RAM. We connect M1 and M2 with Ethernet cable. All trace analysis and static pruning are on M1. After the static pruning in Chapter 4.4, all the remaining bug candidates are considered **DCatch bug reports**. We will then try to trigger each reported bug leveraging the DCatch triggering module. Note that, the triggering result does **not** change the count of DCatch bug reports.

Evaluation metrics We will evaluate the following aspects of DCatch: the coverage and accuracy of bug detection, and the overhead of bug detection, including run-time overhead, off-line analysis time, and log size. All the performance numbers are based on an average of 5 runs. We will also compare DCatch with a few alternative designs.

We will put a DCatch bug report (s, t) into one of three categories: if s and t are not

BugID	Detected?	#Static Ins. Pair			#CallStack Pair		
		Bug	Benign	Serial	**Bug**	Benign	Serial
CA-1011	✓	$\mathbf{3}_1$	0	0	$\mathbf{5}_1$	2	0
HB-4539	✓	$\mathbf{3}_3$	0	1	$\mathbf{3}_3$	0	1
HB-4729	✓	$\mathbf{4}_4$	1	0	$\mathbf{5}_5$	5	0
MR-3274	✓	$\mathbf{2}_1$	0	4	$\mathbf{2}_1$	0	9
MR-4637	✓	$\mathbf{1}_1$	2	4	$\mathbf{1}_1$	3	9
ZK-1144	✓	$\mathbf{5}_1$	1	1	$\mathbf{5}_1$	1	1
ZK-1270	✓	$\mathbf{6}_1$	2	0	$\mathbf{6}_1$	2	0
Total*		$\mathbf{20}_{12}$	5	7	$\mathbf{23}_{13}$	12	12

Table 4.4: DCatch Bug Detection Results

concurrent with each other, it is a *serial* report (i.e., not concurrent); if s and t are concurrent with each other, but their concurrent execution does not lead to failures, it is a *benign* bug; if their concurrent execution leads to failures, it is a true *bug*.

We report message timing bug counts by the unique number of static instruction pairs and the unique number of callstack pairs as shown in Table 4.4. Since these two numbers do not differ much, we use static-instruction count by default unless otherwise specified.

4.7.2 Bug Detection Results

Overall, DCatch has successfully detected message timing bugs for all benchmarks while monitoring *correct* execution of these applications, as shown by the ✓ in Table 4.4. In addition, DCatch found a few truly harmful message timing bugs we were unaware of and outside the TaxDC suite [51]. DCatch is also accurate: only about one third of all the 32 DCatch bug reports are false positives based on static count.

Harmful bug reports DCatch has found root-cause message timing bugs for every benchmark. In some cases, DCatch found multiple root-cause message timing bugs for one benchmark. For example, in HB-4729, users report that "clash between region unassign and splitting kills the master". DCatch found that one thread t_1 could delete a zknode concurrently with another thread t_2 reads this zknode and deletes this zknode. Consequently, multiple message timing bugs are reported here between delete and reads, and between delete and delete. They are all truly harmful bugs: any one of these zknode operations in t_2 would fail and cause HMaster to crash, if the delete from t_1 executes right before it.

DCatch also found a few harmful message timing bugs, 8 in static count and 10 in callstack count, that go beyond the 7 benchmarks. We were unaware of these bugs, and they are not part of the TaxDC bug suite. We have triggered all of them and observed their harmful impact, such as node crashes and unavailable services, through DCatch triggering module. We have carefully checked the change log of each software project, and found that two among these 8 message timing bugs have never been discovered or patched and the remaining have already been patched by developers in later versions.

Benign bug reports DCatch only reported few benign message timing bugs, 5 out of 32 across all benchmarks, benefiting from its static pruning module. In Cassandra, DCatch reports some message timing bugs that can indeed cause inconsistent meta-data across nodes. However, this inconsistency will soon get resolved by the next gossip message. Therefore, they are benign. Other benign reports are similar. Note that, for CA-1011, the benign report count is 0 in static count but 2 in callstack count, because the two benign reports share the same static identities with some truly harmful bug reports.

Serial bug reports DCatch HB model and HB analysis did well in identifying concurrent memory accesses. For only 7 out of 32 message timing bug reports, DCatch mistakenly reports two HB-ordered memory accesses as concurrent. Some of them are caused by unidentified RPC functions, which do not follow the regular prototype and hence are missed by our static analysis. Some of them are caused by custom synchronization related to address transfer [100]. The remaining are caused by distributed custom synchronization. For example, ZK has a function `waitForEpoch`, essentially a distributed barrier — accesses before

BugID	#Static Ins. Pair			#Callstack Pair		
	TA	TA+SP	TA+SP+LP	TA	TA+SP	TA+SP+LP
CA-1011	46	4	3	175	9	7
HB-4539	24	4	4	57	5	4
HB-4729	52	6	5	219	12	10
MR-3274	53	8	6	553	18	11
MR-4637	61	8	7	568	21	13
ZK-1144	29	8	7	52	8	7
ZK-1270	25	10	8	25	10	8

Table 4.5: # of message timing bugs reported by DCatch trace analysis

`waitForEpoch` in n_1 happens before accesses after corresponding `waitForEpoch` in n_2. The implementation of `waitForEpoch` is complicated and cannot be inferred by existing HB rules. Single-machine custom synchronization is an important research topic in LCbug detection [89, 93]. DCatch is just a starting point for research on distributed custom synchronization.

DCatch false-positive pruning As shown in Table 4.5, our static pruning pruned out a big portion of message timing bug candidates reported by DCatch trace analysis: less than 10% of message timing bug candidates (callstack count) remain after the static pruning for CA, HB, and MR benchmarks.

To evaluate the quality of static pruning, we randomly sampled and checked 35 message timing bug candidates that have been pruned out, 5 from each benchmark. We found that all of them are indeed false positives. A few of them would lead to exceptions, but the exceptions are well handled with only warning or debugging messages printed out through `LOG.warn` or `LOG.debug`.

Of course, our static pruning could prune out truly harmful bugs. However, given the huge number of message timing bug candidates reported by DCatch trace analysis, DCatch static pruning is valuable for prioritizing the bug detection focus.

Finally, our loop-based synchronization analysis is effective (Chapter 4.3.2). This analysis discovers both local while-loop custom synchronization and distributed pull-based custom synchronization. It pruned out false positives even after the intensive static pruning for all benchmarks, as shown in Table 4.5.

Triggering Overall, DCatch triggering module has been very useful for us to trigger message timing bugs and prune out false positives. As shown in Table 4.4, among the 47 DCatch bug reports with unique call stacks, the triggering module is able to automatically confirm 35 of them to be true races, with 23 of them causing severe failures, and the remaining 12 to be false positives in DCatch race detection.

The analysis conducted by DCatch about how to avoid hangs (challenge-1) and avoid large numbers of dynamic `_requests` (challenge-2) is crucial to trigger many DCatch bug reports. In fact, the naive approach that inserts `_request` just before the racing heap accesses failed to confirm 23 DCatch bug reports to be true races, out of the total 35 true races, exactly due to these two challenges. DCatch handles the challenge-1 by putting _requests outside critical sections (17 cases) or outside event/RPC handlers (6 cases), and handles the challenge-2 by moving the _requests along the happens-before graph into nodes different from the original race instructions (2 cases), exactly like what described in Chapter 4.5.2. To avoid hangs, DCatch first move `_request` from inside RPC handlers into RPC callers and then move `_request` to be right outside the critical sections that enclose corresponding RPC callers.

Of course, DCatch triggering module is not perfect. We expect two remaining challenges that it could encounter. First, DCatch cannot guarantee to find a location along the HB chains with few dynamic instances. Automated message timing bug triggering would be challenging in these cases, if failures only happen at a specific dynamic instance. The current prototype of DCatch focuses on the first dynamic instance of every racing instruction. This strategy allows DCatch to trigger the desired executing order among race instructions with 100% frequency for 33 true races in DCatch bug reports and with about 50% frequency for the remaining 2 true races. Second, DCatch does not record all non-deterministic environmental events and hence its triggering module may fail to observe a race instruction whose execution depends on unrecorded non-deterministic events.

False negative discussion DCatch is definitely not a panacea. DCatch could miss message timing bugs for several reasons. First, given how its static pruning is configured, the current prototype of DCatch only reports message timing bugs that lead to explicit failures, as discussed in Chapter 4.4.1. True message timing bugs that lead to severe but silent failures would be missed. This problem could be addressed by skipping the static pruning step, and simply applying the triggering module for all message timing bug candidates. This could be an option if the testing budget allows. Second, DCatch selectively monitors only memory accesses related to inter-node communication and corresponding computation. This strategy is crucial for the scalability of DCatch, as we will see soon in Chapter 4.7.4. However, there could be message timing bugs that are between communication-related memory accesses and communication-unrelated accesses. These bugs would be missed by DCatch. Fortunately, they are very rare in real world based on our study. Third, DCatch may not process extremely large traces. The scalability bottleneck of DCatch, when facing huge traces, is its trace analysis. It currently takes about 4G memory for the three largest traces in our benchmarks (HB-4729, MR-3274, and MR-4637). DCatch will need to chunk the traces and conduct detection within each chunk, an approach used by previous LCbug detection tools.

4.7.3 Performance Results

Run-time and off-line analysis time As shown in Table 4.6, DCatch performance is reasonable for in-house testing. DCatch tracing causes 1.9X – 5.5X slowdowns across all benchmarks. Furthermore, we found that up to 60% of the tracing time is actually spent in dynamic analysis and code transformation in Javassist. If we use a static instrumentation tool, the tracing performance could be largely improved. The trace analysis time is about 2–10 times of the baseline execution time. Fortunately, it scales well, roughly linearly, with

BugID	Base	Tracing	Trace Analysis	Static Pruning	Trace Size
CA-1011	6.6s	13.0s	15.9s	324s	7.7MB
HB-4539	1.1s	3.8s	11.9s	87s	4.9MB
HB-4729	3.5s	16.1s	36.8s	278s	19MB
MR-3274	21.2s	94.4s	62.2s	341s	22MB
MR-4637	11.7s	36.4s	51.5s	356s	18MB
ZK-1144	0.8s	3.6s	4.8s	25s	1.9MB
ZK-1270	0.2s	1.1s	4.5s	15s	1.3MB

Table 4.6: DCatch Performance Results

BugID	Total	Mem	RPC/Socket	Event	Thread	Lock
CA-1011	19,984	17,722	0 / 196	0	634	1432
HB-4539	3,907	3,233	260 / 0	21	89	304
HB-4729	11,297	9,694	449 / 0	18	144	992
MR-3274	31,526	23,528	752 / 0	3,540	1,390	2316
MR-4637	24,437	17,201	406 / 0	2,996	1,780	2054
ZK-1144	3,820	3,303	0 / 120	0	79	318
ZK-1270	5,367	4,227	0 / 389	0	329	422

Table 4.7: Break-downs of # of Major Types of Trace Records in DCatch

the trace size: taking about 2–3 second to process every 1MB of trace.

The static pruning phase takes 15 seconds to about 6 minutes for each benchmark. It is the most time consuming phase in DCatch as shown in the table. 20% – 89% of this analysis time is spent for WALA to build the Program Dependency Graph (PDG). Therefore, the pruning time would be greatly reduced, if future work can pre-compute the whole program PDG, store it to file, and load it on demand.

The time consumed by loop-based synchronization analysis is negligible comparing with tracing, trace analysis, and static pruning, and hence is not included in Table 4.6.

Tracing Details DCatch produces 1.2–21MB of traces for these benchmarks. These traces could have been much larger if DCatch did not selectively trace memory accesses, as we will see in Table 4.8. As shown in Table 4.7, these traces mostly contain memory access information. There are also a good number of RPC, socket, event, and thread related

BugID	Trace Size	Tracing Time	TraceAnalysis Time
CA-1011	77MB	15.9s	Out of Memory
HB-4539	26MB	10.2s	64.5s
HB-4729	60MB	49.9s	Out of Memory
MR-3274	839MB	215.3s	Out of Memory
MR-4637	639MB	137.8s	Out of Memory
ZK-1144	6.9MB	5.7s	6.5s
ZK-1270	25MB	4.4s	232.7s

Table 4.8: Full Memory Tracing Results.

records in DCatch traces. MapReduce benchmarks particularly have many event and thread related records, because MapReduce heavily uses event-driven computation. There are many event-handling threads and many event handlers further spawn threads. On the other hand, our workload did not touch the event-driven computation part of Cassandra and Zookeeper, consequently their traces do not contain event operations.

4.7.4 Comparison with Alternative Designs

Unselective memory-access logging DCatch tracing only selectively traces memory accesses related to inter-node communication and computation. This design choice is crucial in making DCatch scale to real-world systems. As shown in Table 4.8, full memory-access tracing will increase the trace size by up to 40 times. More importantly, for 4 out of the 7 benchmarks, trace analysis will run out of JVM memory (50GB of RAM) and cannot finish.

Alternative HB design DCatch HB model contains many rules. We want to check whether they have all taken effects in message timing bug detection, particularly those rules that do not exist in traditional multi-threaded programs. Therefore, we evaluated how many extra false positives and false negatives are reported by DCatch trace analysis when it ignores event, RPC, socket, and push-synchronization operations in traces, respectively, as shown in Table 4.9. Note that, the traces are the same as those used to produce results in Table 4.4,

BugID	#Static Ins. Pair			#Callstack Pair		
	Event	RPC_Soc	Push	Event	RPC_Soc	Push
CA-1011	-	-	-	-	-	-
HB-4539	-	-3/+35	-3/+35	-	-12/+115	-11/+110
HB-4729	-	-7/+37	-9/+36	-	-23/+109	-24/+106
MR-3274	-46/+4	-5/+16	-	-349/+ 8	-20/+ 18	-
MR-4637	-51/+4	-4/+17	-	-369/+11	-24/+ 20	-
ZK-1144	-	-	-	-	-	-
ZK-1270	-	-	-	-	-	-

Table 4.9: False negatives (before '/') and false positives (after '/') of ignoring certain HB-related operations

except that some trace records are ignored by analyzer.

Overall, modeling these HB-related operations are *all* very useful. Excluding any one type of them would lead to a good number of false positives and false negatives for multiple benchmarks, as shown in Table 4.9.

The false positives are easy to understand. Without these operations, corresponding HB relationships, related to Rule-E$^{\text{enq}}$, Rule-M$^{\text{RPC}}$, Rule-M$^{\text{soc}}$, and Rule-M$^{\text{push}}$, would be missed by trace analysis. Consequently, some memory access pairs would be mistakenly judged as concurrent.

The false negatives are all related to Rule-P$^{\text{nreg}}$. For example, when event-handler *Begin*s and *End*s are not traced, DCatch trace analysis would conclude that all memory accesses from the same event-handling thread are HB ordered. Consequently, DCatch would miss message timing bugs caused by conflicting memory accesses from concurrent event handlers. The same applies to false negatives caused by not tracking RPC/socket and Push-based synchronization operations.

Finally, CA-1011 and the two ZK benchmarks did not encounter extra false positives or negatives in static/callstack counts[3] due to lucky "*two wrongs make a right*": ignoring

3. CA-1011 did encounter 8% more false positives in raw counts – each pair of callstacks may have many dynamic instances.

socket-related operations misses some true HB relationships and also mistakenly establishes some non-existing HB relationships. Imagine node n_1 sends a message m_1 to node n_2, and n_2 sends m_2 back in response. Tracing socket operations or not would both reach the conclusion that send m_1 happens before receive m_2 on n_1, through different reasoning. Tracing socket operations provide correct HB relationships: *Send* $(m_1,\ n_1) \overset{M^{\text{soc}}}{\Longrightarrow}$ *Recv* $(m_1,\ n_2) \overset{P^{\text{nreg}}}{\Longrightarrow}$ *Send* $(m_2,\ n_2) \overset{M^{\text{soc}}}{\Longrightarrow}$ *Recv* $(m_2,\ n_1)$. Not tracing socket would mistakenly apply Rule-P$^{\text{reg}}$ to message-handling threads, and do the wrong reasoning: *Send* $(m_1,\ n_1) \overset{P^{\text{reg}}}{\Longrightarrow}$ *Recv* $(m_2,\ n_1)$. In short, tracing socket operations is still useful in providing accurate HB relationships.

