THE UNIVERSITY OF CHICAGO

PERFORMANCE-FRIENDLY CONCURRENCY BUG FAILURE RECOVERY AND FIXING

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
YUXI CHEN

CHICAGO, ILLINOIS
MARCH, 2020
Dedicated to my parents, for their continuous support and endless love.
Program testing can be used to show the presence of bugs, but never to show their absence!

– Edsger W. Dijkstra
# TABLE OF CONTENTS

LIST OF FIGURES ....................................................... vii
LIST OF TABLES ........................................................ ix
ACKNOWLEDGMENTS ..................................................... x
ABSTRACT ................................................................. xii

1 INTRODUCTION .......................................................... 1
  1.1 Motivation ............................................................ 5
    1.1.1 Concurrency bugs failure recovery ....................... 5
    1.1.2 Automatic concurrency bug fixing ....................... 7
  1.2 Contributions ..................................................... 11
  1.3 Dissertation Organization ..................................... 16

2 BACKGROUND AND RELATED WORK ................................ 17
  2.1 Concurrency bug .................................................. 17
  2.2 Empirical study of concurrency bugs ......................... 18
  2.3 Concurrency bug detection ..................................... 19
  2.4 Concurrency-bug failure prevention ......................... 19
  2.5 Failure recovery ................................................ 20
  2.6 Concurrency bug fixing ........................................ 21
  2.7 Condition synthesis in program repair ..................... 21

3 APPLYING TRANSACTIONAL MEMORY FOR CONCURRENCY-BUG FAILURE RECOVERY IN PRODUCTION RUNS ..................... 23
  3.1 Challenges for applying transactional memory for concurrency bug failure recovery ........................................ 23
  3.2 Transactional Memory (TM) ...................................... 24
  3.3 ConAir ............................................................. 25
  3.4 BugTM_H ........................................................... 27
    3.4.1 High-Level Design ........................................ 27
    3.4.2 Design about AbortTx .................................... 29
    3.4.3 Design about StartTx and CommitTx ................... 30
    3.4.4 Design for fallback and retry .......................... 32
    3.4.5 Inter-procedural Designs and Others .................. 33
  3.5 Roadmap for Further Exploration ............................. 34
  3.6 BugTM_S .......................................................... 35
    3.6.1 Deferred Writes for Failure-Unrelated \( w_{\text{kill}} \) .......... 36
    3.6.2 Undo Log for Failure-Related Killing Writes .......... 38
  3.7 BugTM_HS ........................................................ 40
  3.8 Failure Diagnosis ............................................... 42
3.9 Methodology ................................................. 45
3.10 Experimental results ....................................... 46
  3.10.1 Failure recovery capability .......................... 46
  3.10.2 Performance ........................................ 51
  3.10.3 Diagnosis .......................................... 53
  3.10.4 Alternative designs of BugTM ....................... 55
3.11 Discussion ................................................ 56
3.12 Conclusion ............................................... 58

4 AUTOMATICALLY FIX CONCURRENCY BUGS VIA BYPASSING COMPUTATION SEMANTICS .......................................................... 59
  4.1 Challenges for automatically generating correct bypassing patches for concurrency bugs ............................................ 59
  4.2 Existing auto-fixing tools .................................. 61
  4.3 Bug-fixing framework ...................................... 62
  4.4 Three bypassing patterns .................................. 64
    4.4.1 Overwritten ....................................... 64
    4.4.2 Conditional ....................................... 67
    4.4.3 Exclusive ......................................... 70
  4.5 BFix .................................................... 73
    4.5.1 Workflow of BFix .................................. 73
    4.5.2 Feasibility checking ................................ 74
    4.5.3 Patch generation .................................. 81
    4.5.4 Patch merging .................................... 86
  4.6 Methodology .............................................. 87
  4.7 Experimental results ..................................... 88
    4.7.1 Correctness ....................................... 89
    4.7.2 Performance ....................................... 90
    4.7.3 Simplicity ......................................... 92
  4.8 Discussion ............................................... 94
  4.9 Conclusion .............................................. 95

5 CONCLUSION AND FUTURE WORK ............................. 96
# LIST OF FIGURES

1.1 Main phases of software life cycle and their corresponding tasks .............................................. 2  
1.2 Single-threaded recovery for concurrency bugs ............................................................................ 6  
1.3 A real-world concurrency bug from Mozilla ............................................................................. 7  
1.4 A simplified real-world concurrency bug from Mozilla DNS module. ‘+/−’ indicates the manual patch .............................................................................................................. 9  
1.5 Design space of concurrency-bug failure recovery (Heart: non-existing optimal design; Rx [60] changes OS) ............................................................................................................. 10  
1.6 Execution timeline for the patched version in bug-triggering timing and the original program in a bug-free timing of Figure 1.4. All timeline figures in this dissertation use the same symbols: gray font means FALSE Boolean expression; rectangle represents skipped regions. .................................................................................. 14  

2.1 A real-world OV bug (simplified from Transmission) ............................................................... 18  
2.2 A real-world DD bug (simplified from HawkNL) ..................................................................... 18  

3.1 A real-world concurrency bug from MySQL ............................................................................ 26  
3.2 BugTMAbortTx wrapper function (my_xabort) ..................................................................... 29  
3.3 A toy example adapted from Figure 1.3 (left-side) and its BugTMH transformation (right-side) ................................................................................................................................. 31  
3.4 BugTMHCommitTx wrapper function (my_xend) .................................................................... 32  
3.5 BugTMHStartTx wrapper function (my_xbegin) .................................................................... 32  
3.6 BugTMS deferred write transformation, denoted by ‘+/−’, makes a ConAir-unrecoverable bug recoverable ............................................................................................................................... 36  
3.7 Moving a killing write (ttag is initialized as FALSE) ............................................................... 38  
3.8 Memory-checkpoint example .................................................................................................... 38  
3.9 BugTMHS rollback wrapper function .......................................................................................... 42  
3.10 Recovery-guided root-cause diagnosis for BugTMH and BugTMS ........................................... 43  

4.1 Why bypassing works for AV bugs: (i) bug context for write-after-write AV bug; (ii) skipping failure-triggering operations; (iii) skipping buggy operations ............................................. 61  
4.2 MySQL example from CFix. Interleaving ‘A’ and ‘B’ are two bug-free interleavings, ‘+’ indicates the generated patch by CFix ............................................................................................................................. 62  
4.3 Notation for the buggy code snippet. Arrows indicate the bug-triggering timing .......... 63  
4.4 Types of AV bugs. ‘x’ is a shared memory location ................................................................. 64  
4.5 A simplified bug in Jscript module of Mozilla, ‘+’ indicates the manual patch ................. 65  
4.6 Execution timeline for the patched version in bug-triggering timing and a bug-free run in Figure 4.5 ............................................................................................................................................ 66  
4.7 Simplified overwritten pattern for read-after-write AV bugs and their patch pattern .......... 66  
4.8 Simplified overwritten pattern for write-after-read AV bugs and their patch pattern .......... 67  
4.9 Simplified overwritten pattern for OV bugs and their patch pattern .................................. 67  
4.10 A simplified bug from Apache Xerces. ‘+/−’ indicates the manual patch ......................... 68  
4.11 Execution timeline for the patched version in bug-triggering timing and a bug-free run in Figure 4.10 .............................................................................................................................................. 69  

vii
4.12 Simplified conditional pattern for bypassing $R$ and the patch pattern .... 70
4.13 Simplified conditional pattern for bypassing $C$ and the patch pattern .... 70
4.14 Simplified conditional pattern for bypassing $A$ and the patch pattern .... 71
4.15 All conditional patterns that BFIX handles. $global\_field$ is a shared memory location, it can be a variable or expression. For each case, the inferred correct timing and bypass-able operation are also shown ......................... 71
4.16 A modified version of a concurrency bug in Apache, ‘+’ indicates the manual patch 72
4.17 Workflow of BFIX .................................................. 74
4.18 Bypass-patched program may change the execution order of original program; dotted arrows means execution order is changed; same notation is used as Figure 4.3 ................................................................. 78
4.19 Patch pattern for bypassing $R$ ($pid$ is thread ID) .................... 83
4.20 Patch pattern for bypassing $C$ ($pid$ is thread ID) .................... 83
4.21 Patch pattern for bypassing $A$ ($pid$ is thread ID) .................... 84
4.22 A naive bypassing patch with a new bug .......................... 85
4.23 Simple patch for AV bugs ........................................ 85
4.24 A patch pattern used by developers ............................ 91
4.25 A simplified Mozilla bug BFIX can not help; ‘+’ indicates patch .... 94
## LIST OF TABLES

1.1 Design comparisons ......................................................... 12

3.1 Common types of concurrency bugs and how BugTM\(_H\) and ConAir attempt to recover from them ......................................................... 28

3.2 Recovery capability comparison ............................................... 47

3.3 Static failure sites hardened by BugTM\(_{HS}\) ........................................ 49

3.4 Overhead during regular execution and detailed performance comparison ...... 50

3.5 Failure recovery details by BugTM\(_{HS}\) ........................................ 52

3.6 Diagnosis capability comparison ............................................. 54

3.7 BugTM\(_H\) vs. alternative designs ............................................... 55

4.1 Failure Rates comparison ..................................................... 89

4.2 Performance comparison ..................................................... 90

4.3 Simplicity comparison (#Sync: number of new synchronization operations added by the patch; #line: line of code modification; L: lock; U: unlock; S: signal; W: wait; +: add; -: remove; ↑: move) ..................................................... 92
ACKNOWLEDGMENTS

The past five years had been the most beautiful memories in my life. I still remember the day I got the offer from the University of Chicago, as if it were yesterday. Looking back over the last five years, it is not an easy journey for me. I would like to express my most sincere gratitude to people who have helped me.

First and foremost, I would like to thank my advisor, Prof. Shan Lu, for the endless effort that she spent on me. She is a great mentor not only in the research but also in the daily life. From her, I knew how to be a passionate researcher with critical thinking. She always pushed me to think and express my idea bravely, which is the most important lesson that I have learnt from the graduate school. From her, I knew how to be an optimistic person in my own life. I had a really tough time in my third year and kept doubting myself. Her patience and invaluable suggestions pull me back on track. It is my honor to be her PhD student. Without her help, I do not think I can reach the destination.

I would like to thank Prof. Haryadi S. Gunawi, Prof. Ravi Chugh for being on my doctoral dissertation committee, and Prof. Andrew A. Chien for being on my master’s committee. Their constructive suggestions help improve my research and make my dissertation stand on a solid foundation. I am really honored to have them on my committee.