4.8 Conclusion

Message timing bugs severely threat the reliability of distributed systems. They linger even in distributed transaction implementations [18, 51, 75]. In this Chapter, we designed and implemented an automated message timing bug detection tool for large real-world distributed systems. The DCatch happens-before model nicely combines causality relationships previously studied in synchronous and asynchronous single-machine systems and causality relationships unique to distributed systems. The four components of DCatch tool are carefully designed to suit the unique features of message timing bugs and distributed systems. The triggering module of DCatch can be used as a stand-alone testing framework. We believe DCatch is just a starting point in combating message timing bugs. The understanding about false negatives and false positives of DCatch will provide guidance for future work in detecting message timing bugs.

CHAPTER 5
FCATCH: AUTOMATICALLY DETECTING FAULT TIMING BUGS IN CLOUD SYSTEMS

It is crucial for distributed systems to achieve high availability. Unfortunately, this is challenging given the common component failures (i.e., faults). Developers often cannot anticipate all the timing conditions and system states under which a fault might occur, and introduce fault timing bugs, a new type of DCbugs, that only manifest when a node crashes or a message drops at a special moment. Although challenging, modeling and detecting fault timing bugs are fundamental to developing highly available distributed systems. Unlike previous work that relies on fault-injection to expose fault timing bugs, this chapter carefully models fault timing bugs as a new type of concurrency bugs, and develops FCatch [57] to automatically predict fault timing bugs by observing correct execution. Evaluation on four representative cloud-scale distributed systems shows that FCatch is effective, accurately finding severe fault timing bugs.

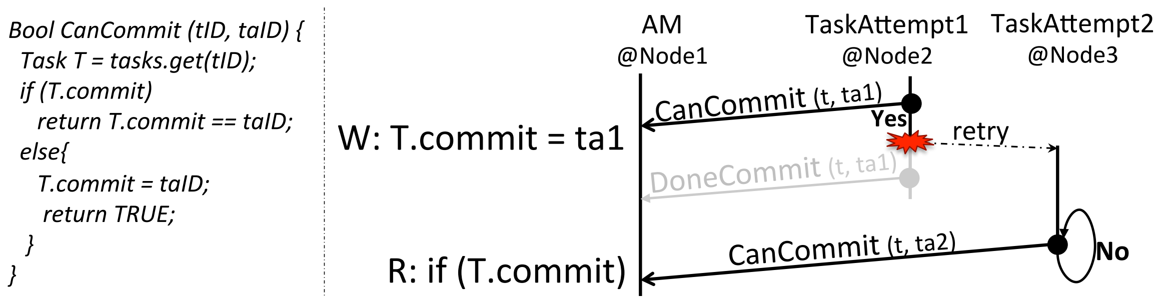
5.1 Introduction

5.1.1 Motivation

Fault timing bugs are difficult to expose during in-house testing due to their complicated triggering conditions — component failures, which need to be injected during testing, have to occur under special timing. As a result, they widely exist in deployed distributed systems [51, 34, 15].

Fault timing bugs are also *unique* to distributed systems, as single-machine systems, except for storage systems, are not expected to tolerate node crashes.

Figure 5.1 illustrates a fault timing bug from Hadoop-MapReduce. Here, a task attempt contacts Application Manager (AM) through an RPC `CanCommit` to get commit permission



```
Bool CanCommit (tID, taID) {
  Task T = tasks.get(tID);
  if (T.commit)
    return T.commit == taID;
  else{
    T.commit = taID;
    return TRUE;
  }
}
```

Figure 5.1: An example of fault timing bugs from MapReduce

and get its attempt-ID `ta1` recorded. Soon later, it will inform AM that it has successfully committed through `DoneCommit`. Hadoop usually can tolerate a task-attempt crash: if the crash is after `DoneCommit`, no recovery is needed as the global state is consistent; if the crash is before `CanCommit`, another attempt `ta2` will redo the task. Unfortunately, if the attempt crashes between the two RPC calls, which is a small time window comparing with the attempt's whole life time, the job will never finish. The reason is that `T.commit` on AM has been contaminated by the crashed attempt `ta1`, causing every recovery attempt to fail the `CanCommit` checking.

As we can see, the complexity of fault timing bugs is inherent to distributed systems, where nodes interact with each other through global files and messages. When a node N_{Crash} crashes, its remaining states scatter around the system, including global files updated by N_{Crash}, remaining nodes' heaps updated by RPC functions remotely invoked by N_{Crash} (e.g., `T.commit` on AM in Figure 5.1), and others. Depending on which node crashes at which point of the system execution, different system states could be left behind and demand different handling from the remaining live nodes and the recovery routine. Searching the huge state space of a distributed system for small time windows where a particular node's crash is improperly handled is a daunting task.

The state of practice in catching fault timing bugs is ineffective. It injects faults randomly, hoping to hit small bug-triggering time windows through many tries. Previous work [17] and our own experiments all show that real-world fault timing bugs often do not manifest even after hundreds of such random fault-injection runs with bug-triggering workload.

Recent research improves the efficiency of fault-injection testing, leveraging manual specification or expert knowledge of the system under test. Distributed-system model checkers use heuristics, like injecting faults only at message sending/receiving points, and rules specified by developers to avoid some unnecessary fault injections [33, 46, 50, 87, 96]. Lineage-driven fault injectors [16, 17] require system models manually written in a domain specific language. Some recent effort relies on domain experts to decide which part of the system is important and hence should take fault injection [15, 27, 30].

Although recent work has achieved great progresses, they all require much manual effort and/or deep understanding of the system under test. Furthermore, they still struggle at searching through the huge state space of distributed systems, with most (e.g. >99%) of their carefully designed fault injections not revealing any fault timing bugs [27, 15, 30, 50].

5.1.2 Contributions

Different from previous work that relies on random or manually-guided fault injections to *hit* fault timing bugs, our tool FCatch uses program analysis to automatically *predict* fault timing bugs with high accuracy, without manual specification or domain knowledge about the software under test. That is, by observing a correct execution with no faults or a successfully recovered fault at time t, FCatch can predict that the system execution would fail when a fault occurs at t' ($t' \neq t$).

A new fault timing bug model FCatch is built upon a new model of fault timing bugs. Different from previous work that essentially models fault timing bugs as semantic bugs, and hence requires semantic specifications in bug detection. We model fault timing bugs as a type of concurrency bugs, with two key properties:

- Triggering conditions: a fault timing bug is triggered by a special timing of fault f. If the fault f, like the task-attempt crash in Figure 5.1, occurs a bit earlier or later, the system would behave correctly.

- Root causes: once triggered, a fault timing bug can cause system failures when the faulty node N_{Crash} leaves a shared resource in a state that cannot be handled by another node N. For example, the failure in Figure 5.1 is caused by the original task attempt leaving a non-NULL `T.commit` in AM that cannot be handled by the recovery.

Starting from these two key properties, other detailed properties of fault timing bugs can then be reasoned about and guide fault timing bug detection, which we will describe more in Chapter 5.2.

A new fault timing bug detection approach This model provides new opportunities to detect fault timing bugs.

- The triggering conditions suggest that we can predict fault timing bugs by analyzing correct runs to see whether/how the system might behave differently when the time of fault changes. This approach uses runs under common timing and workload as correctness specifications, greatly reducing manual effort in fault timing bug detection.
- The root causes suggest that we can predict fault timing bugs by analyzing operations that read and write the same resource from different nodes, referred to as *conflicting* operations. This approach can help greatly shrink the fault timing bug search space, avoiding unnecessary checking.

FCatch Following the above observations and approach, we build FCatch that predicts fault timing bugs in three main steps, as shown in Figure 5.2.

First, FCatch monitors correct fault-free or faulty runs of a distributed system, tracing resource-access operations and fault-tolerance related operations. Particularly, FCatch carefully designs which correct runs to monitor — a fault timing bug like that in Figure 5.1

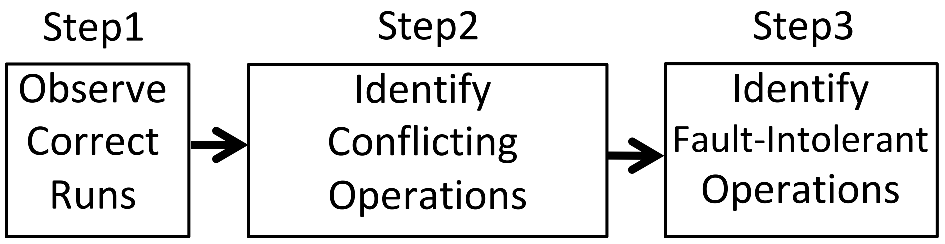

Step1
Observe Correct Runs
Step2
Identify Conflicting Operations
Step3
Identify Fault-Intolerant Operations

Figure 5.2: The flow of FCatch

requires monitoring and comparing more than one correct run to discover — and what to trace for each run. The details are presented in Chapter 5.3.

Second, FCatch analyzes traces to identify pairs of conflicting operations that write and read the same resource, such as data in heap or persistent storage, from different nodes. Particularly, following our fault timing bug model in Chapter 5.2, FCatch adapts traditional happens-before analysis to identify every pair of conflicting operations whose interaction can potentially be perturbed by the time of fault. The details are in Chapter 5.4.

Third, FCatch analyzes the distributed system to identify conflicting operations that are not protected by existing fault-tolerance mechanisms such as timeouts, sanity checks, and data resets. These operations are referred to as fault-intolerant operations. They are reported by FCatch as fault timing bugs. The details will be presented in Chapter 5.4.

Finally, FCatch tries to trigger every fault timing bug reported above, helping developers confirm which reported bugs can truly cause failures. The details are in Chapter 5.5.

We evaluated FCatch using a set of 7 fault timing bugs collected by an existing benchmark suite of real-world distributed-system concurrency bugs [51]. These 7 fault timing bugs come from 4 widely used distributed systems, Cassandra, HBase, MapReduce, and ZooKeeper, and can be triggered by 6 common workloads under special time of faults.

By analyzing only one or two **correct** runs of each workload, FCatch generates 31 fault timing bug reports. These include 8 reports that explain the 7 benchmarks, 8 reportsthat are truly severe fault timing bugs beyond the initial benchmark suite [51], which we were

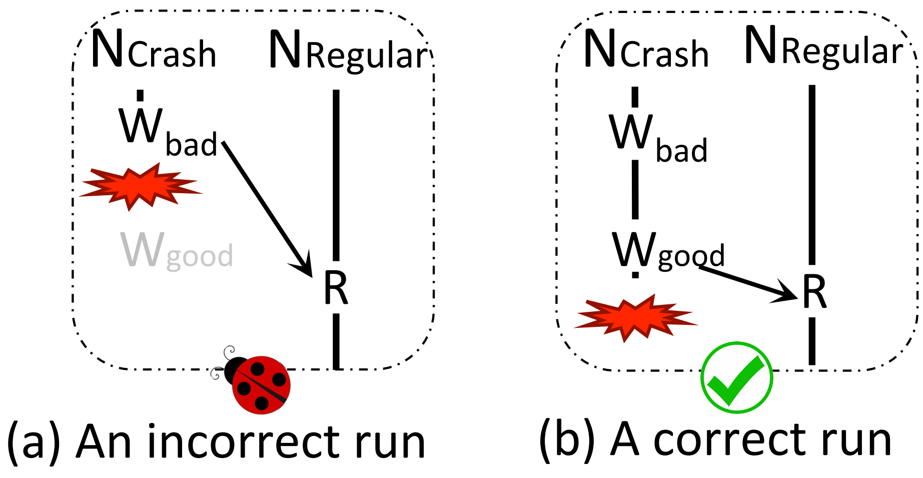


Figure 5.3: Conflicting operations in a crash-regular bug.

unaware of before our experiments, 6 that cause well-handled exceptions, and 9 that are benign. All the false positives can be easily pruned by FCatch's automated bug-triggering module. In comparison, only one out of all these bugs could be exposed after 400 fault-injection runs of corresponding bug-triggering workload, and even this one bug has less than 3% of manifestation rate. FCatch bug detection introduces 5X – 15X slowdown to the baseline fault-free execution, suitable for in-house testing.

5.2 Modeling Fault Timing Bugs

We categorize fault timing bugs into two types: (1) crash-regular bugs, where conflicting operations are from the crash node N_{Crash} and a non-crash node N_{Regular}, illustrated in Figure 5.3; (2) crash-recovery bugs, where conflicting operations are from the crash node N_{Crash} and the recovery node N_{Recovery}, exemplified in Figure 5.1 and Figure 5.4.

In the following, we present our detailed models of these two types of fault timing bugs, which guide the design of FCatch.

Terminology We refer to *faults* as component, rather than system, failures that need to be tolerated. Among different types of faults, we focus on node crashes and message drops.

We use *faults* and *crashes* interchangeably, unless otherwise specified. When deploying a distributed system, one can configure a process to run by itself or co-located with other processes in a physical node. Therefore, we use *node* and *process* interchangeably. When we say an operation O is *from* a node N, O could be physically executing on N or causally initiated by N, like in an RPC function remotely invoked by N. Chapter 5.4.1 presents how we analyze such causal relationships. The *happens-before* relationship discussed below is the same logical timing relationship discussed in previous work [48, 56], which we will elaborate in Chapter 5.4.1.

5.2.1 Crash-regular Fault Timing Bugs

What are these bugs? As shown in Figure 5.3a, by definition, a crash-regular bug manifests when a crash causes a regular node N_{Regular} to read, denoted by R, a shared resource defined by an unexpected source W_{bad} in node N_{Crash}.

Thinking about why the time of fault could make W_{bad} an unexpected source for R reveals more information:

1. The expected source for R, denoted as W_{good}, must execute after W_{bad} from N_{Crash} during correct runs (i.e., Figure 5.3b), and consequently could disappear due to untimely crashes in incorrect runs (i.e., Figure 5.3a). Otherwise, if W_{good} executes before W_{bad}, it would not be affected by any crashes after W_{bad} and hence leaves no chances for any crashes to make R read from W_{bad} instead of W_{good}.

2. W_{good} must always execute before R during correct runs. That is, W_{good} must *happen before* R. Otherwise, if it is concurrent with R, W_{bad} could become a source for R even without untimely crashes.

3. The disappearance of W_{good} must block the execution of R (e.g., the missing of a `signal` causing a `wait` to block forever), leading to failures. As we will discuss in

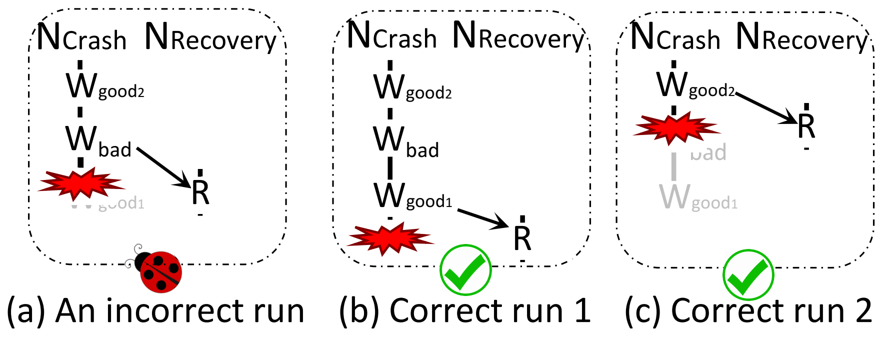


Figure 5.4: Conflicting operations in a crash-recovery bug.

Chapter 5.4.1, given that W_{good} happens before R, the only other possibility is that W_{good}'s disappearance causes the whole thread/function of R to disappear (e.g., the missing of an RPC call causing the RPC-handler not to execute), which is a natural consequence of the node crash yet not a bug.