At UChicago, I was fortunate enough to work with many excellent colleagues. I would like to thank every member in Shan’s and Haryadi’s groups: Wei Zhang, Guoliang Jin, Linhai Song, Haopeng Liu, Guangpu Li, Shu Wang, Junwen Yang, Chi Li, Lefan Zhang, Chengcheng Wan, Xu Wang, Xianglan Chen, Jiaxin Li, Wei Yuan, Yu Gao, Jeffrey Lukman, Tanakorn Leesatapornwongsa, Riza Suminto, Huan Ke, Shiqin Yan and Huaicheng Li for the opportunity of collaborations and technical discussions. I especially thank Haopeng Liu and Shiqin Yan. In the past five years, we wen to campus and went home together; we went to Chinatown every weekends together; we travelled together. You are not only friends for me, but also brothers. I also want to thank the technical support from Bob Bartlett,
administrational help from Margaret P. Jaffey.

I am very happy to make lots of fantastic friends here: Steven Basart, Min Xu, Yuwanwei Fang, Zhixuan Zhou, Liwen Zhang, Dixin Tang, Aiman Fang, Qingqin Zheng, Yun Li, Yan Liu, Huazhe Zhang, Cong Zhang, Jialin Li, Yan Yang and many more I can not list all of them here. I like hanging out, travelling, studying with you guys.

I would also thank my roommates: Yanzhe Wu, Hao Tong, Hing Yin Tsang, Cheng Wu. Even we have different lifestyles, it is still a great experience for me to live with you in the past five years.

This dissertation means nothing for me if it did not make my family proud. Thank you, Dad, Mom and my younger brother, for your unconditional and greatest love. And I am very lucky to find my true love Yalin Sun during this journey, thanks for your company.
ABSTRACT

Concurrency bugs widely exist and severely threaten system availability and reliability. Their unique non-deterministic nature has made them difficult to avoid, diagnose, and fix by developers. In the past, much research has been done to fight with concurrency bugs during the early phases of software life cycle. For example, people designed new languages (e.g. Go) and new static analysis tools to help avoid some concurrency bugs during the code design and the implementation. And many testing techniques and dynamic tools are proposed to expose concurrency bugs during in-house testing. However, none of these techniques are perfect. Many concurrency bugs still inevitably slip into deployment, cause production failures and incur huge cost for code maintenance, such as 60 million ether theft in the Bitcoin system. Consequently, besides the early phases of software life cycle, automated tools that help fight with concurrency bugs in the late phases of software life cycle are well desired.

To help fight with concurrency bugs in the late phases of software life cycle, this dissertation tries to solve two problems: (1) During the production deployment, how to help software survive from failures caused by concurrency bugs with low overhead? (2) During the software maintenance, how to automatically fix concurrency bugs so that the patches are correct and simple without unnecessary performance degradation? In other words, this dissertation aims to improve the availability and reliability of multi-threaded programs through efficient concurrency bug failure recovery and fixing.

Along the direction of failure recovery, we present BugTM, an approach that applies transactional memory techniques for failure recovery in production runs. Requiring no knowledge about where are concurrency bugs, BugTM uses static analysis and code transformation to enable BugTM-transformed software to recover from a concurrency-bug failure by rolling back and re-executing the recent history of a failure thread. BugTM greatly improves the recovery capability of state-of-the-art techniques with low run-time overhead and no changes to OS or hardware, while guarantees not to introduce new bugs.
Along the direction of efficient concurrency bug fixing, we present BFix, a tool that automatically generates computation-bypassing patches for some concurrency bugs. Given a bug report, BFix first uses static analysis to check whether the bypassing strategy is suitable for the reported bug and, if so, constructs a patch. It further tries to combine its patches for better performance and code readability. We have compared BFix patches with bypassing patches manually developed by programmers, as well as the patches generated by state-of-the-art auto-fixing tools. The experimental results showed that BFix patches have the similar quality to the manual patches, outperforming auto-generated patches from previous tools in terms of patch performance, and patch simplicity.

These two approaches can greatly improve software availability and reliability by tackling detected or undetected concurrency bugs in the late phases of software life cycle. BugTM leverages transactional memory techniques for concurrency bug failure recovery without intensively using checkpointing. BFix tries to skip certain part of the original computation in the bug-triggering context without using time-consuming synchronization primitives (e.g., locks). Both of them achieve good efficiency, which makes them serve as good choices to help fight with concurrency bugs in the late phases of software life cycle for developers.
CHAPTER 1
INTRODUCTION

Concurrency bugs are synchronization mistakes in multi-threaded software. Their unique non-deterministic nature has made them difficult to avoid, detect, diagnose, and fix. Even with lots of in-house testing, many concurrency bugs continue to slip into production runs, causing disasters [4, 36, 56, 66]. To make things worse, previous studies have shown that it often takes several months for developers to diagnose and correctly fix concurrency bugs [46]—concurrency bugs are the most difficult to fix correctly among common bug types [78], with many incorrect patches releases. Consequently, techniques or tools that can help fight with concurrency bugs are highly desired.

For different phases of software life cycle, since each of them has distinct task and requirement, the desired techniques or tools that can help fight with concurrency bugs are different. Software life cycle is a process for planning, creating, testing, and deploying a software system [3]. This cycle usually has five main phases, as shown in Figure 1.1: (1) design: describe the desired features and operations into complete, detailed software design documents; (2) implementation: convert the software specifications into an executable software, the real code is written during this phase; (3) testing: bring all components into a special testing environment, then check errors, bugs and interoperability; (4) deployment: put software into production and run actual business; (5) maintenance: assess and evaluate the software to ensure it does not become obsolete, and make changes into initial software for better user experience. As mentioned, different techniques or tools are needed in different phases. During the design phase, developers mainly want to avoid some concurrency bugs at very beginning by specifying details of design documents and tools. For example, developers prefer to adopt a multi-threaded-programming-friendly language so that some compiler-related concurrency bugs, like double-checked locking [65] can be avoided early on. During the implementation phase, efficient tools that can help find the violation [37] be-
Figure 1.1: Main phases of software life cycle and their corresponding tasks

tween specifications and real implementations for multi-threaded programs are very helpful. During the testing phase, static and dynamic bug detection tools, which have high coverage and high accuracy, are expected to expose the concurrency bugs. During the deployment phase, software availability requires effective and efficient recovery mechanisms to help software survive from the failures caused by concurrency bugs. During the maintenance phase, automatically patching concurrency bugs so that the patches are correct and simple without unnecessary performance degradation is anticipated.

People have made huge progress in dealing with concurrency bugs in different phases of software life cycle. For example, much research effort has been made on bug finding [33, 45, 48, 64, 80, 85] during in-house testing. However, none of them has helped expose all concurrency bugs. People even design a specific language Go for multi-threaded programming to avoid concurrency bug during even earlier phase of software life cycle (code design and implementation phases). Unfortunately, Tu et al. [69] finds it is even easier to introduce concurrency bugs if developers do not have strong background about the inner mechanism of Go. As you can see, people always focus on the early phases of software life cycle to fight with concurrency bugs. However, since none of these techniques or tools are prefect, many
Concurrency bugs still slip into the late phases of software life cycle, such as the deployment and maintenance phases, causing disasters and incurring huge cost for software maintenance. This is catastrophic and irreversible, since concurrency bugs would directly affect the end users. As a result, tools or techniques that help fight with concurrency bugs during the late phases of software life cycle are even more highly desired.

Late in the software life cycle, software has already been deployed in production, hence solutions to help fight with concurrency bugs during these phases should meet higher standards than any early phase. First of all, they should guarantee the correctness. Early in the software life cycle, the newly-introduced problems caused by imperfect tools or techniques can be still addressed before deployment. However, all problems during the late phases will lead to direct and irreversible financial loss and user loss. According to statistics, downtime can cost companies $5,600 per minute and up to $300,000 per hour [2]. Second, performance really matters, but it may not be a major problem during the early phases. For example, the concurrency bug detection tools used during testing phase can incur 10X to 100X slowdown [13]. However, during the late phases, high overhead and long latency means huge user loss. For example, the BBC found they lost an additional 10% of users for every additional second their site took to load [5].

People have already paid more attentions to proposing new solutions to help fight with concurrency bugs during the late phases of software life cycle. However, none of them can completely satisfy both requirements described above. Performance is usually scarified for correctness and generality. For example, rollback-and-reexecution is used to recover failures caused by concurrency bugs during the deployment phase, but aggressive intrusiveness and high overhead prevent it from being widely deployed in production run. Although great progress has been made in automated concurrency fixing during the code maintenance, patches generated by existing tools are mostly different from patches designed by developers and most of them are not performance-friendly since time-consuming synchronization
primitives are introduced. More details are covered in chapter 1.1.

In this dissertation, we focus on not only correctness but also performance. We aim to present new solutions to help software survive from concurrency bug failures with low overhead and automatically fix concurrency bugs with high quality patches. In other words, we try to improve the availability and reliability of multi-threaded programs through efficient concurrency bug failure recovery and fixing, which take effect during the late phases of software life cycle. For concurrency bug failure recovery, our solution actually does not eliminate the concurrency bugs and it does not know where they are; it just helps software survive from the failures caused by concurrency bugs to improve the availability of multi-threaded programs. As for the concurrency bug fixing, our solution tries to automatically patch the buggy programs and eliminate the reported concurrency bugs so that the software reliability is improved.

We design low-overhead production-run concurrency bug failure recovery tool BugTM [10, 11] to help software survive from failure caused by concurrency bugs that fail to be detected during in-housing testing. And for those which can be detected during in-housing testing, we propose BFix, which is the first automatic fixing tool using bypassing strategy to fix concurrency bugs.

These two solutions share several similarities: (1) They are both high-quality solutions. Whether for concurrency bug failure recovery or concurrency bug fixing, our solutions guarantee correctness, without introducing new problems, and great performance, with negligible overhead in production environment. These are different from the traditional ones. (2) Both solutions greatly explore the current design space. Unlike existing solutions, which touch corners of the design space, BugTM achieves good combination between recovery capability and overhead, while BFix achieves good balance among fixing capability, overhead and simplicity. (3) From the perspective of used techniques, somehow, both would change sequential semantics in the bug-triggering context. BugTM would skip the failure-triggering operations
and then trigger execution when bug is about to happen. BFix would skip failure-inducing and related computation to fix concurrency bugs in the bug-triggering timing. (4) Both need a few to no human effort. BugTM does not need any knowledge about where the concurrency bugs are in the software. As for BFix, its input is directly obtained from existing concurrency bug detectors. Hence, both of them greatly alleviate the burden of developers. (5) Both BugTM and BFix tremendously complements existing solutions. They can greatly improve the availability and reliability of software during the late phases of software life cycle. Before the final patch is released, BugTM solution can serve as tentative patch for availability purpose. The diagnosis part of BugTM can complement bug detection tools, offering input for BFix and helping generate correct patch. Moreover, together with existing solutions, they can provide multi-phase, effective and efficient protections for multi-threaded programs.

These two solutions clearly have their own distinct goals and features given the fact that they are applied in different phases of software life cycle. Chapter 1.1 and 1.2 cover these parts. Chapter 1.1 introduces the specific goals for our concurrency bugs failure recovery and bug fixing solutions, while chapter 1.2 covers the novelty of our tools.