4. There must be no fault-tolerance mechanism that can unblock R. For example, with a timeout, a common fault-tolerance mechanism, the above bad interaction between W_{bad} and R would have been tolerated.

This gives us a profile of a crash-regular fault timing bug. It is related to a pair of conflicting operations $\{W, R\}$. R consumes the state of a shared resource, which is a heap object on a non-crash node or persistent storage data anywhere, defined by W during correct runs (i.e., W_{good} in Figure 5.3). When the node, which W comes from, crashes before W, R's execution is blocked forever, causing hangs and related failures.

How to detect these bugs? With the above model, instantiating the general flow in Figure 5.2 to detect crash-regular fault timing bugs is straightforward.

First, FCatch monitors a fault-free run of the target system, recording resource-access operations, happens-before operations, and time-out operations (Chapter 5.3).

Second, FCatch identifies pairs of conflicting operations from different nodes that have blocking happens-before relationship with each other (Chapter 5.4.2).

Third, FCatch checks time-out mechanisms in the target system and reports conflicting and fault-intolerant crash-regular operation pairs as candidate bugs (Chapter 5.4.2).

Finally, FCatch tries to trigger a reported {W, R} crash-regular bug by crashing the node of W right before W or dropping the message where W is from (Chapter 5.5).

5.2.2 Crash-recovery Fault Timing Bugs

What are these bugs? By definition, crash-recovery bugs differ from crash-regular bugs in where the read operation R comes from. In other words, they differ on whether the impact of the crash is incorrectly handled by the recovery node N_{Recovery} or a regular node N_{Regular}.

This seemingly small difference fundamentally changes the define-use relationship for R. When R is in a regular node, what resource content it consumes is affected by the happens-before relationship and program synchronization. For example, given an update W in node N_{Crash} and that R happens before W, W can never define the content consumed by R no matter when the crash occurs. However, when R is in a recovery node, what resource content it consumes is *completely* determined by the time of crash. Traditional happens-before relationship and program synchronization can*not* affect when N_{Crash} crashes and hence can*not* regulate when N_{Recovery} starts.

With this in mind, we think about why the time of fault could make W_{bad} an unexpected source for R:

1. The expected source for R could execute either after W_{bad} (Figure 5.4b) or before W_{bad} (Figure 5.4c) during a correct run, when the fault occurs later or earlier than that during the incorrect run.

2. There must be no fault-tolerance mechanism that can prevent R from consuming content defined by W_{bad}. Analyzing and identifying this type of fault-tolerance mechanisms is much more complicated than identifying time outs, and involves both data-dependence and control-dependence analysis, which we will discuss in Chapter 5.4.3.

This gives us a profile of a crash-recovery fault timing bug. It is related to a pair of conflicting operations $\{W, R\}$, with W from N_{Crash} and R from N_{Recovery}. N_{Recovery} tries to recover the crash of N_{Crash}. Depending on the time of the crash, the recovery attempt sometimes succeeds, with R consuming shared-resource content defined by W, and sometimes fails, with R consuming shared-resource content defined after W or before W.

How to detect these bugs? We follow the above model to instantiate a crash-recovery bug detection flow.

The first step is more challenging than that for crash-regular fault timing bugs because we may need to observe both a fault-free run and a faulty run, and stitch them together. If only observe a fault-free run, we cannot possibly know what are the recovery operations like R. If only observe a correct faulty run like the one in Figure 5.4c, we cannot know what operations, including W_{bad}, could have happened if the node crashed later. Essentially, we may need to predict a fault timing bug {W, R}, with its W observed in one run and its R observed in another. Chapter 5.3 will discuss our solutions.

At the second step, we identify every pair of conflicting operations $\{W, R\}$, so that W from node N_{Crash} defines the shared-resource content consumed by R from node N_{Recovery} during a correct faulty run, together with an alternate W' that updates the same shared resource right after or right before W from N_{Crash} (Chapter 5.4.3).

Third, we conduct control-flow and data-flow analysis to prune out a bug candidate $\{W, R\}$, if fault-tolerance mechanisms in software prevent R from consuming content defined by the alternate update W' (Chapter 5.4.3).

Finally, FCatch tries to trigger a reported crash-recovery bug {W, R} by crashing the node of W right before or after W (Chapter 5.5).

5.2.3 Discussion

Our model above does not cover all bugs that are related to faults in distributed systems. We only cover node-crash and some message-drop faults. We do not cover bugs that are related to faults, but not the timing of faults. For example, some exception handlers are incorrectly implemented and would lead to failures every time they are invoked. We also do not cover bugs that may involve the interaction among more than one fault or more than one resource. Previous work on detecting multi-variable execution timing bugs could help extend FCatch to tackle these bugs.

5.3 Tracing Correct Runs

This chapter discusses how FCatch monitors correct system execution and generates traces. The next chapter will explain how FCatch analyzes these traces to detect fault timing bugs.

5.3.1 Which Correct Runs to Observe?

As mentioned in Chapter 5.2.1, to predict crash-regular fault timing bugs, FCatch can trace any correct run. However, to predict crash-recovery fault timing bugs like the one in Figure 5.1, the tracing is much more challenging and hence is the focus below.

The first challenge is that conflicting operations of a crash-recovery bug may *never* appear together in the same correct run. Consequently, no dynamic detection tool can possibly predict such a bug by observing just one correct run, a unique challenge that does not exist in traditional concurrency-bug detection.

For example, in Figure 5.1, after AM executes the first `CanCommit` RPC for a task T, it will never return `TRUE` for `CanCommit` invoked by other attempts on T. Since both conflicting operations are in `CanCommit`, once one executes, the other can only execute in a task attempt that cannot finish due to the repeated `FALSE` returns from `CanComit`. At that time, it is

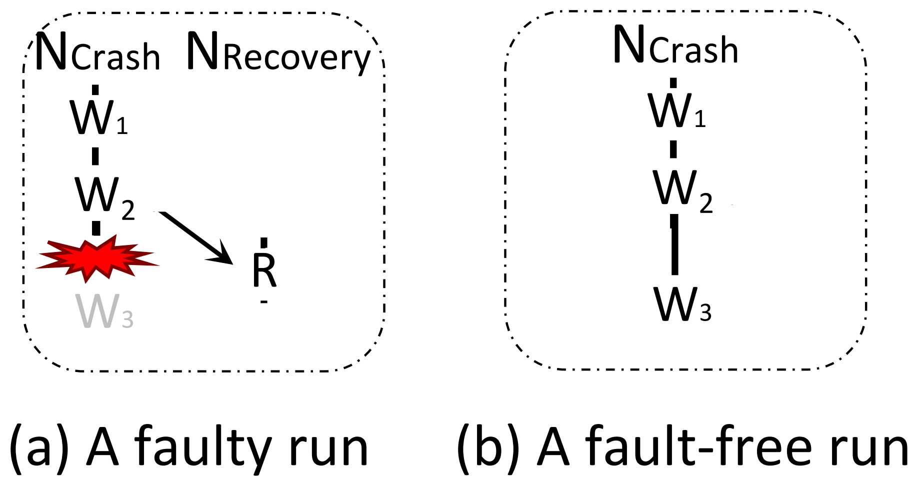


Figure 5.5: Observe two runs to detect a crash-recovery bug

already too late to *predict* the bug.

To address this challenge, we predict a crash-recovery bug based on not one correct run, but two complementing correct runs, a faulty one and a fault-free one under the same workload as shown in Figure 5.5. From these two runs, we can observe not only what happened on N_{Crash} prior to the crash (e.g., W_1 and W_2 in Figure 5.5a) and what happened in the recovery routine on N_{recovery} based on the faulty run (e.g., R in Figure 5.5a), but also what *could have* happened on N_{crash} if the fault occurred later based on the fault-free run (e.g., W_3 in Figure 5.5b).

Take the bug in Figure 5.1 as an example. To predict this bug, we observe two runs: one has the attempt crashing *anywhere* before `CanCommit`, which reveals the `T.commit` checking from the recovery attempt, and one has the attempt finishing successfully, which reveals the setting of `T.commit` from the initial attempt.

However, **a second challenge** comes: how to stitch traces from two runs together. Since an object's hash code changes from run to run, we cannot tell whether an operation from one run, like the write on `T.commit` in a fault-free run, accesses the same object as an operation from another run, like the read of `T.commit` in a faulty run. Furthermore, given the inherent non-determinism in distributed systems, the two runs likely already diverged prior to the crash point. Consequently, naively using the fault-free run to predict what might happen if the fault occurs later in the faulty run might introduce a lot of inaccuracies in detection.

Our solution leverages virtual machines' checkpointing mechanism, a standard technique supported by all types of virtual machine (VirtualBox, VMWare, etc.) and container (Docker) systems. The high-level idea is to run the whole distributed system inside a Virtual Machine, and take a whole system checkpoint at a (random) point. We then make the system resume from the checkpoint in two ways, one without any fault injection and the other one injects a node crash immediately after the checkpoint. After both runs finish the workload, we will get a pair of perfectly complementing runs. These two runs share exactly the same execution prefix and heap object layout prior to the checkpoint. Consequently, we completely eliminate the execution non-determinism problem prior to the crash, and we can simply use object hash code to identify conflicting operations across the two runs. This idea can be implemented on any virtual machine infrastructure. Our current implementation is built upon VirtualBox Virtual Machine Monitor [76] and we will explain in Chapter 5.6.

Note that, although the scheme above requires a faulty run, it is fundamentally different from previous fault-injection testing. FCatch needs a correct faulty run, which is abundant in a mature distributed system, to predict fault timing bugs; previous fault-injection testing needs an incorrect faulty run, which is naturally rare, to observe a fault timing bug. To detect the bug in Figure 5.1, FCatch needs a run where the attempt crashes before `CanCommit`. Since an attempt spends most time before `CanCommit`, almost every random fault injection works.

5.3.2 *What to Trace?*

FCatch generates one trace for every thread in system. It traces two sets of operations. The first set includes operations that affect intra-node and inter-node causal and blocking relationship, such as thread creation/join, signal/wait, event operations, message, RPC, etc (more details in Chapter 5.4.1). The second set includes all candidates for *conflicting* *operations*, which are operations that access two types of shared resources.

The first type of resources are heap objects on a non-crashing node, like `T.commit` on AM in Figure 5.1. Specifically, FCatch traces all heap accesses inside message handlers, RPC functions, event handlers and their callee functions, which are most related to inter-node communication and computation. FCatch does not trace all heap accesses in a system so that its dynamic tracing and trace analysis can scale, which we will evaluate in Chapter 5.8.2. FCatch will not consider heap objects on the crash node in its analysis, as their content will be wiped off by the crash and hence have no impact to the remaining system.

The second type of resource is persistent data in file systems, key-value stores, etc. Specifically, FCatch traces operations on local and global files (e.g., HDFS files), and records in distributed key-value stores (ZooKeeper ZKnodes). We consider create, delete, read, write, rename, and check-if-exist APIs for each storage system.

Every record of a traced operation consists of four parts: (1) operation type, specifying which type of operations on which type of resources; (2) callstack; (3) local time stamp counter obtained by `RDTSCP`, which provides an approximate nano-second level ordering among operations in one machine; (4) ID, which uniquely identifies every shared resource and every message/RPC/event. We use JVM hash code for heap object ID[1]; use path string for file ID; use random integer inserted as extra parameter (field) for RPC (message) ID.

5.4 Predicting Fault Timing Bugs

5.4.1 Causal and Blocking Relationship Analysis

We first discuss two sub-types of happens-before relationship, referred to as causal and blocking relationships. The latter helps FCatch detect crash-regular fault timing bugs; the former helps FCatch judge whether an operation that physically executes on a node N

1. Object::hashCode() is a practical method provided by JVM [12] to return distinct integers for distinct objects.

logically comes from another node N', which is used in detecting both crash-regular and crash-recovery fault timing bugs.

Causal relationship and blocking relationship We define an operation A to *causally depend* on an operation B, denoted as $B \overset{c}{\rightarrow} A$, if the disappearance of B causes A to also disappear. For example, the disappearance of a message-sending operation causes operations inside the message handler to also disappear. Given $B \overset{c}{\rightarrow} A$, we consider A to logically come from or get initiated by the node N_B that B physically executes on.

We define an operation A to *blocking depend* on B, denoted as $B \overset{b}{\rightarrow} A$, if the disappearance of B causes the thread that contains A to hang. For example, the disappearance of a signal operation causes the thread conducting the wait operation to hang.

Causal operations and blocking operations We take distributed systems' common happens-before operations [56], and separate them into causal and blocking operations.

FCatch considers all common intra-thread (event-based), inter-thread, and inter-node causal operations in distributed systems: (1) event enqueue. $EnQ(e) \overset{c}{\rightarrow}_d o$ where o is any operation inside the handler of `e` and its callee functions ($\overset{c}{\rightarrow}_d$ denotes *direct* causal relationship); (2) thread creation. $create(t) \overset{c}{\rightarrow}_d o$, where o is any operation inside thread `t`; (3) RPC invocation. $call(R) \overset{c}{\rightarrow}_d o$, where o is inside RPC function `R` and callees; (4) message sending. $send(m) \overset{c}{\rightarrow}_d o$, where o is inside the message handler of `m` and callees; (5) a state update through a synchronization service like ZooKeeper. $update(s) \overset{c}{\rightarrow}_d notify(s)$, where $notify(s)$ is the corresponding notification operation.

Given $B \overset{c}{\rightarrow}_d A$, we call A as the *Causee* of B and B the *Causor* of A. An operation has at most one causor. A causal operation may have more than one causee.

FCatch considers two common types of blocking operations: standard condition-variable signal and custom while-loop signal. The disappearance of the former would block corresponding waits and the waiting threads. The disappearance of the latter would block a corresponding while loop.