1.1 Motivation

1.1.1 Concurrency bugs failure recovery

During the late phases of software life cycle, like the deployment phase, rollback-and-reexecution is a classic approach to recover failures caused by concurrency bugs. When a failure happens during a production run, the program rolls back and re-executes from an earlier checkpoint. Due to the unique non-deterministic nature of concurrency bugs, the re-execution could get around the failure.

This approach is appealing for several reasons. It is generic, requiring no prior knowledge
about bugs; it improves availability, masking the manifestation of concurrency bugs from end users; it avoids causing system inconsistency or wasting computation resources, which often come together with naive failure restarts; even if not successful, the recovery attempts only delays the failure by a negligible amount of time.

This approach also faces challenges in performance, recovery capability, and correctness (i.e., not introducing new bugs), as we elaborate below.

Traditional rollback recovery conducts full-blown multi-threaded re-execution and whole-memory checkpointing. It can help recover almost all concurrency-bug failures, but incurs too large overhead to be deployed in production runs [60, 63]. Even with support from operating systems changes, periodic full-blown checkpointing still often incurs more than 10% overhead [60].

A recently proposed recovery technique, ConAir, conducts single-threaded re-execution and register-only checkpointing [82]. As shown in Figure 1.2, when a failure happens at a thread, ConAir rolls back the register content of this thread through an automatically inserted `longjmp` and re-executes from the return of an automatically inserted `setjmp`, which took register checkpoints. This design offers great performance (<1% overhead), but also imposes severe limitations to failure-recovery capability. Particularly, with no memory checkpoints and re-executing only one thread, ConAir does not allow its re-execution regions to contain writes to shared variables (referred to as `wkill`) for correctness concerns, severely hurting its chance to recover many failures.

This limitation can be demonstrated by the real-world example in Figure 1.3. In this example, the `NULL` assignment from Thread-2 could execute between the write ($A_1$) and the
read \((A_2)\) on \(s\rightarrow\text{table}\) from Thread-1, and cause failures. At the first glance, the failure could be recovered if we could rollback Thread-1 and re-execute both \(A_1\) and \(A_2\). However, such rollback and re-execution cannot be allowed by ConAir, as correctness can no longer be guaranteed if a write to a shared variable is re-executed (\(A_1\) in Figure 1.3): another thread \(t\) could have read the old value of \(s\rightarrow\text{table}\), saved it to a local pointer, the re-execution then gave \(s\rightarrow\text{table}\) a new value, causing inconsistency between \(t\) and Thread-1 and deviation from the original program semantics.

A better approach for concurrency bug failure recovery, which is as powerful as traditional rollback recovery and as lightweight as ConAir, is highly desired.

### 1.1.2 Automatic concurrency bug fixing

Although failure recovery techniques can help improve the availability of software, they do not fundamentally eliminate the concurrency bugs. They only help software survive from the failures caused by concurrency bugs when buggy timing occurs. The concurrency bugs still exist. Patching is the fundamental way to eliminate concurrency bugs.

Recently, many automated fixing techniques for concurrency bugs have been proposed \([9, 25, 27, 28, 39, 40, 41, 42, 74]\). These tools can handle all common types of concurrency bugs, such as order violations (OV), atomicity violations (AV). They leverage 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 [27, 40, 42, 74] and condition variable signal/wait [28] into software. These synchronization changes help eliminate bug-triggering timing from the program space and hence fix concurrency bugs.

Although helpful, these automatically generated synchronization fixes are sometimes much more complicated and performance degrading than patches that are manually designed by developers. According to a previous patch study [39], more than 30% of the studied real-world concurrency bugs were not fixed by eliminating buggy timings; nearly 20% of the bugs were actually fixed by simply bypassing certain computation in the original program under those buggy timings, which we refer to as bypassing patches. These manual patches are simple and light weight, involving no synchronization changes, but they deviate from the fundamental approach taken by existing concurrency-bug fixing tools, which is about eliminating buggy timings.

To understand bypassing patches designed by developers, we can look at a real-world concurrency bug from Mozilla. In Mozilla, several threads can invoke the ProcessLookup function, as illustrated in Figure 1.4, in parallel to fetch and process requests from the pending-request queue mQ. Although the request fetching (Line 12 of ProcessLookup) is conducted with lock protection, the request processing (through FireStop on Line 16) is conducted without lock protection. Concurrently, another thread might execute FindLookup, illustrated on the right side of Figure 1.4. Without lock protection, a DNS request processing thread’s (Thread-1) two consecutive accesses to mStatus in Line 8 & 28 of ProcessRequest could be interleaved by Thread-2’s remote write to mStatus (Line 9 on the right side of Figure 1.4), which can lead to a browser crash by assertion failure (Line 28).

The best patch that can be automatically generated by previous techniques [28, 39] would move the invocation of FireStop and hence its enclosing read access to mStatus into the critical section protected by lock DNS::Lock. However, such a patch would create an extremely long lock critical section—every invocation of FireStop involves network operations and hence
void ProcessLookup(...){
    DNS::Lock();
    lookup->mStatus = COMPLETE;
    ط
    while(!lookup->mQ.empty()){
        request = lookup->mQ.front();
        lookup->mQ.pop();
        DNS::Unlock();
        request->FireStop(...);
        DNS::Lock();
        lookup->mprocess--;
        DNS::Unlock();
    }
    DNS::Unlock();
}

void FireStop(...){
    assert checking for mStatus
    NS_ASSERT(lookup->mStatus == COMPLETE);
    ...
}

Figure 1.4: A simplified real-world concurrency bug from Mozilla DNS module. ‘+/-’ indicates the manual patch

takes a long time—and would essentially serialize the concurrent invocations of ProcessLookup from all threads, which is clearly unacceptable in terms of performance.

In practice, instead of relying on lock synchronizations, developers simply add a field mProcess to record the number of threads that are currently processing requests, and add an extra checking in Thread-2 so that the reset procedure is bypassed when there is any thread processing DNS requests.

Unlike the earlier patch, this manual patch does not eliminate the original buggy timing—the FireStop function can still be invoked concurrently with the FindLookup function. It does however ensure that the original bug never happens, as the new counter mProcess is guaranteed to be non-zero and hence cause any concurrent reset to be bypassed whenever a FireStop is invoked.
In addition to fixing the original bug, this manual patch also has a great *performance* advantage over the earlier patch that relies on lock synchronizations, as it does not impose any slowdowns on the processing of DNS requests. The patch also involves only a few lines of code changes and no synchronization changes.

Given the advantage of bypassing patches outlined above, it would be great if they could be generated automatically. To do so, we would need to automatically (1) decide which computation to bypass; (2) decide what is the condition to bypass specific computation; and (3) prove that the patched software can not only fix the original bug but also guarantee not to introduce new bugs into the software.

Automating these three tasks is challenging. Particularly, proving that the patched software does not introduce new bugs (i.e., the third task above) is difficult—bypassing a group of operations that have side effects at run time would clearly change the semantics of that run, and proving that the changed semantics is still correct is difficult. Indeed, a previous study has shown that \[68\] naive computation bypassing can easily introduce new bugs: more than 50\% of condition-synthesis patches generated by automatic program-repair tools, like GenProg [35], actually introduce new bugs even though they can eliminate the original failure symptoms. Furthermore, even bypassing patches designed by skilled developers can sometimes introduce new bugs \[59\], which further motivates systematic and automated bypassing-patch generation techniques.

![Figure 1.5: Design space of concurrency-bug failure recovery](image-url)

*Figure 1.5: Design space of concurrency-bug failure recovery (Heart: non-existing optimal design; Rx [60] changes OS)*
1.2 Contributions

This dissertation works on two components to address concurrency bug failure recovery and fixing problems: (1) Leveraging the transactional memory techniques to help software survive from failures caused by concurrency bugs with low overhead during production deployment; (2) Using bypassing strategy to fix concurrency bugs with high-quality patches during code maintenance.

(1) Applying transactional memory for concurrency-bug failure recovery in production runs

Existing recovery techniques only touch two corners of the design space — good performance but limited recovery capability or good recovery capability but limited performance — as shown in Figure 1.5.

Along the direction of efficient concurrency bug failure recovery, we present BugTM, a set of transactional-memory (TM) inspired designs that thoroughly explore the design space of concurrency-bug failure recovery, also shown in Figure 1.5. It greatly improves the combination of recovery-capability and performance over existing techniques, while still guarantees correctness.

BugTM explores 3 designs, all implemented as compiler passes that automatically instrument software at the byte-code level. The instrumented software conducts checkpoint, rollback, and re-execution for failure recovery in different ways across these three designs, as shown in Table 1.1.

**Hardware BugTM**, short for BugTM\(_H\), uses hardware transactional memory (HTM) techniques\(^1\) *exclusively* to help failure recovery. When a failure is going to happen, a hardware transaction abort causes the failing thread to roll back. The re-execution naturally starts from the beginning of the enclosing transaction, carefully inserted by BugTM\(_H\).

**Software BugTM**, short for BugTM\(_S\), extends ConAir through software transactional

---

\(^1\) This dissertation’s implementation is based on Intel TSX. However, the principles apply to other vendors’ HTM implementations.
memory (STM) version-management techniques. When a failure is to happen, just like ConAir, a `longjmp` rolls back register content of the failing thread to the latest `setjmp`. Different from ConAir, selected memory checkpoints\(^2\) are conducted at some `setjmp` locations, offering the option to rollback some memory content before re-execution. Furthermore, BugTM\(_S\) conducts *deferred-write* code transformation that delays some shared-variable writes to further address ConAir’s failure-recovery limitations.

**Hybrid BugTM**, short for BugTM\(_{HS}\), uses HTM and `setjmp/longjmp` together to help failure recovery. BugTM\(_{HS}\) inserts both `setjmp/longjmp` and HTM APIs into software, with the latter inserted only when beneficial (i.e., when able to extend re-execution regions). When a failure is going to happen, the rollback is carried out through transaction abort if under an active transaction or `longjmp` otherwise.

These three BugTM designs offer different performance and recovery-capability tradeoffs, as illustrated by Figure 1.5.

BugTM\(_H\) and BugTM\(_S\) both support longer re-execution regions than ConAir by allowing shared-variable writes inside re-execution regions, and hence both achieve better recovery capability than ConAir. BugTM\(_H\) tends to have better recovery capability than BugTM\(_S\) benefiting from HTM, which we will explain in details in chapter 3.4 and chapter 3.6. On the other hand, BugTM\(_H\) and BugTM\(_S\) are both slower than ConAir. For BugTM\(_H\), this is

\(^2\) That is, backing up selected variables using undo logs.
because HTM costs more than setjmp/longjmp. For BugTM$_S$, this is because BugTM$_S$ takes extra lightweight memory checkpoints. Overall, their performance is still good, much better than full-blow memory checkpointing.