Causality analysis Causal relationship is transitive. If $B \overset{c}{\rightarrow} A$ and $C \overset{c}{\rightarrow} B$, then $C \overset{c}{\rightarrow} A$. Since FCatch traces the execution of all causal operations, it is straightforward to identify all the operations $\mathbb{O}_{\mathbb{S}\rightarrow}$ that causally depend on a set of seed operations $\mathbb{S}$ (Algorithm 1), and to identify all the operations $\mathbb{O}_{\rightarrow\mathbb{S}}$ that a set of seed operations $\mathbb{S}$ causally depend on (Algorithm 2). The former will be used to identify operations that logically come from N_{Crash} and N_{Recovery} (Chapter 5.4.3), and the latter will be used to identify operations that physically execute on one node yet logically come from another (Chapter 5.4.2).

```
Input: A set of seed operations $\mathbb{S}$
Output: A set of operations $\mathbb{O}_{\mathbb{S}\rightarrow}$ that causally depend on $\mathbb{S}$
Stack<Op> WorkingSet = $\mathbb{S}$;
Set<Op> VisitedSet = Null;
Set<Op> $\mathbb{O}_{\mathbb{S}\rightarrow}$ = Null;
while $h$ = WorkingSet.pop() do
    if h.type == Causal_Operation then
        while $c$ = Causee(h).pop() do
            if VisitedSet.Add (c) then
                WorkingSet.push ($c$);
        end
    $\mathbb{O}$.push ($h$);
end
return $\mathbb{O}$;
```

Algorithm 1: Identify operations causally depending on $\mathbb{S}$

```
Input: A set of seed operations $\mathbb{S}$
Output: A set of operations $\mathbb{O}_{\rightarrow\mathbb{S}}$ that causally cause $\mathbb{S}$
Stack<Op> WorkingSet = $\mathbb{S}$;
Set<Op> VisitedSet = Null;
Set<Op> $\mathbb{O}_{\rightarrow\mathbb{S}}$ = Null;
while $h$ = WorkingSet.pop() do
    if VisitedSet.Add (Causor(h)) then
        WorkingSet.push (Causor(h));
    $\mathbb{O}$.push ($h$);
end
return $\mathbb{O}$;
```

Algorithm 2: Identify operations $\mathbb{S}$ causally depending on

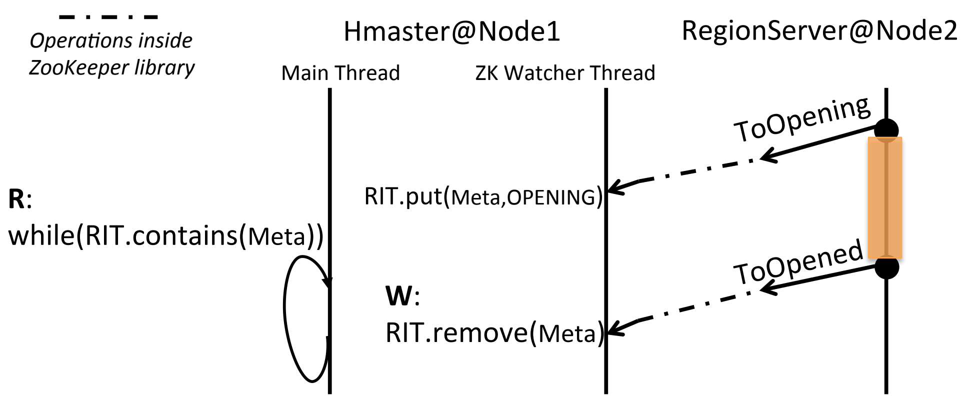


Figure 5.6: An example of crash-regular bugs from HBase

5.4.2 Predicting Crash-Regular Fault Timing Bugs

A crash-regular fault timing bug is triggered by unexpected interaction between a crashing node and a regular node. Figure 5.6 illustrates a bug from HBase, a widely used key-value store system. When a RegionServer (RS) starts to open the `Meta` table, it registers `OPENING` with ZooKeeper, which then causes a `Meta` record to be added in the `RIT` map. The master node HMaster then waits for RS to finish by repeatedly checking `RIT` in a loop, denoted as R in Figure 5.6. When RS finishes, it registers `OPENED` with ZooKeeper, which through a chain of events causes the `Meta` record to disappear from `RIT`, denoted as W in figure. W helps HMaster to jump out of its loop. Unfortunately, if RS crashes after registering `OPENING` and before registering `OPENED` with ZooKeeper (or if the `ToOpened` message drops), HMaster will hang in the while-loop and make the **whole** HBase system unavailable

This type of fault timing bugs also have small triggering time-windows, and hence are difficult to expose during testing. In Figure 5.6, the failure-triggering fault window, denoted by an orange stripe, only lasts while RS creates two HDFS files and a ZooKeeper znode, a small window comparing with the whole HBase start-up and request-serving time. If RS crashes outside this window, HMaster will correctly re-assign the `Meta` table to another region server without hangs.

As discussed in Chapter 5.2, FCatch predicts this type of bugs by analyzing a correct-run trace in two steps.

Identifying conflicting operations

Is one blocking the other? FCatch first identifies W–R pairs that satisfy blocking relationship $W \xrightarrow{b} R$, as defined in Chapter 5.4.1. That is, FCatch needs to identify standard and custom signal-wait pairs, where the disappearance of the signal would block the thread-/handler of the wait.

Identifying the former is straightforward. FCatch records every signal/wait right before it is invoked with a timestamp and an ID of the corresponding condition variable (CV). When processing the trace, FCatch orders all the `signal` and `wait` operations on a CV based on their timestamps and matches each `wait` with the first `signal` after it.

Identifying the latter follows state-of-art techniques in finding custom synchronization loops [89, 93]. We first use static analysis to identify likely-synchronization loops and heap reads that affect the exits of these loops through control and data dependency analysis (details in Chapter 5.6). FCatch traces all these reads dynamically. During trace analysis, FCatch identifies every heap write W whose update is used by at least one such heap read R from a different thread or handler, based on time-stamp and object hash-code comparison. This pair of W and R is then treated as a pair of custom signal and wait.

Are they from different nodes? Every pair of fault timing bug candidate identified above, W and R, physically executes on the same node N. Next, we check whether W might logically come from another node, whose fault can cause R to hang.

We simply use Algorithm 2 to find all operations that W causally depends on. Once we find an operation W', $W' \xrightarrow{c} W$, that physically executes on a different node N', we consider W and R to pass the checking and will report them as a fault timing bug— once W' disappears due to a node crash or a message drop, W would disappear and the thread holding R on N would hang. For example, while analyzing W in Figure 5.6, back tracking along the arrows eventually brings us to the RegionServer. Consequently, we know that a fault on the RegionServer will cause HMaster to hang. This analysis cannot be done statically, as the dynamic context could decide whether a function and hence operations within it are invoked by a local thread or a remote RPC call.

Checking fault-tolerance mechanisms

FCatch statically checks whether there are time-outs that prevent R from waiting for the corresponding W forever. Given a signal-wait fault timing bug candidate, we simply check whether the wait API is associated with timeout parameters, such as `Object::wait(long timeout)` instead of `Object::wait(void)`. Given a while-loop fault timing bug candidate, we first check whether there exists a system-time acquisition function such as `System.currentTimeMillis` inside the loop body, and then check whether the time variable affects the loop exit condition through control or data dependence.

Of course, our current implementation may miss some time-out mechanisms, such as when dedicated monitoring threads are used to terminate a hanging loop or a wait.

5.4.3 Predicting Crash-Recovery Fault Timing Bugs

Crash-Recovery fault timing bugs are caused by interaction between the crashing node and the recovery node, like in Figure 5.1. Given a pair of traces from a faulty run, where N_{crash} crashes at time T_C, and a complementing fault-free run, FCatch will report conflicting operations that might cause the system to misbehave if the crash occurs before or after T_C.

Identifying conflicting operations

FCatch needs to identify operation pairs $\{W, R\}$ that access the same shared resource from the crashing node and the recovery node.

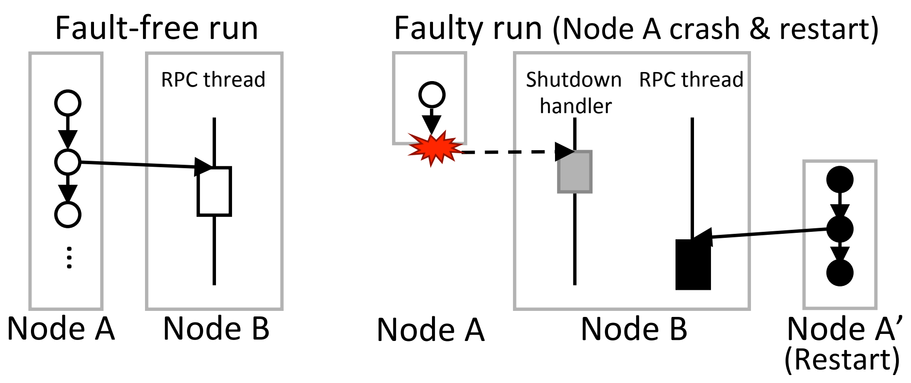


Figure 5.7: Crashing (white) and recovery operations (black, gray).

To identify operations that access persistent resources *physically* on the crashing or recovery node is trivial.

To identify operations that causally come from N_{Crash} (N_{Recovery}) but access persistent or heap resources outside it, we simply use all the RPC invocations and message sendings that escape N_{Crash} (N_{Recovery}) as seed operations and feed them into Algorithm 1.

Once all the recovery operations and crash operations are identified from the faulty trace and the fault-free trace respectively, finding pairs of them that access the same resource simply requires comparing corresponding resource IDs in the traces. Benefiting from how we obtain the perfectly pairing traces, no resource-ID translation is needed across traces[2].

So far, we have not discussed what exactly is the *recovery node*. In practice, it could be a restarted node that tries to recover from its recent crash, which happens in almost all systems; it could be a new process launched on a remaining node, which happens in MapReduce; it could also be a node-crash handler running inside an existing process through a unified interface, such as `IFailureDetectionEventListener::convict` in Cassandra and `ZooKeeperListener::nodeDeleted` in HBase, or a diverse set of functions in MapReduce and ZooKeeper.

In theory, FCatch can work with all these cases. By default, FCatch only treats processes

2. There could be inaccuracy if the crash op. and the recovery op. both access a heap object created after the crash point, but this is very rare.

that exist in the faulty trace but not in the fault-free trace as recovery nodes, obtained by process ID comparison (e.g., node A' in Figure 5.7). All operations in these processes or causally depend on these processes (e.g., all black shapes on node A' and B in Figure 5.7) are considered recovery operations. If developers specify recovery-handler interfaces or functions, FCatch can identify more recovery operations, such as the gray operations in Figure 5.7.

Checking fault-intolerance mechanisms

We generalize fault-tolerance mechanisms that can help prevent crash-recovery fault timing bugs into two types: (1) the ones that use sanity checks to allow R skipping the consumption of unexpected content, and (2) the ones that update the content of a shared resource at the beginning of a recovery routine to make sure that no left-over content can reach R. We identify these two types of fault-tolerance mechanisms through the following control-dependency and data-dependency analysis.

Control-dependency analysis The recovery node may conduct sanity checks to decide which shared resources it will access and how, which helps prevent many crash-recovery fault timing bugs. Figure 5.8 shows a simplified example from ZooKeeper. Here, a restarted ZooKeeper node checks the length and checksum segment of snapshot files, denoted by `f.valid()`, so that it can use the latest consistent snapshot file to recover its `DataTree`. This sanity check, denoted by `R1` in figure, prevents the ZooKeeper recovery routine from consuming invalid snapshot files left by the crash node.

To identify the above fault-tolerance mechanism, we first generalize it into the following pattern. There are two pairs of conflicting crash-recovery operations (i.e., W–R_1 and W–R_2 in Figure 5.8). The two recovery operations have control dependence with each other. Consequently, one serves as the fault-tolerance sanity check for the other.

Following this pattern, FCatch goes through every pair of recovery operations R_1 and R_2

```
//crash operations                 //recovery operations during reboot
f.write(); //W: take snapshot      DataTree dt = null;
…                                  for (File f : dir.sortedSnapshots())
                                     if (f.valid()) { //R1
                                       dt = f.read(); //restore; R2
                                       break;
                                     }
```

Figure 5.8: An example of sanity check in Zookeeper

touching the same resource, with each involved in any least one crash-recovery fault timing bug candidate. If FCatch finds R_2 to control depend on R_1, all conflicting-operation pairs that contains R_2 are pruned.

All the dependency checking in FCatch is implemented in WALA. We check both intra-procedural and inter-procedural dependence following the call-stack recorded in trace where R_1 affects the return value of a function in its call-stack and this return value affects whether R_2 executes or not.

Data-dependency analysis The recovery node sometimes resets the content of a shared resource before using it. This way, later reads will not consume content left by the crashing node. For example, in MapReduce, every submitted job is associated with a state variable and a job-report URL string. When an AM crashes, the recovery handler in RM immediately resets the URL string to `null` and the state to `INIT`.

To identify this type of fault-tolerance mechanisms, given a recovery operation R that is part of a crash-recovery fault timing bug candidate, we search through all traced recovery operations for a write operation that accesses the same resource as R and executes before R judged by the causality relationship or their timestamps. If such a write operation exists, we prune out any fault timing bug candidate that contains R.

Impact estimation

Not all conflicting fault-intolerance pairs identified above can cause harms. FCatch statically estimates the impact of each pair and prunes out the ones that are likely benign.

Specifically, given an operation R observed in a correct run, FCatch statically checks whether R might lead to any signs of potential harms either locally or globally because of consuming unexpected content. FCatch considers R to have a failure-prone local impact if it affects an exception `throw`, the printing of a severe error (i.e., fatal-level log printing), the creation of an event, and the startup of a service through data or control-dependence. FCatch considers R to have a failure-prone global impact if it can affect the return value of an RPC function or an RPC/message invocation/sending through data or control dependency. If FCatch finds that R has no failure-prone impact either locally or globally, FCatch prunes all the races that contain R out from the final fault timing bug reports, similar as previous work on distributed concurrency bug detection [56].

5.5 Fault Timing Bug Triggering

FCatch tries to automatically trigger every FCatch bug report, so that we can observe whether the reported bug is harmful or not.

Each report of a crash-regular fault timing bug contains a triplet (W, R, W'). W and R physically execute on one node; W' is an RPC/message sending operation physically located on another node; $W \overset{b}{\rightarrow} R$ and $W' \overset{c}{\rightarrow} W$. To trigger this bug, FCatch tries injecting three types of faults right before W': 1) node crash emulated by `Runtime.getRuntime.halt(-1)`; 2) kernel-level message drop due to network-connection broken, emulated by skipping W' and throwing `java.net. SocketException`; 3) application-level message drop emulated by skipping W'. This only applies for Cassandra's droppable messages, which are dropped if staying for too long in the sending queue. We inject a special tag for the W' message, so that the networking-related code can differentiate this message from others and apply fault-injection correctly. We will evaluate the difference between these different types of faults in Chapter 5.8.4.

Each report of a crash-recovery fault timing bug contains a pair (W, R), with W from

```
void snapshotANDCrash (Runtime t) {
  r.exec("ssh host snapshot.sh");
  sleep(5);
  bool crash_flag = r.exec("ssh host read flagfile");
  if (crash_flag == "crash") {
    r.halt(-1);
  }
}
```

Figure 5.9: Our snapshot-and-crash wrapper

N_{crash} and R from N_{recovery}. If W was already observed by FCatch during a correct faulty run, FCatch triggering script injects a node crash right before W to check what might happen if W does not appear due to the crash. If W was not already observed in a faulty run but only observed by FCatch in a fault-free run, FCatch triggering script injects a node crash right after W to check what might happen if W appears prior to the node crash.

5.6 Implementation Details

Code Instrumentation FCatch implements its code instrumentation in Javassist [39], a Java byte-code transformation tool, and its static analysis in WALA [38], a static Java analysis framework. Specifically, FCatch uses WALA to statically identify functions to instrument following super-class interfaces like `VersionedProtocol` interface for RPC, `java.Thread.thread` for thread functions, `org.apache.hadoop.yarn.event.EventHandler` for event handling, etc. FCatch uses Javassist to insert tracing functions before heap accesses (i.e., `getfield/putfield` instructions) and static-variable accesses (i.e., `getstatic/putstatic` instructions).

Virtual machine checkpointing Once randomly decide a crash point in the program, we insert a snapshot-and-crash wrapper function into the program, as shown in Figure 5.9. We then run the distributed system inside a VirtualBox VM. Before the crash point is reached, our `snapshotANDcrash` function invokes a script in the host OS to execute a VM-snapshot command asynchronously. The execution then reaches the `sleep` statement, which helps make sure that the snapshot is taken in the middle of the sleep. After the sleep, the original execution reaches Line 5 in Figure 5.9. There, a configuration file will decide not to inject a node-crash fault, and the execution will eventually finish as the *fault-free run*. After the *fault-free run*, we change the `flagfile` to allow fault injection, and then launch the Virtual Machine again starting from the snapshot recently taken. This time, the program resumes from inside the `sleep`, reaches the reading of `flagfile`, and eventually triggers a node crash through `r.halt(-1)` (Line 6 in Figure 5.9). The system continues after the node crash and contributes a *fault run* for FCatch bug detection. We currently emulate a node crash by terminating a process through `halt(-1)`. If we want to emulate crashing a multi-process node, we can simply `halt` multiple processes accordingly.

Synchronization loop identification To detect crash-regular fault timing bugs, the tracing component of FCatch statically identifies loops satisfying the following two conditions as likely synchronization loops. First, the loop is not bounded on a constant value or the size of a container (e.g., `for(element e:  list)` and `while(iterator.hasNext())`). Second, the loop body relinquishes the CPU through APIs like `Thread::sleep` and `Object::wait`.

FCatch false positives and negatives Like most bug detectors, FCatch is neither sound nor complete for several reasons. (1) Bug modeling. FCatch does not cover bugs beyond our fault timing bug model (Chapter 5.2.3). (2) (no) Specifications. FCatch eases fault timing bug detection by not relying on any manual specifications. Inevitably, this could cause inaccuracies. FCatch may miss recovery operations as discussed in Chapter 5.4.3, and may ignore true bugs whose harm does not match what FCatch automatically checks in Chapter 5.4.3. (3) Custom synchronization. Like all techniques that identify custom synchronization [21, 89, 93], FCatch cannot guarantee to accurately identify all fault-tolerance mechanisms or synchronization loops. For example, FCatch does not consider sanity checks involving

Table 5.1: FCatch Benchmarks.

App.	Version	Workload	Bench. Bugs
CA	1.1.12	Startup + AntiEntropy (AE)	CA1 [7], CA2 [8]
HB	0.96.0	Startup + HMasterRestart	HB1 [9]
HB	0.90.1	Startup	HB2 [5]
MR	0.23.1	Startup + WordCount(WC)	MR1 [6]
MR	2.1.1	Startup + WordCount(WC)	MR2 [10]
ZK	3.4.5	Startup	ZK [11]

multiple variables or time-out mechanisms enforced by monitoring threads. (4) `RDTSCP` time stamp may cause inaccurate write-read and signal-wait pairing in FCatch. In practice, replay researchers have found it sufficiently accurate [70], which matches our experience. (5) FCatch selectively traces heap accesses for scalability concerns. Although neither sound nor complete, FCatch provides a good trade off in effectively detecting real-world fault timing bugs with good accuracy in real-world large distributed systems, as shown by our evaluation.

5.7 Methodology

Benchmark software We evaluate FCatch on six common workloads in four widely used open-source distributed systems (Table 5.1), MapReduce distributed computing framework (MR); Cassandra distributed key-value stores (CA); HBase distributed key-value stores (HB); ZooKeeper distributed synchronization service (ZK), with 172K – 1,651K lines of code.

Benchmark bugs We obtain these workloads from TaxDC benchmark suite [51]: real-world users reported 7 fault timing bugs, listed in Table 5.1, that could be triggered when specially-timed faults occur during these workloads. As the upper half of Table 5.2 shows, these 7 bugs cover both types of fault timing bugs, a variety of contention resources like heap objects, files, and more, and a variety of failure symptoms, including system hangs, data losses, and job/system failures. These failures are all non-recoverable by these distributed systems — users/administrators have to re-submit the job or restart the system.

Evaluation Workload The 6 workloads under evaluation are all common ones, as shown in Table 5.1. Note that, the systems can tolerate *most* of the faults that occur while running these workloads. Our evaluation will show that these workloads can correctly finish even with hundreds of times' tries of randomly injected faults. FCatch detects fault timing bugs by monitoring **correct** runs.

As discussed earlier, to detect Crash-Recovery fault timing bugs, we randomly inject a node crash to trace recovery operations. To measure how sensitive FCatch is towards the time of fault injection, we did a sensitivity study. That is, for every benchmark, we apply FCatch three times, with faults injected at three different execution moments — near the beginning, in the middle, and near the end of the execution. We then check whether FCatch produces different bug reports among three different bug-detection attempts.

Experiment setting We run each benchmark in one virtual machine with VirtualBox 4.3.36. We use `VBoxManage` commands to take snapshot and restore each VM. Both host machine and guest OS in VM use Ubuntu 14.04 and JVM v1.7. Host machine has Intel® Xeon® CPU E5-2620 and 96GB of RAM. All trace analysis and pruning are on host machine.

Evaluation metrics We will evaluate FCatch from several aspects: the coverage and accuracy of its bug detection, the overhead of its run-time and static analysis, and how it compares with alternative designs and random fault injection.

All the performance numbers are based on an average of 3 runs during **correct** runs, with the baseline performance measured with **no** sleep inserted.

5.8 Experimental Results

5.8.1 Bug Detection Results

FCatch issues 31 fault timing bug reports by analyzing correct runs:

Table 5.2: fault timing bugs found by FCatch. Res.: resources in contention; H: heap object; ZK: ZKnode; GF: global file; LF: local file.

ID	Operations	Res.	Symptom
Benchmark Crash-Regular fault timing bugs			
CA1	Signal vs Wait	H	AE hangs @ Snapshot
CA2	Signal vs Wait	H	AE hangs @ Mtree compare
HB1	Write vs Loop	H	HMaster hangs @ MetaOpen (Fig.5.6)
Benchmark Crash-Recovery fault timing bugs			
HB2	Create vs Create	ZK	Data loss as Get lock fail
MR1	Write vs Read	H	Task recovery hangs (Fig. 5.1)
MR2	Delete vs Open	GF	AM restart fails as Dir. deleted
ZK	Write vs Read	LF	Restart fails
Non-Benchmark Crash-Regular fault timing bugs			
CA3	Write vs Loop	H	AE hangs @ Mtree repair
HB3	Signal vs Wait	H	HMaster hangs @ ROOT open
HB4	Write vs Loop	H	HMaster hangs @ ROOT open
MR3	Signal vs Wait	H	Hangs @ Any RPC call
Non-Benchmark Crash-Recovery fault timing bugs			
HB5	Delete vs Read	ZK	Data loss as HLog skipped
HB6	Delete vs Read	ZK	Data loss as HLog dir. skipped
MR4	Write vs Read	H	Task recovery killed
MR5	Create vs Exists	GF	AM restart fails as Flag-file exists

- 8 are severe bugs explaining all the 7 benchmark issues;
- 8 are severe bugs that we were unaware of[3];
- 6 are false positives that cause benign exceptions;
- 9 are false positives that are incorrectly reported.

FCatch is effective in accurately predicting fault timing bugs.

3. These 8 bugs are not part of existing benchmark suites [51, 31]. After carefully checking software change logs, we later found that (1) 4 bugs have been acknowledged and fixed by developers, and (2) the other 4 have never been reported, but have disappeared in latest versions due to software functionality changes. More details about these bugs, including their reproduction scripts, are available at `https://github.com/haopeng-liu/TOF-bugs`.

Crash-Regular fault timing bugs detecting

As shown in Table 5.3, FCatch not only correctly predicts 3 crash-regular benchmark fault timing bugs (CA1, CA2, and HB1), but also finds 4 non-benchmark harmful fault timing bugs (CA3, HB3, HB4, and MR3 in Table 5.2), with only 5 false positives across all benchmarks.

These four bugs are severe, as shown in Table 5.2. For example, MR3 affects *all* RPC calling. It can be triggered by either a crash of the RPC caller or a drop of an RPC-return message, and causes the RPC caller to hang forever. As another example, HB3 and HB4 can both cause the whole HBase system to hang. We have triggered all these bugs.

False positives The 5 false positive crash-regular fault timing bug reports fall into three categories. Two of them, both in HB2, indeed cause HMaster to hang. However, this is an expected behavior: when all region servers crash during registration, HMaster is expected to stuck in a loop waiting for at least one region server to come alive. One false positive in HB1 is caused by incorrectly identified custom loop-signal operation. The remaining two false positives in HB1 are caused by un-recognized time-out mechanisms: some components in the distributed system did hang, but another watcher-component in the system discovers and terminates the hang.

Crash-Recovery fault timing bugs detecting

As shown in Table 5.3, FCatch not only correctly predicts 4 crash-recovery benchmark fault timing bugs, including 2 fault timing bugs that represent two ways to trigger MR2, but also finds 4 harmful fault timing bugs we were unaware of, with two in MR and two in HB.

These 4 bugs are severe, as shown in Table 5.2, and have all been triggered by us. For example, HB5 and HB6 can cause HLogs or HLog directories to be skipped during HLog replication, which could then cause silent data loss in HBase.

Table 5.3: FCatch bug detection results. *: same bug; Old: benchmark bugs; - : benchmark bug is not of this category; Exp.: false positives that throw exceptions.

	Crash–Regular				Crash–Recovery			
	Bugs		False		Bugs		False	
	Old	New	Exp.		Old	New	Exp.	
CA1&2	2	1	0	0	-	0	0	2
HB1	1	0	0	3	-	0	4	2
HB2	-	2	2	0	1	2	0	0
MR1	-	1*	0	0	1	1	0	0
MR2	-	1*	0	0	2	1	0	0
ZK	-	0	0	0	1	0	0	2
Total	3	4	2	3	5	4	4	6

False positives The 10 false positives of FCatch crash-recovery bug detection falls into two categories. For 4 of them (4 in HB-1), the reported {W, R} pairs can indeed cause exceptions, like `ZKNodeNotExistException` and `FileAlreadyExistsException`, when a fault happens at the reported moment. However, these exceptions are well handled. For example, when HBase' recovery routine finds that a temporary file F it plans to create already exists (i.e., left by the crashing node), it simply creates a temporary file F' with a different name. For the other 6 false positives, although the reported fault timing can indeed cause a recovery operation to consume different content from a shared resource from what it used to consume in correct runs, the difference is valid and does not lead to exceptions or failures.

Crash point sensitivity To detect crash-regular fault timing bugs, FCatch only analyzes a fault-free run; to detect crash-recovery fault timing bugs, FCatch analyzes not only a fault-free run but also a faulty run. Therefore, we evaluate whether the bug-detection results of FCatch is sensitive to where the fault occurs during the correct faulty run. Specifically, for each benchmark, we tried random fault-injection during three phases of the execution to collect the correct faulty-run traces: at the beginning (all other results in this chapter comes from this setting), in the middle, and near the end.

Table 5.4: FCatch performance.

	Baseline		Tracing		Analysis		Overall	
	NF	F	NF	F	Reg	Rec	Time	Slowdown
CA1&2	17.5	25.4	36	59	9	104	208	11.9X
HB1	40.6	84.3	191	233	43	124	592	14.5X
HB2	14.5	54.4	67	125	10	19	221	15.2X
MR1	64.3	78.9	158	180	24	15	376	5.9X
MR2	68.4	114	274	284	21	93	672	9.8X
ZK	10.3	14.7	12	31	6	8	58	5.6X

Our evaluation found that, the crash-recovery fault timing bugs detected by FCatch are **not** sensitive to the fault location. No matter the fault occurs at the beginning or in the middle, FCatch reports exactly the same 16 severe bugs listed in Table 5.3. The only difference among the three sets of experiments is that, while injecting faults near the end of the execution, FCatch misses 2 bug reports, 1 in MR and 1 in ZK (FCatch actually misses 2 bug reports in MR1 benchmark, but one of them is reported by MR2 benchmark). These two bug reports are missed because when a node crashes, it has finished most of the work and hence the corresponding recovery routine conducted fewer operations comparing with other settings.

This evaluation shows that FCatch did not predict those bugs in Table 5.3 by luck and is much less sensitive to the timing of fault than traditional fault-injection testing.

5.8.2 Performance Evaluation

Table 5.4 presents performance breakdowns of FCatch bug detection. Overall, FCatch is fast enough for in-house testing. Comparing with running a benchmark **once** with **no** fault **no** instrumentation **no** sleep injected, FCatch finishes its fault timing bug detection with 5.6x – 15.2x slowdown. In fact, FCatch only imposes 3.8x – 8.2x slowdown for correct faulty runs.

For most benchmarks, FCatch spent more time in tracing than doing trace analysis. A key factor for the tracing overhead is that we use Javassist, a dynamic instrumentation tool, to inject tracing code. The performance would be better if we use a static code transformation tool. Trace analysis is mostly fast. The most time-consuming part in trace analysis to check data-dependency based fault tolerance mechanism.

The size of the traces produced by FCatch ranges from 9MB (ZK) to about 450 MB (MR). The trace size does not directly affect FCatch trace analysis time, as larger traces do not always contain more conflicting operations.

The performance of FCatch greatly benefits from its selective, instead of exhaustive, heap-access monitoring (Chapter 5.3.2). To demonstrate that, we also measured the run-time overhead of tracing *all* heap accesses. Our evaluation shows that CA benchmarks simply cannot finish, as the huge tracing overhead affects gossip protocols and causes nodes to treat neighbors as dead; in MR benchmarks, mappers finish only 20% and reducers 0% after 20 minutes of execution (original FCatch finishes everything in 1–2 minutes, as shown in Table 5.4); HB benchmarks finish with 51X slowdowns; ZK starts up successfully but incurs 12X slowdowns. Overall, tracing all heap accesses is impractical for applying FCatch to large real-world distributed systems.

5.8.3 Random Crash Injection

We also compare FCatch with random fault injection. Specifically, our fault-injection script runs each workload for 400 times, and in every run picks a random time point to crash a node. We then analyze the log to see if any bugs is exposed.

Performance As shown in Table 5.4, comparing with 1 fault-injection run, FCatch incurs about 4x slowdown to HB2, MR, and ZK, and 7x–8x slowdown to CA and HB1.

Bug detection 400 runs of random fault-injection is much less effective than 1 round of FCatch— FCatch finds 16 unique true bugs, yet random fault-injection finds only 2. One is benchmark CA2, exposed 10 times in 400 runs. CA2 can be triggered if a node crashes in the

Table 5.5: # false positives pruned by different analysis

	Crash–Regular		Crash–Recovery	
	Loop TimeOut	Wait TimeOut	Dependence	Impact
CA1&2	0	1	54	194
HB1	3	7	97	115
HB2	0	2	14	40
MR1	0	1	2	4
MR2	0	2	19	38
ZK	2	2	35	40

middle of taking snapshot. Since taking snapshot takes time, CA2 has a decent probability (∼2.5%) to be triggered by random fault injection. The other bug is in MR, a false negative of FCatch. It causes the system to hang while AM waits for a heartbeat change yet the node that can make the change crashes. FCatch missed this bug, because it traces only heap accesses inside event/message/RPC handlers. Unfortunately, the W operation in this bug is not inside such handlers, and instead has data dependence on its return value.

We did not try other fault-injection schemes, as it is difficult to do better than random injection without sophisticated analysis or modeling. One possible alternative is to only inject faults around message sending and receiving. However, since many fault timing bugs (7 out of 16 fault timing bugs reported by FCatch) require faults between specific memory/file accesses to manifest, injecting faults around message operations has no chances to expose them. Furthermore, many of our benchmark executions contain more than 1000 messages, and hence still leave a huge search space for fault injection around message operations.

5.8.4 Pruning and Triggering

False positive pruning Table 5.5 shows that the fault-tolerance analysis (i.e., time-out analysis for crash-regular bugs in Chapter 5.4.2 and dependence analysis for crash-recovery bugs in Chapter 5.4.3) and the impact estimation (Chapter 5.4.3) greatly improve FCatch accuracy. Without them, the number of false positives will increase by about 5X for crash-regular bugs and about 40X for crash-recovery bugs.

Table 5.5 shows fewer crash-regular false positives pruned than those crash-recovery false positives pruned. A main reason is that before applying any static pruning analysis, FCatch already quires the conflicting operations W and R to satisfy blocking happens-before relationship in crash-regular fault timing bug detection, and consequently leaves fewer false positives to be pruned by static analysis shown in Table 5.5.

Naively reporting all synchronization loops or file operations without existence checks as fault timing bugs are naturally very inaccurate. We skip the numbers for space constraints.

Of course, our analysis could prune out true bugs. We randomly checked 10 pruned bug reports from each category. Among them, all are indeed false-positives.

Triggering The triggering component of FCatch, as discussed in Chapter 5.5, can easily trigger each reported bug and tell whether the bug report is false positive or not. While trying three different types of faults to trigger crash-regular fault timing bug reports, we have sometimes experienced different impact of different faults. For example, the HB1 benchmark bug can only be triggered by a node-crash fault, but not by a kernel-level message drop fault, as the latter is handled by HBase through message resend. On the other hand, 2 out of the 3 crash-regular bugs from CA can be triggered by kernel-level message drops, but not node crashed, as the latter is handled by crash-recovery mechanisms in CA. For all other true crash-regular fault timing bugs in MR and HB2, they can be triggered by both node crashes and message drops.