BugTM$_{HS}$ provides performance almost as good as ConAir and recovery capability even better than BugTM$_H$ by carefully combining BugTM$_H$ and ConAir.

(2) Automatically fix concurrency bugs via bypassing computation semantics

Along the direction of efficient concurrency bug fixing, we present a tool, BFix, that automates the whole process of fixing concurrency bugs via computation bypassing—skipping failure-inducing computation under bug-triggering contexts. BFix uses static analysis and code transformation to automatically decide which computation to bypass, to synthesize the computation-bypassing predicate, and to prove the correctness of the patch. Bypassing a group of operations clearly changes the sequential semantics of the running instance; how to make sure such execution behavior is acceptable and hence not lead software to invalid or unknown state is crucial to prove bypassing patch correctness. In this dissertation, a bypassing patch is considered to be correct only when it can help fix the reported concurrency bug as well as guarantee not to introduce new semantics, which means the bypassing semantics should already exist in the original software.

BFix uses the following insight that is unique to concurrent programs to tackle the core challenge of proving the patch correctness. Instead of trying to prove that the behavior of the patched program $P'$ under a buggy timing $T$, denoted as $P'_T$, is equivalent with the behavior of the original program $P$ also under $T$, denoted as $P_T$ (which is impossible as $P_T$ is an execution failure), BFix aims to prove that $P'_T$ is equivalent with the behavior of $P$ under a different non-buggy timing $T'$, denoted as $P_{T'}$.

Take the bypassing patch in Figure 1.4 as an example. It is difficult to prove the patch correctness by looking at one instance of FindLookup alone—skipping the reset of mStatus clearly would change the semantics of this individual function call. Fortunately, in the
original program, Thread-2 periodically invokes FindLookup and sometimes also skips the reset of mStatus in FindLookup. For the ease of discussion, Figure 1.6(a) illustrates how the patched program $P'$ performs under a bug-triggering timing $T$: at the first invocation of FindLookup, which occurs at the original buggy timing, the reset of mStatus is skipped; then, at the second invocation of FindLookup, the reset of mStatus is successfully done. This execution behavior is actually equivalent with the original program $P$ under a slightly different timing $T'$, as shown in Figure 1.6(b): the first invocation of FindLookup occurs a little bit earlier, right before ProcessLookup is invoked, and hence the reset of mStatus is skipped as the completeness checking of lookup returns FALSE. Finally, at the second invocation of FindLookup, the reset of mStatus is successfully done.

\begin{itemize}
  \item \textbf{// Thread-1}
  \item ProcessLookup:
    \begin{itemize}
      \item lookup->mStatus = COMPLETE;
    \end{itemize}
  \item NS_ASSERT(lookup->mStatus == COMPLETE);
  \item FindLookup:
    \begin{itemize}
      \item if (lookup->mStatus == COMPLETE && (lookup->mProcess == 0))
        \begin{itemize}
          \item lookup->mStatus = NEW
        \end{itemize}
    \end{itemize}
  \item \textbf{// Thread-2}
  \item FindLookup:
    \begin{itemize}
      \item if (lookup->mStatus == COMPLETE)
        \begin{itemize}
          \item lookup->mStatus = NEW
        \end{itemize}
    \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textbf{// Thread-1}
  \item ProcessLookup:
    \begin{itemize}
      \item lookup->mStatus = COMPLETE;
    \end{itemize}
  \item NS_ASSERT(lookup->mStatus == COMPLETE);
  \item FindLookup:
    \begin{itemize}
      \item if (lookup->mStatus == COMPLETE && (lookup->mProcess == 0))
        \begin{itemize}
          \item lookup->mStatus = NEW
        \end{itemize}
    \end{itemize}
  \item \textbf{// Thread-2}
  \item FindLookup:
    \begin{itemize}
      \item if (lookup->mStatus == COMPLETE)
        \begin{itemize}
          \item lookup->mStatus = NEW
        \end{itemize}
    \end{itemize}
\end{itemize}

(a) Execution timeline for the patched version when bug happens  \hspace{1cm} (b) Execution timeline for original program when FindLookup is conducted early

Figure 1.6: Execution timeline for the patched version in bug-triggering timing and the original program in a bug-free timing of Figure 1.4. All timeline figures in this dissertation use the same symbols: gray font means FALSE Boolean expression; rectangle represents skipped regions.

Following this insight, BF\textsc{ix} first identifies three common patterns where the behavior of skipping operations $O$ in the patched program $P'$ under timing $T$ is equivalent with the original program $P$ under timing $T'$: (1) overwritten pattern, where $O$'s side effect is overwritten by following writes and hence $P'_T$ is equivalent with skipping $O$ in $P_T$; (2) conditional pattern, where $O$ is skipped due to condition checking in both $P'_T$ and $P_T$ like that in Figure 1.6; and (3) exclusive pattern, where the repeated execution of $O$ could be
skipped in both $P_T'$ and $P_T$ since the exclusive property guarantees only its first execution instance to succeed and repeated invocations are redundant, leading to software crash. The details are presented in chapter 4.4.

Guided by these three patterns, BFix bug fixing goes through the following steps.

First, given a bug report and the program source code, BFix first uses static analysis to decide whether any one of the three equivalence patterns might be suitable for the concurrency bug under fixing (see chapter 4.5.2 for details).

Second, if the answer is yes, BFix carefully identifies the program statements to be bypassed. Particularly, BFix makes sure that (1) enough operations are skipped so that the original bug’s failure symptom will disappear, and (2) operations that depend on to-be-skipped operations are also skipped so as not to introduce new bugs. BFix makes the above decisions by analyzing the bug report and the program source code, then following the particular equivalence pattern the bug matches with. The details are presented in chapter 4.5.2.

Third, BFix synthesizes the condition predicate that decides when to bypass the above statements. Intuitively, failure-inducing operations only need to be bypassed under the bug-triggering timing, like when the FireStop is concurrently executed with the reset of mStatus in Figure 1.4. BFix carefully analyzes the program to decide whether new variables are needed or whether any existing expressions can be used to represent the bug-triggering timing. It then generates the condition checking and predicate-variable updates, if needed, accordingly. The details are presented in chapter 4.5.3.

Fourth, and finally, BFix merges related bug patches to further optimize performance and patch simplicity (chapter 4.5.4).

Although computation-bypassing does not apply for all concurrency bugs, BFix helps to automatically generate high quality patches for a large family of concurrency bugs, greatly complementing existing auto-fixing tools. We have evaluated BFix on eight real-world con-
currency bugs. The patches generated BFix eliminate 5 out of them and significantly decrease the failure probability in the other 3 cases. Comparing these patches with synchronization-adding patches generated by CFix [28] and synchronization-moving patches generated by HFix [39] generated by previous auto-fixing tools [28, 39], these bypassing patches generated by BFix offer much better performance and code simplicity. The experimental results show that the quality of BFix patches is similar to the manual bypassing patches in terms of patch correctness, patch performance and patch simplicity.

### 1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 introduces previous work on empirical study of concurrency bugs, concurrency bug detection and fixing, concurrency bug prevention and failure recovery. Chapter 3 presents our efficient solution BugTM for concurrency bug failure recovery during production runs. Chapter 4 introduces our efficient fixing tool BFix for concurrency bugs during code maintenance. Chapter 5 concludes this dissertation and discusses future research work.
CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter presents the background and the related work of this dissertation. Chapter 2.1 introduces concurrency bugs and the types we are working on. Chapter 2.2 discusses related empirical study of concurrency bugs. Chapter 2.3 discusses previous work on concurrency bug detection. Chapter 2.4 discusses how to prevent concurrency-bug failures, while chapter 2.5 discusses existing approaches on failure recovery. Chapter 2.6 introduces related work of concurrency bug fixing. Chapter 2.7 introduces condition synthesis in program repair.

2.1 Concurrency bug

Concurrency bugs are one notorious type of software bugs. These timing-related bugs manifest non-deterministically, and hence are extremely difficult to detect, diagnose, and fix. Concurrency bug problems are becoming more difficult and important with the prevalence of multi-core hardware [6].

In this dissertation, we mainly focus on fighting with three types of concurrency bugs, atomicity violation (AV), order violation (OV) and deadlock (DD).

An atomicity violation happens when the atomicity of a code region $C$ is unexpectedly violated, such as the bug shown in Figure 1.3. If Thread-2 assigns NULL to $s \rightarrow \text{table}$ between operations $A_1$ and $A_2$ in Thread-1, sanity checking in $A_2$ would fail, and fatal-error message would be thrown. However, no matter this assignment is executed before $A_1$ or after $A_2$, program can run as normal without bug manifestation.

An order violation happens when an operation $A$ unexpectedly executes after, instead of before, operation $B$, such as the bug in Figure 2.1. If $A$ in Thread-2 is executed before $B$ in Thread-1, the program can successfully execute. If the order is opposite ($B$ is executed before $A$), Thread-1 would throw assertion failure.
Deadlock bugs occur when different threads each holds resources and circularly waits for each other. In Figure 2.2, Thread-1 and Thread-2 acquire nlock and slock in reversed orders and lead to deadlock.

```c
1 //Thread-1
2 Close(){
3 ... 
4 Lock(&nlock);
5  
6 drive->Close();
7  
8 Lock(&slock);
9 ... 
10 }

1 //Thread-2
2 Shutdown(){
3 ... 
4 Lock(&slock);
5  
6 if(nSockets[i]){ 
7  
8 Lock(&nlock);
9 } 
10 }
```

Figure 2.2: A real-world DD bug (simplified from HawkNL)

### 2.2 Empirical study of concurrency bugs

Past research studies looked at real-world concurrency bugs [18, 47] and synchronization-related code changes [20, 55, 62, 75]. They provide important guidance for concurrency bug detection.

Several empirical studies have looked at concurrency-bug patches with different focuses. One focuses on the correctness of intermediate patches [78]; one studies patches to understand how file systems evolve [44]; one focuses on how transactional memory might help simplify concurrency bug patches [73], this is strongly related with our BugTM work. They manually apply transactional memory on concurrency bugs. However, as shown in chapter 3.1, our work is much more challenging.

Lu et al. [46] summarize concurrency-bug patches into five categories: condition check,
code switch, design change, add/change locks, and others. Their study gives a sense how the manual patches look like. Another study conducted by Cerney et al. [9] check the patches of concurrency bugs in Linux device drivers. Liu et al. [39] provide an in-depth understanding of concurrency bug patches, through a thorough study of manual patches for 77 real-world concurrency bugs. They find the patches generated by auto fixing tools are always different with the ones developed by human. Their findings motivates our fixing tool BFix.

2.3 Concurrency bug detection

Many automated detection tools have been proposed for a variety of concurrency bugs, including data races [8, 17, 24, 30, 53, 64, 71, 80], atomicity violations [48, 50], order violations [19, 67, 85], and deadlocks [74]. These tools aim to discover bugs during in-house testing and are not a good fit for production-run deployment — they often incur large overhead (e.g., 10X slowdowns) and cannot provide the desired bug/failure coverage. And, the software availability and reliability are not improved if only focusing on detection. Bug fixing is the fundamental way to eliminate the concurrency bug and improve the software availability and reliability.