5.9 Conclusion

Fault timing bugs are a unique type of bugs for distributed systems. They all severely hurt the system availability, because they do not have the fault-tolerance safety net as many other bugs. Different from traditional fault-injection approach, our work explores a new perspective to model and detect fault timing bugs. Our evaluation shows that FCatch can indeed effectively detect fault timing bugs with high accuracy and decent overhead.

CHAPTER 6

HFIX: GENERATING HIGH QUALITY PATCHES FOR TIMING BUGS IN MULTI-THREADED SYSTEMS

Concurrency bugs in multi-threaded programs are time-consuming to fix correctly by developers and a severe threat to software reliability. Although many auto-fixing techniques have been proposed recently for concurrency bugs, there is still a big gap between the quality of automatically generated patches and manually designed ones as it is shown in Chapter 3.2. Motivated by the study findings, a new tool HFix [55] is designed. It can automatically generate patches, which have matching quality as manual patches, for many concurrency bugs.

6.1 Introduction

6.1.1 Motivation

Concurrency bugs are caused by synchronization problems in multi-threaded software. They have caused real-world disasters [52, 83] in the past, and are a severe threat to software reliability with the pervasive use of multi-threaded software. The unique non-determinism nature has made them difficult to avoid, detect, diagnose, and fix by developers. Previous studies of open-source software [64] have shown that it often takes developers several months to correctly fix concurrency bugs. Furthermore, concurrency bugs are the most difficult to fix correctly among common bug types [97], with many incorrect patches released. Consequently, automated tools that help fix real-world concurrency bugs are well desired.

Recently, many automated fixing techniques have been proposed. Some are for general bugs [20, 43, 45, 47, 49, 62], and some are for specific type of bugs [23, 28, 81]. Particularly, several tools dedicated to concurrency bugs have been proposed [19, 40, 42, 58, 59, 60, 90]. Existing concurrency-bug fixing tools can handle all common types of concurrency bugs

```
//child thread                     //parent thread
if(...){                           + thread_join(...);
  unlock(fifo->mut); //A             if(...){
  return;                              fifo->mut = NULL; //B1
}                                    }
                                     fifo = NULL; //B2
```

a. Manual and HFix patch

```
  //child thread                   //parent thread
  if(...){                         + lock(L);
    unlock(fifo->mut); //A         + while(...){
+   lock(L);                       +   wait(con, L);
+   signal(con);                   + }
+   unlock(L);                     + unlock(L);
    return;                          if(...){
  }                                    fifo->mut = NULL; //B1
+ lock(L);                           }
+ signal(con);                       fifo = NULL; //B2
+ unlock(L);
```

b. Simplified CFix patch

Figure 6.1: A simplified concurrency bug in PBZIP2.

leveraging a unique property of concurrency bugs — since concurrency bugs manifest non-deterministically, the correct computation semantics already exist in software. Consequently, these tools work not by changing computation semantics, which is required for most non-concurrency-bug fixing, but by adding constraints to software timing. They mostly achieve this by adding synchronization operations, including locks [40, 58, 60, 90] and condition variable signal/waits [42], into software.

However, as shown in our empirical study, the big gap between automatically generated patches and manually generated patches makes it appealing to design auto-fixing tools that generate not only correct but also simple and well-performing patches.

6.1.2 Contributions

Guided by this study, we build a new bug fixing tool HFix that can automatically generate patches with matching quality as manual patches for many concurrency bugs. HFix can fix many real-world concurrency bugs in large software by two strategies that have not been well explored before.

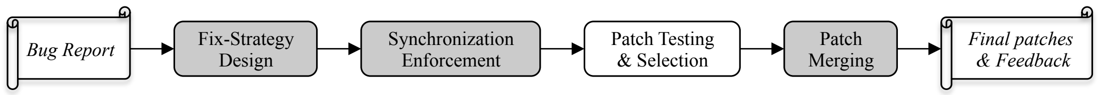


Figure 6.2: General bug fixing process

The first, HFix$_{join}$, enforces ordering relationship by adding thread-join operations, instead of signal/waits (Chapter 6.3). HFix$_{join}$ can produce simple patches.

The second, HFix$_{move}$, enforces ordering and atomicity relationship by leveraging synchronization operations that already exist in software (Chapter 6.4).

HFix works for a wide variety of concurrency bugs. Evaluation using real-world bugs shows that it can indeed automatically generate patches that have matching quality with manual patches and are much simpler than those generated by previous state of the art technique (Chapter 6.6).

6.2 Bug Fix Background and Overview

HFix reuses the general bug-fixing framework proposed by CFix [42] (Figure 6.2). The inputs to the bug-fixing framework are bug reports, which can be automatically generated by bug detection tools [53, 77, 82, 101, 102]. Given a bug report, some checkings are conducted to see which fix strategies might be suitable for the bug (Fix-Strategy Design). Then, program analysis and code transformation are conducted to enforce the desired synchronization following the fix strategy (Synchronization Enforcement). After that, the generated patches go through testing and merging, with final patches produced.

HFix will modify and extend three key components of the fixing framework, highlighted by gray background in Figure 6.2. That is, different fix strategies will be considered; different types of program analysis and code transformation will be conducted following the different fix strategies; finally, the patch merging will also be different.

HFix reuses some CFix techniques. Specifically, HFix reuses the function cloning technique used in CFix when it generates patches that take effect under specific calling contexts. CFix makes best effort to avoid introducing deadlocks, but does not prove deadlock free. Instead, it provides the option to instrument patches for light-weight patch-deadlock monitoring, which is reused in HFix.

HFix reuses an important philosophy of CFix — bug fixing works only when correct bug reports are provided. HFix re-uses the bug-report format requirement of CFix. An AV report needs to specify three statements, p, c, and r. As discussed in Chapter 3.2.1, an AV bug manifests when one thread unexpectedly executes r between another thread's execution of p and c. An OV report needs to specify two statements, A and B. As also discussed in Chapter 3.2.1, the OV bug happens when A unexpectedly executes after, instead of before, B. Since OV bugs are sometimes context sensitive, CFix also requires an OV bug report to contain the calling contexts of A and B, including (1) a call stack that executes the buggy instruction in the thread, and (2) a chain of call stacks (referred as thread stack) indicating how that thread has been created. For an instruction i, we call its call stack and thread stack together as *stack* of i, and we call the thread that executes i as thread of i, or $\texttt{thread}_i$.

6.3 HFix$_{join}$

A thread join operation (i.e., `join`), such as `pthread_join`, can enforce all operations after `join` in a parent thread to wait for all operations in the joined thread. Adding `join` is a common way to fix OV bugs, as discussed in Chapter 3.2. This Chapter presents HFix$_{join}$ that automatically identifies suitable OV bugs and fixes them through Add$_{\text{join}}$ strategy.

HFix$_{join}$ takes as input the OV bug report that includes the stack of A and the stack of B with A expected to execute before B, as defined in Chapter 6.2. HFix$_{join}$ will then go through two main steps.

1. Suitability checking (Chapter 6.3.1). Not all OV bugs can be fixed by adding `join`. HFix checks whether $\texttt{thread}_A$ is a never-joined child of $\texttt{thread}_B$ and other conditions

to decide whether Add_{join} is a suitable strategy for the given bug.

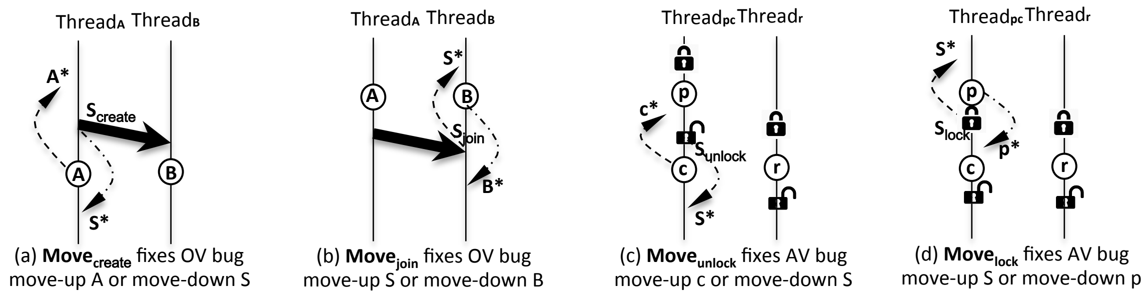
2. Patching (Chapter 6.3.2). Once the suitability is decided, HFix conducts code transformation to fix the bug.

6.3.1 Patch Suitability Checking

Is `thread`$_A$ a joinable child thread of `thread`$_B$? HFix patches follow the common practice and only join a thread from its parent thread. To achieve this, HFix checks whether `thread`$_A$ is a child of `thread`$_B$ by examining the stacks of A and B. From the stack of A, HFix identifies the statement C that creates `thread`$_A$, and then easily tells whether C comes from `thread`$_B$ by comparing the stacks. In addition, HFix checks the stack of B to make sure there will be only one instance of `thread`$_B$ (i.e., HFix checks to make sure that the thread-creation statements are not in loops). Otherwise, inserting `join` cannot guarantee every instance of B to wait for all instances of A.

Is `thread`$_A$ already joined? When the software already contains a `join` for `thread`$_A$, adding an extra `join` is often not a good strategy. This analysis goes through two steps. We first go through all functions in software to identify `join` statements. For every existing `join`, we then check whether it is joining `thread`$_A$. In our implementation, this is done by checking whether the first parameter of the `pthread_create` statement for `thread`$_A$ may point to the first parameter of the `pthread_join` under study.

Will there be deadlock? An added `join` will force B to wait for not only A, but also all operations following A in `thread`$_A$. This extra synchronization is not required by bug fixing. Therefore, we need to check whether it may lead to deadlocks or severe performance slowdown. Specifically, we conduct inter-procedural analysis to see if any blocking operations (i.e., `pthread_cond_wait`, `pthread_join`, and `pthread_mutex_lock` in our implementation) may execute after A in thread-A. If any of these are found, HFix aborts the Add_{join} fix strategy, as adding `join` will bring a risk of potential deadlocks and/or severe performance



ThreadA ThreadB
A*
Screate
A
B
S*
(a) Movecreate fixes OV bug move-up A or move-down S
ThreadA ThreadB
S*
A
B
Sjoin
B*
(b) Movejoin fixes OV bug move-up S or move-down B
Threadpc Threadr
p
c*
Sunlock
c
r
S*
(c) Moveunlock fixes AV bug move-up c or move-down S
Threadpc Threadr
S*
p
Slock
p*
c
r
(d) Movelock fixes AV bug move-up S or move-down p

Figure 6.3: Fixing OV timing bugs and AV bugs through Move

slowdowns. Furthermore, HFix also aborts the patch if B is inside a critical section, because-inserting a blocking operation (i.e., `join`) inside a critical section may lead to deadlocks.

—

6.3.2 Patch Generation

To generate an Add$_{\text{join}}$ patch, we need to first decide the location of the new `join` — right before B. If B is inside a loop, we will use a flag to make sure that the `join` executes only once. Note that, an alternative and simpler patch is to insert the `join` before the loop, which does not require any flags. The current implementation of HFix does not take this option, fearing that the early execution of `join` may hurt performance or introduce deadlocks.

The parameter of `join` needs to contain an object that represents the child thread (i.e., the `thread_t` object returned by `pthread_create` in POSIX standard). To obtain this object, our patch creates a global vector; pushes every newly created `thread_t` object into this vector right after an instance of $\texttt{thread}_A$ is created; and invokes `join` for each object in the vector right before B.

Our current implementation targets POSIX standard and POSIX functions. Small modifications can make HFix work for non-POSIX, customized synchronization operations.

6.4 HFix$_{move}$

As shown in Chapter 3.2 and Table 3.3, many concurrency bugs, including both AV bugs and OV bugs, can be fixed by leveraging existing synchronization in software, instead of adding new synchronization. Specifically, a Move patch re-arranges the placement of memory-access statements, which are involved in a concurrency bug, and synchronization operations, which already exist in the same thread, so that the memory-access statements will be better synchronized, as illustrated in Figure 6.3. This chapter will present HFix$_{move}$ that automates the Move fix strategy.

6.4.1 Overview of HFix$_{move}$

Applying a Move patch is more complicated than applying an Add$_{\text{join}}$ patch. The suitability checking and patch generation are conducted together in four steps.

1. Identify two operations in one thread, so that flipping their execution order can fix the reported bug. How to conduct this step varies depending on the type of the bug and the type of synchronization nearby.

2. Control flow checking. We need to make sure the movement does not break control dependencies, causing a statement to execute for more or fewer times than it should be. At the end of this step, a candidate patch will be generated and go through the next two steps.

3. Data flow checking. We need to make sure the movement does not break existing define-use data dependency within one thread, causing the patched software to deviate from the expected program semantic.

4. Deadlock and performance checking. We need to check whether the movement could bring risks of deadlocks or severe performance slowdowns.

6.4.2 Identifying Move Opportunities

Move$_{\textbf{join}}$ opportunities for OV bugs Since `join` can enforce ordering between operations in parent and child threads, we can leverage existing `join` to fix OV bugs, as shown in Figure 6.3b. Given an OV bug (AB), we check whether $\texttt{thread}_A$ is `join`-ed by $\texttt{thread}_B$ in the buggy software. If such a `join` exists, we know that this bug can potentially be fixed by moving B after the `join` (Move-Down) or moving the `join` before B (Move-Up), as shown in Figure 6.3b. Of course, if we want to make sure all instances of A will execute before B, we need to check the stack of B to make sure there is only one dynamic instance of $\texttt{thread}_B$.

Move$_{\textbf{create}}$ opportunities for OV bugs A thread-creation operation, denoted as `create`, forces all operations before `create` inside the parent thread to execute before all operations inside the child thread. Consequently, `create` can be leveraged to fix some OV bugs, as shown in Figure 6.3a.

Specifically, given an AB order violation bug, we will check the stack of B to see if $\texttt{thread}_B$ is created by $\texttt{thread}_A$. If so, a Move$_{\text{create}}$ opportunity is identified: the bug can potentially be fixed by moving A to execute before the `create` (Move-Up) or moving the corresponding `create` to execute after A (Move-Down). Of course, like that in Move$_{\text{join}}$, if the patch wants to force all dynamic instances of A to execute before B, we also need to check the stack of A to make sure that there could be only one dynamic instance of $\texttt{thread}_A$.

Move$_{\textbf{lock}}$ and Move$_{\textbf{unlock}}$ opportunities for AV bugs Given an AV bug report p-c-r, the patch needs to provide mutual exclusion between the p–c code region and r. If r and part of the p–c code region are already protected by a common lock, the patch can leverage existing `lock` or `unlock`, as illustrated by Figure 6.3d and Figure 6.3c.

Specifically, HFix identifies Move$_{\text{lock}}$ or Move$_{\text{unlock}}$ opportunities for an AV bug in two cases. In the first case, p and r are inside critical sections of lock l, but c is not. As shown in Figure 6.3c, this type of bugs can potentially be fixed by re-ordering c and a corresponding

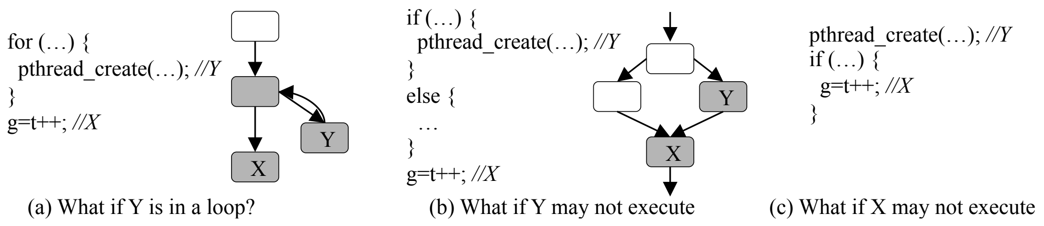


Figure 6.4: Control flow challenge for code movement.

`unlock` operation (Move$_{\text{unlock}}$[1]). In the second case, c and r are inside critical sections of lock l, but p is not. As shown in Figure 6.3d, this type of bugs can potentially be fixed by re-ordering p and a corresponding `lock` operation. Below, we describe our analysis algorithm focusing on the Move$_{\text{unlock}}$ case. The algorithm for Move$_{\text{lock}}$ patches is similar.

To identify the above two cases, we first identify all the critical sections that contain p, c, and r, respectively, and then compare the locks that are used for these critical sections. Note that, accurately identifying all enclosing critical sections and comparing lock sets are very challenging, as it involves inter-procedural and pointer alias analysis. Our current implementation only considers critical sections that are in the same function as p, or c, or r, not in their callers, and are protected by global locks, not heap locks. This way, although we may miss some fix opportunities, we keep our analysis simple and accurate.

Generalize Move$_{\textbf{join}}$, Move$_{\textbf{create}}$, Move$_{\textbf{lock}}$, Move$_{\textbf{unlock}}$ For the ease of discussion, we generalize all these strategies, and use the following terms and symbols in the remainder of this chapter. We use *Move-Up* to refer to patches that make an operation execute earlier by moving it up in its thread, denoted by up-pointing dashed arrows in Figure 6.3; we use *Move-Down* to refer to patches that make an operation execute later by moving it down in its thread, denoted by down-pointing dashed arrows in Figure 6.3.

We use X to denote the operation in the original program that gets moved; use $X*$ to denote its counter-part in the patched program; and use Y to denote the operation that X needs to move around in order to fix the bug. For example, Figure 6.3a demonstrates

1. Sometimes, we use Move$_{\text{lock}}$ to represent both lock-moving patches and unlock-moving patches.

Move$_{\text{create}}$ strategy. In the Move-Up version of a Move$_{\text{create}}$ patch, X represents operation A, and Y represents synchronization S (i.e. `create`); in the Move-Down version of Move$_{\text{create}}$, X represents synchronization S and Y represents operation A.

6.4.3 Control Flow Checking

We conduct several rounds of control flow checking to identify the correct location of X^*, while make sure that X^* will execute for exactly the same number of times as X.

First, check if X and Y are inside the same function. We do not want to make the movement when X and Y are inside different functions, as that would break the original code encapsulation. For the cases of Move$_{\text{join}}$ and Move$_{\text{create}}$, when X and Y are inside different functions, we find a common function f on their call stacks, change the operations under interests from X and Y to their call sites in f, and then conduct all the control-flow analysis and movement inside one function, f. For example, for the bug shown in Figure 3.2, the two operations that we initially want to reorder are `pthread_create` inside function `tr_eventInit` and the write access to `h`$\rightarrow$ `band` inside function `tr_sessionInit`. Since they are not inside the same function, we instead identify their common caller `tr_sessionInit`, and shift our focus to flipping the order between `tr_eventInit` and the write access. The same strategy cannot be directly applied to Move$_{\text{lock}}$ or Move$_{\text{unlock}}$. Imagine we want to move a `lock` operation in function f_1 so that the corresponding critical section can be extended to contain not only c in f_1 but also p in function f_0, the caller of f_1. We cannot simply move the callsite of f_1 in f_0, because that would likely move the whole critical section inside f_1 and still leave p outside the critical section. Consequently, HFix simply makes a clone of the callee function, makes the cloned function inlined, and then conducts code movement inside one function.

Second, check if X is inside a loop. If it is, we abort the fix. The reason is that it would be too difficult to move X without changing the number of times X executes.

Third, identify the location X^*. Note that, this is easy when X and Y are inside the same basic block — X^* would be simply right after or before Y. However, in reality, there could be several challenges. For example, as demonstrated in Figure 6.4a, if Y is inside a loop, simply moving X to be right after or before it could cause X^* to execute for many more times in the patched program than X does in the original program. More importantly, this movement cannot guarantee that X will execute before *all* instances of Y, a property that is needed to fix many OV bugs (e.g., we may want a variable to be initialized before any child thread can read it). As another example demonstrated in Figure 6.4b, when X and Y are not inside the same basic block, a naive movement could also cause X^* to execute for fewer times in the patched program than X does in the original program.

Our algorithm addresses all these challenges. When we want to move X up to execute before all instances of Y, we will do the following. We first identify all control-flow graph nodes in function f that are reachable from Y, referred to as `AfterY` nodes, as shown by the gray boxes in Figure 6.4. We then delete X, and make X^* appear on every edge that goes from a non-`AfterY` node n to an `AfterY` node n_2. This arrangement guarantees that X^* will execute before all instances of Y, because otherwise n would have been an `AfterY` node; it also guarantees that X^* will execute at most once in its function, because once the execution goes from non-`AfterY` nodes to `AfterY` nodes, it can never go back to nodes non-reachable from Y before the function exits.

When we want to move X down to execute after all instances of Y, the handling is similar. We first identify all control-flow graph nodes inside function f that can reach Y, referred to as `BeforeY` nodes. We then delete X, and place X^* on every edge that goes from a `BeforeY` node n to a non-`BeforeY` node n_2. Similarly, this arrangement guarantees that that X^* will execute at most once in function f and will execute before all instances of Y.

Finally, we need to make sure X^* will execute for the same number of times X does. Note that, since X is not inside a loop, it will execute at most once. Our arrangement above guarantees the same property for X^*. In case of Move-Down, our patch would insert a flag setting statement in the original location of X, and a flag checking before every X^*. This way, we guarantee that X^* is executed only when the original X would have been executed. The Move-Up case could be complicated, as illustrated in Figure 6.4c. When X^* is executed, we may not be able to predict whether X would be executed or not later. If later execution decides to bypass X, we will have to revert the effect of X^*. The revert could be impossible within one thread, if X^* creates a thread or writes to a shared variable. Given this complexity, in our implementation, we apply the Move-Up strategy only when (1) X executes exactly once in function f; or (2) the execution of X does not require revert or can be reverted easily (e.g., some moved-up `join` does not require revert; some moved-up `lock` can be easily reverted). The complexity of the above analysis is linear to the size of the control flow graph of function f.

6.4.4 Data Flow Checking

We use data flow analysis to check whether the code movement could affect thread-local data dependency and mistakenly change program semantics.

Specifically, we identify all instructions I that might execute between the original location of X and the new location of X^*, and check if they might access the same memory locations as X. We abort the patch, if either (1) I may access a memory location that is written by X or (2) I may write to a memory location that is read by X. In these cases, the code movement could break the write-after-write, write-after-read or read-after-write dependency in the original program, incorrectly changing program semantics, and hence should not be used in bug fixing. In our current prototype, we use the default MAY-pointer-alias analysis in LLVM to decide whether two sets of instructions may read/write the same memory object. We consider `create`, `join`, and `(un)lock` functions to only access its parameter objects.

6.4.5 Deadlock and Performance Checking

To avoid introducing deadlocks or severe performance slowdowns into the program, we also check and abort some patches to minimize the risk of introducing circular waits among threads. For $\text{Move}_{\text{create}}$, we make sure the patch does not push extra blocking operations to execute before `create`; for $\text{Move}_{\text{join}}$, we make sure the patches does not delay some unblock operations to execute after `join`. In case of $\text{Move}_{\text{lock}}$ and $\text{Move}_{\text{unlock}}$, we make sure not to move any blocking operations into a critical section. In our implementation, the black list of blocking operations includes lock, condition-variable wait, and thread-join operations.

Summary HFix_{move} is sound but not complete. Its patches are guaranteed not to introduce new bugs that violate control dependency or thread-local data dependency of the original software, as discussed in Chapter 6.4.3 and 6.4.4. HFix_{move} guarantees not to introduce deadlocks, as long as its list of (un)blocking operations is complete. However, HFix_{move} may miss the opportunity to generate Move patches for corner-case Move-suitable bugs, as we will see in our evaluation (Chapter 6.6.2).

6.5 Patch Merging

A single synchronization mistake, such as forgetting to join a child thread, can often lead to multiple related bug reports. Fixing these related bugs separately would hurt the patch simplicity and performance. For example, bug detectors report five highly related OV bugs in PBZIP2. Naively fixing these bugs separately would add five `join` within a few lines of code, which is unnecessarily complicated.

Fortunately, both Add_{join} and Move strategies are naturally suitable for patch merging. For example, if multiple OV bugs between a parent thread and a child thread are reported, we may fix one bug through Add_{join}, and the remaining ones through $\text{Move}_{\text{join}}$, leveraging the newly added `join`. We could also fix each bug report separately, and then analyze the patches to see if we can merge them, which is exactly what we have implemented.

We only merge patches with the same fix strategy. We only discuss how to merge two patches below; merging more than two patches is similar. We do not discuss how to merge $\text{Move}_{\text{lock}}$ patches below, because it can be addressed by merging technique proposed in previous work [42].

6.5.1 Merge Add_{join} Patches

Imagine that two Add_{join} patches are generated for two OV bugs A_1B_1 and A_2B_2. Our merger explores merging these two patches if A_1 and A_2 are from the same thread, and B_1 and B_2 are from the same thread. This checking is conducted based on the stack of A_1, A_2, B_1, and B_2.

If the two bugs/patches pass the first checking, our merger will try to create a merged patch. That is, the merger tries to decide where to add `join` in the merged patch. We use j_1 to denote the location of the added `join` in the patch for A_1B_1 (i.e., right before B_1 in Figure 6.1a), and j_2 to denote the location of the added `join` in the patch for A_2B_2 (i.e., right before B_2 in Figure 6.1a). The merger will identify the nearest-common-dominator of j_1 and j_2, denoted as j_{12}, and put the `join` there in the merged patch. This merged patch can guarantee to fix both A_1B_1 and A_2B_2 OV bugs, because B_1 and B_2 are guaranteed to execute after their common dominator `join` j_{12}, which in turn is guaranteed to execute after A_1 and A_2.

To avoid introducing deadlocks or severe performance slowndowns, the merger stops the merging attempt if there exist any signal or unlock operations along the paths that connect j_{12} and j_1, and j_{12} and j_2. Note that, we decide to put the merged `join` at the nearest, instead of any, common-dominator, exactly because we want to minimize the risk of introducing deadlocks or severe perforfance slowdowns.

BugID	App.	HFix		Manual		CFix	
		Strategy	#Sync	Strategy	#Sync	Strategy	#Sync
OV1	FFT	A $_{\text{join}}$	1j	A $_{\text{join}}$	1j	A $_{\text{s.w.}}$	4s,1w
OV2	HTTrack-20247	-	-	-	-	A $_{\text{s.w.}}$	1s,2w
OV3	Mozilla-61369	-	-	A $_{\text{lock}}$	2l,2u	A $_{\text{s.w.}}$	1s,1w
OV4	PBZIP2	A $_{\text{join}}$	1j	A $_{\text{join}}$	1j	A $_{\text{s.w.}}$	4s,1w
OV5	Transmission-1818	M_{create}	0	M_{create}	0	A $_{\text{s.w.}}$	1s,1w
OV6	X264	M_{join}	0	M_{join}	0	A $_{\text{s.w.}}$	6s,1w
OV7	ZSNES-10918	M_{create}	0	-	-	A $_{\text{s.w.}}$	2s,1w

Table 6.1: Patch comparison for OV bugs

6.5.2 Merge Move$_{join}$ and Move$_{create}$ Patches

Imagine that two $\text{Move}_{\text{join}}$ patches are generated for two OV bugs A_1B_1 and A_2B_2. There is clearly no chance for merging, if one patch uses Move-Up (i.e., moving `join` to execute before B_1 or B_2) and the other patch uses Move-Down (i.e., moving B_1 or B_2 to execute after `join`); there is also no benefit of merging, if both patches use the Move-Down strategy. When both patches use Move-Up, our merger explores merging these two patches if A_1 and A_2 are from the same thread, and B_1 and B_2 are from the same thread, just like that in Add_{join} patch merging.

Once the two patches pass the above checking, HFix creates a merged patch in a similar way as Add_{join} patch merging. That is, the merged patch keeps only one of the `join` from the two patches, at the nearest common dominator of the original locations in the two patches. Similar checking is conducted to make sure the merge does not bring extra risks of deadlocks or severe slowdowns.

HFix merges $\text{Move}_{\text{create}}$ patches in a similar way as merging $\text{Move}_{\text{join}}$ patches. We skip the discussion here.

6.6 Experimental Evaluation

6.6.1 Methodology

HFix is implemented using LLVM 3.6.1. All the experiments are conducted on eight-core Intel Xeon machines.

Benchmark Suite To evaluate HFix, we use two sets of real-world concurrency bugs. The first is the bug-fixing benchmark suite set up by CFix [42]. It contains 7 OV and 6 AV bugs. These 13 bugs are all representative benchmarks used by many previous works [41, 85, 101, 102]. They come from publicly released versions of 10 open-source C/C++ multithreaded applications, most of which contain tens to hundreds of thousands lines of code. They lead to severe crashes and security vulnerabilities. They are fixed by developers through a wide variety of strategies, which we will discuss in Chapter 6.6.2. For each bug, CFix suite contains the following information: (1) the original buggy software, (2) bug reports that can be used as inputs to auto-fixing tools, following the format discussed in Chapter 6.2, and (3) scripts for patch performance and correctness testing. Particularly, there is a slightly modified buggy program that contains random `sleep`s to make a bug manifests more frequently and hence more suitable for rigorous correctness testing.

The second set includes *all* the AV bugs in the concurrency-bug benchmark suite composed by Lu et. al. [64] that are fixed by developers through moving synchronization operations. There are six bugs in this set as shown in Table 3.3. For each bug, there is a bug report prepared by us using the format discussed in Chapter 6.2.

We use the first set of benchmarks, because it enables direct comparison between HFix and the state-of-the-art concurrency-bug fixing tool, CFix. We use the second set, because it allows a targeted evaluation about how well HFix can automate the Move fix strategy for AV bugs.

Evaluation Metrics We evaluate the quality of HFix patches mainly by comparing them with manual patches. We will explain what are the differences, if any, and whether/how the differences affect patch quality. It is infeasible to prove the correctness of a big multi-threaded software. Fortunately, HFix already provides soundness guarantees for its patches (discussed at the end of Chapter 6.4), and we will use comparison with manual patches to further demonstrate the correctness of HFix patches.

Since the main goal of HFix is to improve the patch simplicity of the state of the art, we will quantitatively measure patch simplicity by counting the number of new synchronization operations added by each patch.

In addition to comparing with manual patches, we will also compare HFix with CFix using CFix benchmark suite. We will use the simplicity metric discussed above, and also run the performance and correctness testing provided by CFix benchmark suite to show that HFix patches do not hurt performance or correctness.

6.6.2 Experimental Results

Overall Results For OV Bugs

As shown in Table 6.5.2, HFix correctly identifies OV bugs that are suitable for Add or Move fix strategies, and effectively generates patches that are as simple as manual patches, well complementing the state of the art.

Among all the 7 OV bug benchmarks, HFix automatically and correctly generates simple patches for five. HFix also makes correct decisions for the other two that indeed cannot be fixed by Add_{join} or Move. Specifically, OV3 happens when a child thread unexpectedly reads a variable before it is initialized in the parent thread. As correctly pointed out by HFix, OV3 cannot be fixed by Add_{join} or Move, because `join` cannot force parent-thread operations to execute before child-thread operations and data-dependency prevents Move from being applied. The situation in OV2 is opposite: the parent thread could read a variable before it is initialized in the child thread. $\text{Move}_{\text{join}}$ is tried and correctly aborted by HFix due to failed deadlock checking.

Comparing with manual patches, HFix performs very well. HFix produces *exactly the same* patches as developers manually did for OV1, OV4, and OV5. HFix patch for OV6 has a trivial control-flow difference from the corresponding manual patch — HFix patch conducts "`if(..){..; X;}else{X;}`", while manual patch does "`if(..){..}; X;`".