2.4 Concurrency-bug failure prevention

The prevention approach works by perturbing the execution timing, hoping that failure-triggering interleavings would not happen. It either relies on prior knowledge about a bug/failure [29, 49] to prevent the same bug from manifesting again, or relies on extensive off-line training [81, 79] to guide the production run towards likely failure-free timing. It is not suitable for avoiding production-run failures caused by previously unknown concurrency bugs. Particularly, the LiteTx work [79] proposes hardware extensions that are like lightweight HTM (i.e., without versioning or rollback) to constrain production-run thread interleavings,
proactively prohibiting interleavings that have not been exercised during off-line testing.

2.5 Failure recovery

Rollback and re-execution have long been a valuable recovery [60, 71] and debugging [14, 31, 57, 70] technique. Many rollback-reexecution techniques target full system/application replay, hence have huge run-time overhead.

Feather-weight re-execution based on idempotency has been used before for recovering hardware faults [12, 16]. Using it to help recover from concurrency-bug failures was recently pioneered by ConAir [82]. Our BugTM greatly improved ConAir. BugTM_H and ConAir use not only different rollback/reexecution mechanisms, but also completely different static analysis and code transformation. The \texttt{setjmp} and \texttt{longjmp} used by ConAir have different performance and correctness implications from \texttt{StartTx}, \texttt{CommitTx}, and \texttt{AbortTx}, which naturally led to completely different designs in BugTM_H and ConAir.

Recent work leverages TM to help recover from transient hardware faults [32, 38, 77]. Due to the different types of faults/bugs these tools and BugTM are facing, their designs are different from BugTM. They wrap the whole program into transactions, which inevitably leads to large overhead (around 100% overhead [32, 77]) or lots of hardware changes to existing HTM [38], and different design about how/where to insert Tx APIs. They use different ways to detect and recover from the occurrence of faults, and hence have different Tx abort handling from BugTM. They either rely on non-existence of concurrency bugs to guarantee determinism [32] or only apply for single-threaded software [38, 77], which is completely different from BugTM.
2.6 Concurrency bug fixing

Several researches are proposed to help automatically generate patches for concurrency bugs [27, 28, 40, 74]. They work at off-line and rely on accurate bug-detection results. Tools like AFix [27] and CFix [28], automate the manual bug fixing process for programmers and strengthen it with the static analysis techniques such as the path analysis, the reduction of subsumed bugs, and then merging of overlapping bugs. Grail [40] introduces context-aware concepts. To fix AV bugs, locks in Grail only has effect when shared memory is really accessed during run time. Axis [42] applies Petri to convert concurrency bug fixing to constraints solving, which makes patches are provable.

A recent work [25] proposes a data-privatization technique to automatically avoid some read-after-write and read-after-read atomicity violations. When a thread may access the same shared variable with no blocking operations in between, this technique would create a temporary variable to buffer the result of the earlier access and feed it to the later read access.

Those tools share the same feature — they try to eliminate the buggy timing, which is fundamentally different from BFix. And as HFix [39] points out those patches are always far away from what developers propose. HFix tries to leverage existing synchronizations, moving lock to extend the critical region and moving pthread_join to enforce the operations’ execution order when those operations exist in child/parent threads separately. It complements the existing auto-fixing tools. However, it focuses on different fixing strategy from our BFix.

2.7 Condition synthesis in program repair

Even condition synthesis is firstly introduced to the automatic concurrency bug fixing tool, it is widely used in program repair, especially in search-and-validate program repair area. They
usually firstly generate some possible patches, and verify the correctness by running the test cases. However they can not give any correctness guarantee. Tools like GenProg [35], which uses genetic algorithm to randomly generate condition checking, SPR [43] which proposes a heuristic strategy to search the repair space defined by its rich set of repair schemes, most patches they generate are wrong and can not be deployed. Even for the state-of-the-art approach, ACS [76] which uses topological dependency/data mining to choose condition variable, it can only achieve 73.9 % precision rate. That is not the whole story. Even for the plausible patches, Qi et al. [59] show that if condition can potentially lead to bugs if not well handled. Unlike the condition synthesis in software testing, BFix provides provably correct condition synthesis.
CHAPTER 3
APPLYING TRANSACTIONAL MEMORY FOR CONCURRENcy-BUG FAILURE RECOVERY IN PRODUCTION RUNS

Along the direction of concurrency bug failure recovery, we present BugTM, a TM-inspired techniques to help automatically recover concurrency-bug failures during production runs. It is almost as powerful as the traditional rollback recovery and as lightweight as ConAir. In this chapter, we will introduce the details of BugTM. Chapter 3.1 discusses the challenges of applying transactional memory for concurrency bug failure recovery. Chapter 3.2 introduces the background of transactional memory. Chapter 3.3 introduces the state-of-the-art concurrency bug failure recovery technique – ConAir. Chapter 3.4 presents the first design BugTM_H, while chapter 3.6 and chapter 3.7 present our second and third designs BugTM_S and BugTM_HS. Chapter 3.5 introduces the reason why BugTM_S and BugTM_HS are proposed. Chapter 3.8 introduces the failure diagnosis BugTM provides. Chapter 3.9 introduces our methodology for experiments. Chapter 3.10 shows our experimental results. Chapter 3.11 discussions how to deploy BugTM into real systems. Chapter 3.12 concludes the first part of this dissertation.

3.1 Challenges for applying transactional memory for concurrency bug failure recovery

At the first glance, the opportunity of HTM for concurrency bug failure recovery seems obvious, as HTM provides a powerful mechanism for concurrency control and rollback-reexecution.

Previous work [73] also showed that TM can be used to manually fix concurrency bugs after they are detected.
However, automatically inserting HTMs to help tackle unknown concurrency bugs during production runs faces many challenges not encountered by manually fixing already detected concurrency bugs off-line:

**Performance challenges:** High frequency of transaction uses would cause large overhead unacceptable for production runs. Unsuitable content of transactions, like trapping instructions\(^1\), high levels of transaction nesting, and long loops, would also cause performance degradation due to repeated and unnecessary transaction aborts.

**Correctness challenges:** Unpaired transaction-start and transaction-commit could cause software to crash. Deterministic aborts, such as those caused by trapping instructions, could cause software to hang if not well handled. We need to guarantee these cases do not happen and ensure software semantics remains unmodified.

**Failure recovery challenges:** In order for HTM to help recovery, we need to improve the chances that software executes in a transaction when a failure happens and we need to carefully design HTM-abort handlers to correctly process the corresponding transaction aborts.

BugTM addresses these challenges by its carefully designed and carefully inserted, based on static program analysis, HTM start, commit, and abort routines. It instantiates three designs BugTM\(_H\), BugTM\(_S\), BugTM\(_{HS}\). In chapter 3.4, 3.6, 3.7, we will show details and how BugTM addresses those challenges from HTM perspective.

### 3.2 Transactional Memory (TM)

TM is a widely studied parallel programming construct [22, 23]. Developers can wrap a code region in a transaction (Tx), and the underlying TM system guarantees its atomicity, consistency, and isolation. Hardware transactional memory (HTM) provides much better

\(^1\) Certain instructions such as system calls will deterministically cause HTM abort and are referred to as trapping instructions.
performance than its software counterpart (STM), and has been implemented in IBM [21], Sun [15], and Intel commercial processors [1].

In this dissertation, we focus on Intel Transactional Synchronization Extensions (TSX). TSX provides a set of new instructions: \texttt{XBEGIN}, \texttt{XEND}, \texttt{XABORT}, and \texttt{XTEST}. We will denote them as \texttt{StartTx}, \texttt{CommitTx}, \texttt{AbortTx}, and \texttt{TestTx}, respectively for generality. Here, \texttt{CommitTx} may succeed or fail with the latter causing Tx abort. \texttt{AbortTx} explicitly aborts the current Tx, which leads to Tx re-execution unless special fallback code is provided. \texttt{TestTx} checks whether the current execution is under an active Tx.

There are multiple causes for Tx aborts in TSX. \textit{Unknown abort} is mainly caused by trapping instructions, like exceptions and interrupts (abort code \texttt{0x00}). \textit{Data conflict abort} is caused by conflicting accesses from another thread that accesses (writes) the write (read) set of the current Tx (abort code \texttt{0x06}). \textit{Capacity abort} is due to out of cache capacity (abort code \texttt{0x08}). \textit{Nested transaction abort} happens when there are more than 7 levels Tx nesting (abort code \texttt{0x20}). \textit{Manual abort} is caused by \texttt{AbortTx} operation, with programmers specifying abort code.

### 3.3 ConAir

ConAir is a static code transformation tool built upon LLVM compiler infrastructure [34]. It is a state-of-the-art concurrency bug failure recovery technique as discussed in chapter 1. We describe some techniques and terminologies that will be used in later chapters below.

\textbf{Recovery capability limitations and killing writes} \texttt{w}_{\text{kill}} ConAir does not allow its re-execution regions to contain any writes to shared variables. Consequently, many of its re-execution points (i.e., \texttt{setjmp} locations) are placed right after shared-variable writes, which we refer to as \textit{killing writes} or \texttt{w}_{\text{kill}}. In many cases, ConAir could not recover from a failure because a successful recovery demands killing writes to be re-executed.
ConAir fundamentally cannot recover any RAW\textsuperscript{2} violations (e.g., the bug in Figure 1.3) and WAR violations, as Table 3.1 shows. The reason is that the (RA)W and W(AR) have to be re-executed for successful recoveries, but they are killing writes for ConAir.

Even for those root-cause types that ConAir can handle in Table 3.1, its recovery capability is limited, because a killing write may exist between the failure location and the ideal re-execution point. For example, the RAR atomicity violation in Figure 3.1 cannot be recovered by ConAir due to the write to *buf on Line 3. If Line 3 did not exist, ConAir could have rolled back Thread-1 to re-execute Line 2 and gotten around the failure. With Line 3, ConAir can only repeatedly re-execute the strcat on Line 4, with no chance of recovery.

**Failure instruction** $f$ ConAir automatically identifies where failures may happen so that rollback APIs can be inserted right there. This identification is based on previous observations that $\geq 90\%$ of concurrency bugs lead to four types of failures [83]: assertion violations, segmentation faults, deadlocks, and wrong outputs. BugTM will reuse this technique to identify potential failure locations, denoted as failure instructions $f$ in the remainder of the dissertation. Specifically, ConAir identifies the invocations of \_\_assert\_fail or other sanity-check macros as failure instructions for assertion failures. ConAir then automatically transforms software to turn segmentation faults and deadlocks into assertion failures: ConAir automatically inserts assertions to check whether a shared pointer variable $v$ is null right before $v$’s dereference and check whether a pointer parameter of a string-library function is null right before the library call; ConAir automatically turns lock functions into time-out...
lock functions, with a long timeout indicating a likely deadlock failure, and inserts assertions accordingly. ConAir can help recover from wrong output failures as long as developers provide output specifications using assertions.

3.4 BugTM_H

3.4.1 High-Level Design

We discuss our high-level idea about where to put Txs, and compare with some strawman ideas based on performance and failure-recovery capability.

**Strawman approaches** One approach is to chunk software to many segments and put every segment inside a hardware Tx [51]. This approach could avoid some atomicity violations, the most common type of concurrency bugs. However, it does not help recover from order violations, another major type of concurrency bugs. Furthermore, its excessive use of Txs will lead to unacceptable overhead for production-run deployment. Another approach is to replace all lock critical regions with Tx. However, this approach will not help eliminate many failures that are caused by missing lock.

**Our approach** In BugTM_H, we selectively put hardware Txs around places where failures may happen, like the invocation of an `_assert_fail`, the dereference of a shared pointer, etc. This design has the potential to achieve good performance because it inserts Txs only at selected locations. It also has the potential to achieve good recovery capability because in theory it can recover from all common types of concurrency bugs, as shown in Table 3.1 and explained below.

An atomicity violation (AV) happens when the atomicity of a code region C is unexpectedly violated, such as the bug shown in Figure 1.3. It contributes to more than 70% of non-deadlock concurrency bugs based on empirical studies [46], and can be further categorized into 4 sub-types depending on the nature of C, as demonstrated in Table 3.1.
Table 3.1
Common types of concurrency bugs and how BugTMH and ConAir attempt to recover from them

<table>
<thead>
<tr>
<th>Types</th>
<th>Atomicity Violations</th>
<th>Order Violations</th>
<th>Deadlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) RAR</td>
<td>(b) RAW</td>
<td>(c) WAR</td>
</tr>
<tr>
<td>R/W: read/write to a shared variable; thick vertical line: the execution of one thread; dashed arrowed line: the re-execution region of BugTMH; thin arrowed line: the re-execution region of ConAir; explosion symbol: a failure; -: cannot recover; ✓: sometimes can recover if the recovery does not require re-executing shared-variable writes; ✓✓: mostly can recover. The recovery procedure under BugTMHS is a mix of BugTMH and ConAir and hence is not shown in table.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R/W: read/write to a shared variable; thick vertical line: the execution of one thread; dashed arrowed line: the re-execution region of BugTMH; thin arrowed line: the re-execution region of ConAir; explosion symbol: a failure; -: cannot recover; ✓: sometimes can recover if the recovery does not require re-executing shared-variable writes; ✓✓: mostly can recover. The recovery procedure under BugTMHS is a mix of BugTMH and ConAir and hence is not shown in table.

Conflicting accesses would usually trigger a rollback recovery before the failure occurs, shown by the dashed arrow lines in Table 3.1(a)(b)(c), benefiting from the strong atomicity guarantee of Intel TSX — a Tx will abort even if the conflicting access comes from non-Tx code. For the bug shown in Figure 1.3 (an RAW atomicity violation), if we put the code region in Thread-1 inside a Tx, the interleaving NULL assignment from Thread-2 would trigger a data conflict abort in Thread-1 before the if statement has a chance to read the NULL. The re-execution of Thread-1 Tx will then re-assign the valid value to s \rightarrow table for the if statement to read from, successfully avoiding the failure.

OV bugs are another main type of concurrency bugs. It happens when an instruction A unexpectedly executes after, instead of before, instruction B, such as the bug in Figure 2.1. Different from AVs, conflicting memory accesses related to OVs may not all happen inside a small window. In fact, A may not have executed when a failure occurs in the thread of B. Consequently, the Tx abort probably will be triggered by a software failure, instead of a conflicting access, depicted by the dashed arrow in Table 3.1(e). Fortunately, the rollback reexecution will still give the software a chance to correct the unexpected ordering and recover from the failure. Take the bug shown in Figure 2.1 as an example. If we put
a hardware Tx in Thread-1, when order violation leads to the assertion failure, the Tx will abort, rollback, and re-execute. Eventually, the pointer ptr will be initialized and the Tx will commit.

Deadlock bugs occur when different threads each holds resources and circularly waits for each other. As shown in Table 3.1(f), it can be recovered by Tx rollback and re-execution too, as long as deadlocks are detected.

Of course, BugTM_H cannot recover from all failures, because some error-propagation chains cannot fit into a HTM Tx, which we will discuss more in chapter 3.10.

Next, we will discuss in details how BugTM_H surrounds failure sites with hardware Txs—how to automatically insert StartTx, CommitTx, AbortTx, and fallback/retry code into software, while targeting three goals: (1) good recovery capability; (2) good run-time performance; (3) not changing original program semantics.

3.4.2 Design about AbortTx

BugTM_H uses the same technique as ConAir to identify where failures would happen as discussed in chapter 3.3. BugTM_H puts an AbortTx wrapper function my_xabort right before every failure instruction f, so that a Tx abort and re-execution is triggered right before a failure manifests. my_xabort uses a unique abort code 0xFF for its AbortTx operation (as shown in Figure 3.2), so that BugTM_H can differentiate different causes of Tx aborts and handle them differently.
3.4.3 Design about StartTx and CommitTx

**Challenges** We elaborate on two key challenges in placing StartTx and CommitTx, and explain why we cannot simply insert well-structured atomic blocks (e.g., `__transaction_atomic` supported by GCC) into programs.

First, poor placements could cause frequent Tx aborts. Trapping instructions (e.g., system calls) and heavy TM nesting (>7 level) deterministically cause aborts, while long Txs abort more likely than short ones due to timer-interrupts and memory-footprint threshold. These aborts hurt not only performance, but also recovery — deterministic aborts of a Tx will eventually force us to execute the Tx region\(^3\) in non-transaction mode, leaving no hope for failure recovery.

Second, poor placements could cause unpaired execution of StartTx and CommitTx, hurting both correctness and performance. When CommitTx executes without StartTx, the program will crash; when StartTx executes without a pairing CommitTx, its Tx will repeatedly abort.

Taking Figure 3.3 as an example, we want to put \(A_1\) and \(A_2\), both accessing global variable \(G\), into a Tx together with `__assert_fail` on Line 6 for failure recovery. However, if we naively put StartTx on Line 2 and CommitTx on Line 12, forming a well structured atomic block, correct runs will incur repeated Tx aborts and huge slowdowns due to I/Os on Line 10. Simply moving CommitTx to right after Line 4 and keeping StartTx on Line 2 still will not work — when else is taken, the earlier StartTx has no pairing CommitTx and the Tx still aborts due to I/Os.

We address the first challenge by carefully placing StartTx and CommitTx. We address the second challenge mainly through our StartTx, CommitTx wrapper-functions.

**Where to StartTx and CommitTx** The design principle is to minimize the chance of aborts that are unrelated to concurrency bugs, tackling the first challenge above. BugTM\(_H\) achieves

---

3. We will refer to the code region between our `my_xbegin` and `my_xend` as a Tx region, which may be executed in transactional mode.
void func(...){
  G = g; //A1
  if(!G){ //A2
    __assert_fail; //f: failure instr.
  } else{
    IO(...); //computation & I/O
  }
}

void func(...){
  G = g;
  if(!G){
    __assert_fail;
  } else{
    IO(...);
  }
}

Figure 3.3: A toy example adapted from Figure 1.3 (left-side) and its BugTM transformation (right-side)

this by making sure that its Txs do not contain function calls, which avoids system calls and many trapping instructions, or loops, which avoids large memory footprints. The constraint of not containing function calls will be relaxed in chapter 3.4.5.

Specifically, for every failure instruction $f$ inside a function $F$, BugTM$_H$ puts a StartTx wrapper function right after the first function call instruction or loop-exit instruction or the entrance of $F$, whichever encountered first along every path tracing backward from $f$ to the entrance of $F$. BugTM$_H$ puts CommitTx wrapper functions right before the exit of $F$, every function call in $F$, and every loop header instruction in $F$, unless the corresponding loop contains a failure instruction, in which case we want to extend re-execution regions for possible failures inside the loop.

Analysis for different failure instructions may decide to put multiple StartTx (CommitTx) at the same program location. In these cases, we will only keep one copy.

For the toy example in Figure 3.3, the intra-procedural BugTM$_H$ identifies Line 2 to put a StartTx, and identifies Line 9 and 12 to put CommitTx, as shown in the figure.

How to StartTx and CommitTx The above algorithm does not guarantee one-to-one pairing of the execution of StartTx and CommitTx, the second challenge discussed above. BugTM$_H$ addresses this through TestTx checkings conducted in my_xbegin and my_xend, BugTM$_H$ wrapper functions for StartTx and CommitTx. That is, StartTx will execute only when there is no active
if (_xtest())
  _xend(); //terminate an active transaction

Figure 3.4: BugTM<sub>H</sub>CommitTx wrapper function (my_xend)

if (_xtest() == 0){  //no active Tx
  Retrytimes = 0;
  prev_status = -1;
  retry: if((status = _xbegin()) == _XBEGIN_STARTED){ //Tx starts
    //Tx starts
  } else {
    //abort fallback handler, no active Tx at this point
    Retrytimes++;
    if(status==0x00||status==0x88){ //unknown or capacity abort
      if(!(prev_status==0x00 && status==0x00) &&
        !(prev_status==0x08 && status==0x88))
        { prev_status=status; goto retry; }
      else if(status==0x06 || status==0xFF){
        if(Retrytimes < RetryThreshold)
          {prev_status=status; goto retry; }
      } else {
        //continue execution in non-Tx mode
      }
    } else {
      //abort fallback handler, no active Tx at this point
      Retrytimes++;
      if(status==0x00||status==0x88){
        if(!(prev_status==0x80 && status==0x00) &&
          !(prev_status==0x08 && status==0x88))
          { prev_status=status; goto retry; }
        else if(status==0x06 || status==0xFF){
          if(Retrytimes < RetryThreshold)
            {prev_status=status; goto retry; }
        } else {
          //continue execution in non-Tx mode
        }
      }
  }

Figure 3.5: BugTM<sub>H</sub>StartTx wrapper function (my_xbegin)

Txs, as shown in Figure 3.5; CommitTx will execute only when there exists an active Tx, as shown in Figure 3.4.

Overall, our design so far satisfies performance, correctness, and failure-recovery goals by guaranteeing a few properties. For performance, BugTM<sub>H</sub> guarantees that its Txs do not contain system/library calls or loops or nested Txs, and always terminate by the end of the function where the Tx starts. For correctness, BugTM<sub>H</sub> guarantees not to introduce crashes caused by unpairing CommitTx. For recovery capability, BugTM<sub>H</sub> makes the best effort in letting failures occur under active Txs.

3.4.4 Design for fallback and retry

Challenges It is not trivial to automatically and correctly generate fallback/retry code for all Txs inserted by BugTM<sub>H</sub>. Since many Tx aborts may be unrelated to concurrency
bugs, inappropriate abort handling could lead to performance degradation, hangs, and lost failure-recovery opportunities.