Comparing with CFix, HFix can generate much simpler patches than CFix does. For OV1 and OV4 — OV7, CFix introduces about four signal operations and one wait operation in *each* patch. Instead, HFix only introduces 0 – 1 synchronization operation. Note that, the CFix patch complexity goes beyond the synchronization operations listed in Table 6.5.2. For example, it also contains the declarations of new global synchronization variables; every signal or wait operation also comes with one lock, one unlock, and some corresponding flag setting/checking operations. Of course, CFix can fix all the seven OV bugs, including OV2 and OV3 that cannot be fixed by HFix. This result matches the different design goals of CFix, which emphasizes generality, and HFix, which emphasizes specialization and simplicity.

Overall Results For AV Bugs

HFix can patch not only OV bugs, but also AV bugs, through the Move strategy. However, manual Move patches are not as common for AV bugs than for OV bugs in real world (Table 3.3). Our evaluation shows a consistent trend.

Among the six AV bugs in CFix benchmark suite, none of them is suitable for Move strategy, because there is no lock/unlock operations around the buggy code. HFix correctly figures this out, and did not generate patch for any of them. CFix can fix all these six bugs, introducing one new global lock variable and 2–9 lock/unlock operations in each patch. Developers fixed these bugs by a mix of strategies, including adding lock/unlock operations, data privatization, changing reader-lock to writer-lock, and tolerating buggy timing.

Among the six AV bugs in our second benchmark suite (Table 6.6.2), HFix correctly

BugID	App.	HFix		Manual	
		Strategy	#Sync	Strategy	#Sync
AV1	Ap-21287	-	-	M_{unlock}	0
AV2	Mo-18025	M_{lock}	0	M_{lock}	0
AV3	Mo-141779	M_{lock}	0	M_{lock}	0
AV4	My-169	M_{unlock}	0	M_{unlock}	0
AV5	My-14262	M_{unlock}	0	M_{unlock}	0
AV6	My-14931	M_{unlock}	0	M_{unlock}	0

Table 6.2: HFix patch comparison for AV bugs

generates simple Move patches, involving no new synchronization operations or variables, for five of them. HFix patch for AV4 is exactly the same as the manual patch. HFix patches for AV2 and AV3 only have trivial difference from the manual patches, exactly like the case of OV6 discussed above. HFix patches for AV5 and AV6 are very similar with manual patches. The difference is that manual patches directly moved code between a caller function f_0 and a callee function f_1. Instead, HFix first inlines the corresponding invocation of f_1 inside f_0 before the movement. HFix did not generate Move patches for AV1, because none of the atomicity-violation related statements (`p`, `c`, and `r`) are inside any critical sections in the buggy software. The manual patch moves unlock statements to extend a critical section that did not contain `p`, `c`, or `r` in the buggy version to contain all three of them.

Other Detailed Results

Alternative patches For every evaluated bug, HFix tries all fix strategies and generates as many patches as possible. Having said that, OV6 is the only case where HFix generates more than one patch: one moves `join` up and one moves a memory access down. They only have a trivial difference and are semantically equivalent with each other: there are a few lines of local-variable computation that are executed after `join` in one patch, yet before `join` in the other.

Patch testing using CFix benchmark suite We conducted the patch testing provided by CFix benchmark suite. HFix patches passed the correctness testing: the failure rates of unpatched programs range between 10% and 60%, measured through 1,000 testing runs with random `sleep`s; the failure rates of HFix patched programs are all 0 under the same setting. HFix patches also passed the performance testing: the overhead of HFix patched programs is always under 0.5%, which is each averaged upon 1,000 failure-free runs, similar with that of CFix patches [42].

HFix static analysis HFix static analysis is efficient. It takes less than 5 seconds to generate patches for each bug.

Merging Our patch merging is effective. Among all the bugs, OV1 and OV4 are the two cases that contain multiple related bug reports. Without patch merging, HFix patches would have contained as many as 10 and 5 join operations for these two bugs. Fortunately, after the merging, only one `join` is needed for each, as illustrated in Figure 6.1a.

6.7 Conclusion

Automated bug fixing is both challenging and important. Many automated fixing techniques have been proposed recently for concurrency bugs. This Chapter provides an in-depth understanding of this research direction, through a thorough study of manual patches for 77 real-world concurrency bugs. Our study provides both endorsement for existing techniques and actionable suggestions for future research to further improve the quality of automatically generated patches. Our design of HFix leverages some of these findings, and our evaluation shows that HFix can indeed produce high-quality patches for many real-world concurrency bugs in large applications. We believe future research can further improve the quality of auto-patches following the guidance provided by our patch study and extending HFix.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Contributions

In conclusion, this dissertation makes following three contributions.

1. Conduct an empirical study that studies timing bugs in production-run cloud service systems and multi-threaded systems. For the study of timing bugs in Microsoft Azure incidents, four observations are made: (1) about 15% software-bug incidents in our study set are caused by timing bugs; (2) 60% timing bugs are message timing bugs or fault timing bugs; (3) half of timing bugs in our study are racing on persistent data instead of shared global memory variables; (4) mitigation, especially running-environment mitigation, is a more common strategy to resolve timing-bug incidents in the cloud. For the manual patch study of timing bugs in multi-threaded systems, two key findings are made about local concurrency bug fixing: (1) lock is the dominant synchronization primitive for enforcing atomicity; lock-related signals/waits are not the dominant primitive for enforcing pairwise ordering in patches. (2) leveraging existing synchronization in software is as common as adding extra primitives.

2. Model and design detection tools, DCatch and FCatch, for message timing bugs and fault timing bugs. This dissertation proposes a new model to capture unique properties (new concurrent source, new synchronization mechanisms, and new sharing resources) of these two timing bugs. Guided by these models, our detection tools are proposed to detect message timing bugs and fault timing bugs in four steps: (1) tracing correct runs; (2) trace analysis; (3) static pruning or identifying fault-intolerant operations; (4) triggering. Each step is carefully customized to address unique challenges for DCbugs.

3. Model and design a fixing tool for LCbugs. Motivated by the empirical study, this

dissertation designs HFix to enforce timing relationships and fix local concurrency bugs by leveraging non-lock synchronization primitives. The first, HFix$_{join}$, enforces ordering relationship by adding thread-join primitives, instead of signals/waits. The second, HFix$_{move}$, enforces ordering and atomicity relationship by leveraging synchronization primitives that already exist in software. The evaluation shows that it can automatically generate patches that have matching quality with manual patches.

7.2 Limitation and Future Work

Future work can explore opportunities along three directions to continue tackling timing bugs in multi-threaded systems and distributed systems.

- Our empirical study on timing-bug incidents in production-run cloud-service systems shows that mitigation, especially running-time environment mitigation, is a popular approach to resolve timing bugs. Much recent work looked at how to automatically generate new patches. In comparison, automatically generating mitigation steps for timing bugs has not been well explored and worth more attention in the future. Our manual patch empirical study for timing bugs in multi-threaded systems mainly focuses on the fixing strategy of LCbugs racing on shared global memory. What's the common fixing strategy for timing bugs racing on persistent data? A future empirical study to answer this question would insight and guide automated fixing tool design.

- Right now our DCatch and FCatch rely on logical-time happens before relationship and dynamic analysis techniques to predict DCbugs. They are definitely not a panacea.

  – DCatch could miss message timing bugs for several reasons. First, the current prototype of DCatch only reports message timing bugs that lead to explicit failures. True message timing bugs that lead to severe but silent failures would be missed. Future work could address this problem by skipping the static pruning step, and simply applying the triggering module for all message timing bug candidates. Second, DCatch selectively monitors only memory accesses related to inter-node communication and corresponding computation. This strategy is crucial for the scalability of DCatch. However, there could be message timing bugs that are between communication-related memory accesses and communication-unrelated accesses. These bugs would be missed by DCatch. Future work could selectively trace important variables in both communication-related and communication-unrelated accesses. Third, DCatch may not process extremely large traces. The scalability bottleneck of DCatch, when facing huge traces, is its trace analysis. It currently takes about 4G memory for traces in our benchmarks. Future work can chunk the traces and conduct detection within each chunk, an approach used by previous LCbug detection tools.

  – FCatch could miss fault timing bugs for several reasons. First, FCatch does not cover bugs beyond our fault timing bug model. Future work could extend our model to cover bugs that involve the interaction among more than one fault or more than one resource. Second, FCatch does not relying on any manual specifications. Inevitably, this could cause inaccuracies. FCatch may miss recovery operations, and may ignore true bugs whose harm does not match what FCatch automatically checks. Future work could add new feature to allow users to specify recovery-handler interfaces or functions. Third, FCatch cannot guarantee to accurately identify all fault-tolerance mechanisms or synchronization loops. For example, FCatch does not consider sanity checks involving multiple variables or time-out mechanisms enforced by monitoring threads. Future work can extend FCatch to cover more fault-tolerance mechanisms and synchronization loops.

- HFix enforces timing relationships in LCbugs by leveraging non-lock synchronizations. Similar situation happens for DCbugs in distributed systems and even becomes worse.

Fixing DCbugs often cannot relay on traditional lock-related synchronization primitives like locks and condition variables, as block waits cannot help fix fault timing bugs and can easily introduce new bugs if put in event/RPC handlers where most message timing bugs are located. How to fix DCbugs with non-lock synchronizations remains an open problem. A starting and exciting exploration along this direction is presented in DFix [54], which uses rollback and fast-forward to fix message timing bugs and fault timing bugs. Last, how to fixing LCbugs with the bypass strategy as shown in our empirical study findings is still an open question. Future work can explore and design fixing tools to automatically generate bypass patches.

REFERENCES

[1] Apache bugzilla. https://issues.apache.org/bugzilla/index.cgi.

[2] Mozilla bugzilla. http://bugzilla.mozilla.org/.

[3] Mysql bugs. http://http://bugs.mysql.com/.

[4] Welcome to apache openoffice (aoo) bugzilla. https://bz.apache.org/ooo/.

[5] Hbase-3596. https://issues.apache.org/jira/browse/HBASE-3596, 2011.

[6] Mapreduce-3858. https://issues.apache.org/jira/browse/MAPREDUCE-3858, 2012.

[7] Cassandra-5393. https://issues.apache.org/jira/browse/CASSANDRA-5393, 2013.

[8] Cassandra-6415. https://issues.apache.org/jira/browse/CASSANDRA-6415, 2013.

[9] Hbase-10090. https://issues.apache.org/jira/browse/HBASE-10090, 2013.

[10] Mapreduce-5476. https://issues.apache.org/jira/browse/MAPREDUCE-5476, 2013.

[11] Zookeeper-1653. https://issues.apache.org/jira/browse/ZOOKEEPER-1653, 2013.

[12] Java platform standard edition 7 documentation. https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode(), 2017.

[13] Software fail watch: 5th edition. https://www.tricentis.com/resources/software-fail-watch-5th-edition/, 2018.

[14] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *ISCA*, 1991.

[15] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Correlated crash vulnerabilities. In *OSDI*, 2016.

[16] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *SoCC*, 2016.

[17] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven Fault Injection.

[18] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[19] Pavol Černỳ, Thomas A Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *Computer Aided Verification*, 2013.

[20] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. Angelic debugging. In *ICSE*, 2011.

[21] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for java. In *ICSE*, 2008.

[22] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. In *PLDI*, 2013.

[23] Mattia Fazzini, Prateek Saxena, and Alessandro Orso. Autocsp: Automatically retrofitting CSP to web applications. In *ICSE*, 2015.

[24] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[25] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[26] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *DSN*, 2010.

[27] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *FAST*, 2017.

[28] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *ICSE*, 2015.

[29] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In *FSE*, 2015.

[30] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI*, 2011.

[31] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *SoCC*, 2014.

[32] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *SoCC*, 2016.

[33] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP*, 2011.

[34] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure recovery: When the cure is worse than the disease. In *HotOS*, 2013.

[35] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP*, 2015.

[36] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race Detection for Event-Driven Mobile Applications. In *PLDI*, 2014.

[37] Jeff Huang and Charles Zhang. Execution privatization for scheduler-oblivious concurrent programs. In *OOPSLA*, 2012.

[38] IBM. Main page - walawiki. `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[39] jboss javassist. Javassist. `http://jboss-javassist.github.io/javassist/`.

[40] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[41] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[42] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[43] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. Minthint: automated synthesis of repair hints. In *ICSE*, 2014.

[44] Baris Kasikci, Cristian Zamfir, and George Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, 2012.

[45] Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *ISSTA*, 2015.

[46] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[47] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ICSE*, 2013.

[48] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[49] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, 2012.

[50] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI*, 2014.

[51] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, 2016.

[52] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[53] Du Li, Witawas Srisa-an, and Matthew B. Dwyer. SOS: saving time in dynamic race detection with stationary analysis. In *OOPSLA*, 2011.

[54] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi Gunawi, and Shan Lu. Dfix: Automatically fixing timing bugs in distributed systems. In *PLDI*, 2019.

[55] Haopeng Liu, Yuxi Chen, and Shan Lu. Understanding and generating high quality patches for concurrency bugs. In *FSE*, 2016.

[56] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In *ASPLOS*, 2017.

[57] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. In *ASPLOS*, 2018.

[58] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *FSE*, 2014.

[59] Peng Liu, Omer Tripp, and Xiangyu Zhang. Flint: Fixing linearizability violations. In *OOPSLA*, 2014.

[60] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ICSE*, 2012.

[61] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[62] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *OOPSLA*, 2012.

[63] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, 2007.

[64] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[65] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[66] Brandon Lucia, Luis Ceze, and Karin Strauss. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-Variable Atomicity Violations. In *ISCA*, 2010.

[67] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *EuroSys*, 2019.

[68] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *SOSP*, 2015.

[69] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race Detection for Android Applications. In *PLDI*, 2014.

[70] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *ASPLOS*, 2017.

[71] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[72] Robert H. B. Netzer and Barton P. Miller. Improving The Accuracy of Data Race Detection. In *PPoPP*, 1991.

[73] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[74] Semih Okur and Danny Dig. How do developers use parallel libraries? In *FSE*, 2012.

[75] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *ATC*, 2014.

[76] Oracle. Virtualbox oracle vm virtualbox. `https://www.virtualbox.org/wiki/VirtualBox`.

[77] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *ICSE*, 2010.

[78] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Finding Places. In *ASPLOS*, 2009.

[79] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective Race Detection for Event-Driven Programs. In *OOPSLA*, 2013.

[80] Caitlin Sadowski, Jaeheon Yi, and Sunghun Kim. The evolution of data races. In *MSR*, 2012.

[81] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE*, 2012.

[82] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[83] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[84] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, 2008.

[85] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do I Use the Wrong Definition? DefUse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs. In *OOPSLA*, 2010.

[86] Benjamin H. Sigelman, Luiz Andr Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[87] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *SSV*, 2010.

[88] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Scalable dynamic partial order reduction. In *International Conference on Runtime Verification*, pages 19–34. Springer, 2012.

[89] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic Recognition of Synchronization Operations for Improved Data Race Detection. In *ISSTA*, 2008.

[90] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlk. Gadara: dynamic deadlock avoidance for mult-threaded programs. In *OSDI*, 2008.

[91] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Tom Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI*, 2015.

[92] Rui Xin, Zhengwei Qi, Shiqiu Huang, Chengcheng Xiang, Yudi Zheng, Yin Wang, and Haibing Guan. An automation-assisted empirical study on lock usage for concurrent programs. In *ICSM*, 2013.

[93] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad Hoc Synchronization Considered Harmful. In *OSDI*, 2010.

[94] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[95] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, 2009.

[96] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

[97] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[98] Yuan Yu, Thomas Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[99] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, 2014.

[100] Jiaqi Zhang, Weiwei Xiong, Yang Liu, Soyeon Park, Yuanyuan Zhou, and Zhiqiang Ma. ATDetector: improving the accuracy of a commercial data race detector by identifying address transfer. In *MICRO*, 2011.

[101] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS*, 2011.

[102] Wei Zhang, Chong Sun, and Shan Lu. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. In *ASPLOS*, 2010.