**Solutions** BugTM\textsubscript{H} will check the abort code and react to different types of aborts differently. Specifically, BugTM\textsubscript{H} implements the following fallback/retry strategy through its \texttt{my\_xbegin} wrapper (Figure 3.5).

Aborts caused by \texttt{AbortTx} inserted by BugTM\textsubscript{H} indicates software failures. We should re-execute the Tx under HTM, hoping that the failure will disappear in retry (Line 14–17). To avoid endless retry, BugTM\textsubscript{H} keeps a retry-counter \texttt{Retrytimes} (Figure 3.5). This counter is configurable in BugTM\textsubscript{H}, with the default being 1000000.

Data conflict aborts (Line 14–17) are caused by conflicting accesses from another thread. They are handled in the same way as above, because they could be part of the manifestation of concurrency bugs.

Unknown aborts and capacity aborts (Line 9–13) have nothing to do with concurrency bugs or software failures. In fact, the same abort code may appear repeatedly during retries, causing performance degradation without increasing the chance of failure recovery. Therefore, the fallback code will re-execute the Tx region in non-transaction mode once these two types of aborts are observed in two consecutive aborts. Nested Tx aborts would not be encountered by BugTM\textsubscript{H}, because BugTM\textsubscript{H} Txs are non-nested.

The above wrapper function not only implements fallback/retry strategy, but also allows easy integration into the target software, as demonstrated in Figure 3.3.

3.4.5 **Inter-procedural Designs and Others**

The above algorithm allows no function calls or returns in Txs, keeping the whole recovery attempt within one function \texttt{F}. This is too conservative as many functions contain no trapping instructions and could help recovery.

To extend the re-execution region into callees of \texttt{F}, we put \texttt{my\_xend} before every system/li-
brary call instead of every function call. To extend the re-execution region into the callers of $F$, we slightly change the policy of putting $\text{my\_xbegin}$. When the basic algorithm puts $\text{my\_xbegin}$ at the entrance of $F$, the inter-procedural extension will find all possible callers of $F$, treat the callsite of $F$ in its caller as a failure instruction, and apply $\text{my\_xbegin}$ insertion and $\text{my\_xend}$ insertion in the caller.

We then adjust our strategy about when to finish a BugTM$_H$ Tx. The basic BugTM$_H$ may end a Tx too early: by placing $\text{my\_xend}$ before every function exit, the re-execution will end in a callee function of $F$ before returning to $F$ and reaching the potential failure site in $F$. Our adjustment changes the $\text{my\_xend}$ wrapper inserted at function exits, making it take effect only when the function is the one which starts the active Tx.

Finally, as an optimization, we eliminate Txs that contain no shared-variable reads the failure instruction $f$ has control or data dependency on. In these cases, the execution and outcome of $f$ is deterministic during re-execution, and hence the failure cannot be recovered.

### 3.5 Roadmap for Further Exploration

HTM in BugTM$_H$ and setjmp/longjmp in previous state-of-the-art ConAir [82] are almost at the two ends of the design spectrum. While the former provides much better recovery capability, it has higher overhead than the latter. Furthermore, HTM disallows certain operations in a Tx (e.g. malloc, memcpy, pthread_cond_wait), which could be addressed by software techniques [61, 72].

To further explore the design space, the next two chapters will explore two designs to combine the strengths of BugTM$_H$ and ConAir.

One design, BugTM$_S$, is to implement some functionalities of HTM, tailored for concurrency-bug recovery, as software extensions for ConAir. Looking at the three TM principles of conflict detection, conflict resolution, and version management, we decide to try the latter two and give up conflict detection, as conflict detection is too expensive to implement in soft-
ware. The conflict-resolution technique for shared-variable reads can provide extra options for ConAir’s re-execution: reading the latest copy means delaying the current Tx (thread), whereas reading an earlier one using an undo log means delaying the conflicting one. The version-management technique for shared-variable writes, which completely does not exist in ConAir, can extend the types of regions that can be reexecuted for recovery in ConAir. This design will be discussed next in chapter 3.6.

The other design, BugTM$_{HS}$, directly combines ConAir and BugTM$_H$. This design is feasible as Intel TSX allows setjmp/longjmp to execute inside Txs. Therefore, we can actually apply BugTM$_H$ to a program already hardened by ConAir or any setjmp/longjmp recovery scheme and obtain the union of each component’s recovery capability, which we will discuss in more details in chapter 3.7.

We do not explore combining ConAir and BugTM$_S$, because BugTM$_S$ itself is already a direct extension of ConAir. We also do not explore combining BugTM$_H$ and BugTM$_S$, because this combination will be worse than combining BugTM$_H$ and ConAir (i.e., BugTM$_{HS}$) — in terms of performance, BugTM$_S$ is slower than ConAir; in terms of recovery capability, the advantage of BugTM$_S$ over ConAir is mostly overshadowed by BugTM$_H$.

### 3.6 BugTM$_S$

This chapter will focus on extending the basic setjmp/longjmp recovery scheme ConAir with two TM techniques (1) deferred write version management; and (2) undo log rollback. Our implementation will not rely on HTM and is purely based on compiler techniques. The resulting tool BugTM$_S$ not only improves the failure recovery capability of ConAir with negligible performance impact, but also well complements BugTM$_H$ by offering better performance and more design flexibility at the cost of losing some recovery capability owned by BugTM$_H$. 

35
Figure 3.6: BugTM₅ deferred write transformation, denoted by ‘+’ ‘-’, makes a ConAir-unrecoverable bug recoverable.

3.6.1 Deferred Writes for Failure-Unrelated \( w_{\text{kill}} \)

Figure 3.6 shows an RAR atomicity violation, where the \texttt{NULL} assignment from Thread-2 could cause Thread-1 to crash at Line 7. Theoretically, ConAir can recover RAR atomicity-violation failures. However, with a killing write, which is actually unrelated to the bug, at Line 4, ConAir cannot extend its re-execution region to include both reads of \texttt{thd->proc} in Thread-1 and hence cannot recover from the failure.

To address this problem, BugTM₅ tries moving failure-unrelated \( w_{\text{kill}} \) to after the failure instruction, emulating the deferred write version-management technique in TM, so that the re-execution region can go beyond these killing writes.

**Feasibility checking** For each \( w_{\text{kill}} \) and the corresponding failure site \( f \), BugTM₅ checks two things: (1) whether moving \( w_{\text{kill}} \) would change program semantics; and (2) whether the moving will cut short other failure sites’ re-execution regions. If \( w_{\text{kill}} \) fails either checking, it is not moved.

The second checking is straightforward. For the first condition, BugTM₅ collects all instructions along any path from \( w_{\text{kill}} \) to \( f \), and checks whether there exists any write-after-write, read-after-write, or write-after-read dependency between any of such instruction with \( w_{\text{kill}} \).

If there is no such dependency, moving \( w_{\text{kill}} \) is guaranteed not to change program seman-
tics\textsuperscript{4}. If there exists such a dependency upon global/heap variables, we give up the moving. If the dependency is upon a stack variable, such as \texttt{buf} in Figure 3.6, we try code transformation to eliminate the dependency. Note that, since \texttt{wkill} writes to a shared variable, the stack variable dependency here must be a write-after-read dependency as the one between Line 4 and 5 in Figure 3.6.

To eliminate the write-after-read dependency between \texttt{wkill} and \texttt{i} on a stack variable \texttt{v}s, BugTM\textsubscript{S} will create temporary stack variable \texttt{tmp} to keep a copy of \texttt{v}s at the original code location of \texttt{wkill}, move \texttt{wkill}, and let the moved \texttt{wkill} read from \texttt{tmp} instead of \texttt{v}s, as demonstrated by Figure 3.6.

**Moving the \texttt{wkill}** To make sure the moved \texttt{wkill} will execute for the same number of times as in the original program, BugTM\textsubscript{S} conducts the following analysis and transformation:

First, check if \texttt{wkill} and \texttt{f} are inside one function \texttt{F} with neither inside a loop in \texttt{F}. If not, we give up the move.

Second, collect all the basic blocks \texttt{B} in \texttt{F} that are on path from \texttt{wkill} to \texttt{f}, and copy \texttt{wkill} to every edge that connects a basic block inside \texttt{B} to a basic block outside \texttt{B}. This guarantees that the new location of \texttt{wkill} will be touched exactly once in \texttt{F}, either immediately after \texttt{f} or immediately when there is no chance for \texttt{f} to execute. This way, \texttt{wkill} will get a chance to execute, even if \texttt{f} is not executed.

Third, a stack variable is introduced to make sure that the newly moved \texttt{wkill} would not execute if its original location was not touched, as shown in Figure 3.7.

Now BugTM\textsubscript{S} can recover from some ConAir-unrecoverable failures, like the one shown in Figure 3.6. It has almost no performance impact to the original ConAir, and guarantees to preserve program semantics.

\textsuperscript{4} This guarantee holds based on the fact that almost all architectures, including Alpha, ARM, POWER, SPARC, x86, and many others, allow compilers to reorder stores to execute after independent loads.
When killing writes are dependent upon the corresponding failure instruction, which are true for all RAW violations and WAR violations, deferred write does not apply. For these cases, BugTM\textsubscript{S} enhances ConAir by offering an extra mode of rollback: ConAir only rolls back registers for re-execution; BugTM\textsubscript{S} offers checkpointing and rolling back the content of selected shared-memory locations, emulating undo log in TM. This option can help recover from some Read-After-Write (RAW) atomicity violations, while preserving program semantics and introducing little overhead.

**Basic Algorithm** Figure 3.8(a) shows a toy example of RAW atomicity violation: if another thread changes the value of \( g_1 \) from 1 to 0 between the write on Line 1 and the read on Line 6, an assertion failure could happen. ConAir cannot recover from this failure, because the re-execution will start after the \( w_{\text{kill}} \) in Line 1 and can never change the failure-triggering
value returned by Line 6 in Figure 3.8(b). However, if the value of \( g_1 \) could be checkpointed right at Line 1, as shown in Figure 3.8(c), the failure could be recovered.

In general, taking a memory checkpoint is straightforward: create a local variable \( \text{ckpt}_g_1 \) and copy the right hand side of the \( g_1 \)-assignment to \( \text{ckpt}_g_1 \) right before \text{setjmp}.

Making re-execution use the checkpointed values can be achieved through code transformation. The return value of \text{setjmp} is -1 only when it is jumped to from a \text{longjmp}, indicating re-execution. As shown in Figure 3.8(c) Line 5–8, BugTM\(_S\) makes the read of \( g_1 \) conditional on this return value: the read uses the value in \( \text{ckpt}_g_1 \) during re-execution and uses the up-to-date value in \( g_1 \) during regular execution.

The above BugTM\(_S\) transformation can successfully recover from the failure on Line 11 in Figure 3.8(c), because the checkpointed-reexecution essentially guarantees the RAW atomicity between Line 1 and Line 6. This transformation also guarantees to preserve the original program semantics during re-execution: its re-execution is equivalent with what the original program would behave if the re-executed region was executed instantaneously right after the \text{setjmp}.

**Final Algorithm** When encounters a \( w_{\text{kill}} \) which the failure site \( f \) depends upon, BugTM\(_S\) checks whether there exists a read \( r \) that satisfies all of the following conditions: (1) \( r \) may read from the same memory location written by \( w_{\text{kill}} \); (2) \( f \) depends on \( r \); (3) \( r \) and \( w_{\text{kill}} \) are inside the same basic block. If such a read \( r \) is found, BugTM\(_S\) transforms the code region between \( w_{\text{kill}} \) and \( r \) by (1) recording the \text{setjmp} return value to a thread-local variable \( \text{sj\_ret} \); (2) taking checkpoints right before \text{setjmp} for all the global/heap variables read between \( w_{\text{kill}} \) and \( r \) including \( r \), no matter related to the failure or not, following their load order; (3) making these accesses conditionally read from either the checkpoint or the up-to-date memory location based on \( \text{sj\_ret} \).

Note that, we need to checkpoint multiple global/heap variables in their original load order, because some architectures do not allow compilers to re-order loads for memory-
consistency concerns (e.g., x86). For a similar reason, we only handle \( r \) and \( w_{\text{kill}} \) inside the same basic block, because otherwise there could be inconsistent load orders among different paths from \( w_{\text{kill}} \) to \( f \).

As an optimization, when there are multiple memory reads that BugTM\( S \) needs to checkpoint, BugTM\( S \) simply creates a clone of the region from \( w_{\text{kill}} \) to the end of its basic block, makes every cloned global/heap read gets its value from the checkpoint, and switches between the cloned and the original version based on \( sj_{\text{ret}} \).

When integrating with the original rollback scheme of ConAir, BugTM\( S \) configures the re-execution to use the checkpoints, if they exist, in the first re-execution attempt, and switch to not using checkpoints for following attempts. Since the re-execution using checkpoints is deterministic, there is no point for more attempts if the first attempt fails.

**Limitations** This extension does not allow BugTM\( S \) to recover from write-after-read atomicity violations; and may not fundamentally recover from a read-after-write failure. Take the bug in Figure 1.3 as an example, by using the check-pointed value of \( s->\text{table} \) at Line 4, BugTM\( S \) will recover from the original failure on Line 5. However, after the re-execution ends at Line 4, the execution will continue using the update-to-date value of \( s->\text{table} \), which is \texttt{NULL}. Software probably will still fail, just at a later point. To fundamentally recover from this failure, we will need BugTM\( H \).

### 3.7 BugTM\( HS \)

As discussed in chapter 3.5, another interesting design point that can leverage both the performance strength of \texttt{setjmp}/\texttt{longjmp} and the recovery strength of HTM is to use them both. The high level idea is that we can apply ConAir to insert \texttt{setjmp} and \texttt{longjmp} recovery code into a program first; and then, only at places where the growth of re-execution regions are stopped by killing writes, we apply BugTM\( H \) to extend re-execution regions through HTM-based recovery.
Where to `setjmp` and `StartTx` ConAir and BugTM\(_H\) insert `setjmp` and `StartTx` using similar algorithms, easing the design of BugTM\(_{HS}\). That is, for every failure instruction \(f\) inside a function \(F\), ConAir (BugTM\(_H\)) traverses backward through every path \(p\) that connects \(f\) with the entrance of \(F\) on CFG, and puts a `setjmp` wrapper function (`StartTx` wrapper function) right after the first appearance of a *killing instruction*. We will refer to this location as \(loc_{setjmp}\) and \(loc_{StartTx}\), respectively. For ConAir, the killing instructions include the entrance of \(F\), writes to any global or heap variables, and a selected set of system/library calls; for BugTM\(_H\), the killing instructions include the entrance of \(F\), the loop-exit instruction, and all system/library calls \(^5\).

BugTM\(_{HS}\) slightly modifies the above algorithm. Along every path \(p\), BugTM\(_{HS}\) inserts the `setjmp` wrapper function at every \(loc_{setjmp}\), where ConAir would insert it. In addition, BugTM\(_{HS}\) inserts the `StartTx` wrapper function at \(loc_{StartTx}\), when \(loc_{StartTx}\) is farther away from \(f\) than \(loc_{setjmp}\) (i.e., offering longer re-execution). Note that BugTM\(_{HS}\) inserts `setjmp` at every location \(loc_{setjmp}\) where ConAir would have inserted `setjmp` because every \(loc_{setjmp}\) might be executed without an active hardware transaction due to unexpected HTM aborts and others. When \(loc_{setjmp}\) is same as \(loc_{StartTx}\), BugTM\(_{HS}\) would only insert `setjmp` instead of inserting `StartTx` wrapper function.

Where to `CommitTx` BugTM\(_{HS}\) inserts `CommitTx` wrapper functions exactly where BugTM\(_H\) inserts them. Note that, BugTM\(_{HS}\) inserts fewer `StartTx` than BugTM\(_H\), and hence starts fewer Txs at run time. Fortunately, this does not affect the correctness of how BugTM\(_{HS}\) inserts `CommitTx`, because the wrapper function makes sure that `CommitTx` executes only under an active Tx.

How to retry ConAir and BugTM\(_H\) insert `longjmp` and `AbortTx` wrapper functions, which are responsible for triggering rollback-based failure recovery, using the same algorithm —

---

\(^5\) BugTM\(_{HS}\) also combines the inter-procedural recovery of ConAir and BugTM\(_H\) in a similar way. We skip details for space constraints.
Figure 3.9: BugTM<sub>HS</sub> rollback wrapper function

right before a failure is going to happen as described in chapter 3.3 and chapter 3.4.2.

BugTM<sub>HS</sub> inserts its rollback function (Figure 3.9) at the same locations. We design BugTM<sub>HS</sub> rollback wrapper to first invoke HTM-rollback (i.e., <code>AbortTx</code>) if it is under an active transaction, which will allow a longer re-execution region and hence a higher recovery probability. The BugTM<sub>HS</sub> rollback wrapper invokes <code>longjmp</code> rollback if it is not under an active transaction. To make sure that the program would not keep attempting hopeless recoveries, BugTM<sub>HS</sub> continues to use the HTM-abort statistics in the <code>StartTx</code> wrapper function shown in Figure 3.5 and continues to keep the <code>longjmp</code> retry count threshold shown in Figure 3.9.

For examples shown in Figure 1.3, 3.1, and 3.3, BugTM<sub>HS</sub> would insert both <code>setjmp</code> and <code>StartTx</code> into the buggy code regions, because <code>StartTx</code> would provide longer re-execution regions in all three cases. However, if the <code>*buf++ = ' '</code> statement does not exist in Figure 3.1, BugTM<sub>HS</sub> would not insert <code>StartTx</code> there. Consequently, if failures happen, <code>longjmp</code> will be used for recovery.

Overall, we expect BugTM<sub>HS</sub> to improve the performance of BugTM<sub>H</sub> and improve the recovery capability of both BugTM<sub>H</sub> and ConAir. This will be confirmed through experiments in chapter 3.10.

### 3.8 Failure Diagnosis

Previous recovery techniques, like ConAir [82], leave failure diagnosis completely to the developers, which is often very time consuming. BugTM supports failure diagnosis through
the root-cause inference routine shown in Figure 3.10 and extra logging during recovery.

((a)) diagnosis for BugTM\textsubscript{H}

1 Input: information from a successful recovery
2 if (timeout failures)
3 output: deadlock
4 else if (other explicit failures)
5 output: Order Violation or WAW atomicity violation
6 else if (implicit failures) //HTM data conflict aborts
7 output: maybe RAR, RAW, or RAW atomicity violations

((b)) diagnosis for BugTM\textsubscript{S}

1 Input: information from a successful recovery
2 if (timeout failures)
3 output: deadlock
4 else if (first re-execution succeeds)
5 if (with checkpoint)
6 output: RAW atomicity violation
7 else
8 output: RAR atomicity violation
9 else if (re-execution succeeds after multiple attempts)
10 output: Order Violation or WAW atomicity violation

Figure 3.10: Recovery-guided root-cause diagnosis for BugTM\textsubscript{H} and BugTM\textsubscript{S}

The root-cause inference algorithms for BugTM\textsubscript{H} and BugTM\textsubscript{S} are shown in Figure 3.10a and Figure 3.10b, respectively. These two algorithms both use the time-out failure symptom to infer deadlock root cause as shown in Line 2 of both algorithms. However, the other parts of the algorithms differ from each other.

For BugTM\textsubscript{H}, the manifestation of a concurrency bug could lead to a HTM abort due to either (1) an explicit failure inside a HTM transaction (e.g., an assertion gets violated) or (2) a data conflict abort caused by the buggy data race inside a HTM transaction. As illustrated in Table 3.1, order violations and WAW atomicity violations are much more likely to lead to explicit failure aborts (Line 4–5 in Figure 3.10a), while RAR, RAW, and WAR atomicity violations are much more likely to lead to data conflict aborts (Line 6–7 in Figure 3.10a).

It is difficult for BugTM\textsubscript{H} to provide more detailed root-cause inference. Furthermore, in case of a data conflict abort, BugTM\textsubscript{H} actually cannot conclude for sure whether the abort is caused by a benign data race or a failure-inducing data race. BugTM\textsubscript{H} can only suggest that, if the abort was caused by a buggy race, the buggy race is much more likely to be RAR, RAW, or WAR atomicity violations than order violations or WAW atomicity violations (Line 6–7 in Figure 3.10a). As we will see, this is the major diagnosis-capability limitation of BugTM\textsubscript{H}, particularly in comparison with BugTM\textsubscript{S}. 43
For BugTM$_S$, we can make root-cause triage based on the number of re-executions that were needed for a successful recovery.

When the recovery requires only one re-execution, the root cause is either RAW atomicity violation or RAR atomicity violation (Line 4–8 in Figure 3.10b). In both cases, unserializable interleaving is unlikely to occur again during the re-execution of the expected-to-be-atomic code region, and hence one re-execution should work. We can further differentiate these two types of root causes, as the recovery of RAW atomicity violation requires the use of undo-log, as discussed in chapter 3.6.2 (Line 5–6 in Figure 3.10b).

When the recovery requires multiple re-execution attempts, the root cause is either an order violation or a WAW atomicity violation (Line 9–10 in Figure 3.10b). In case of an order violation, the failure thread, which executes unexpectedly fast, is waiting for the unexpectedly slow thread to catch up, which is likely to take more than one re-execution. In case of a WAW atomicity violation (e.g., code region $w_1 - w_2$ unserializably interleaved by a read $r$), the failure thread rolls back and re-executes $r$, hoping that the re-executed $r$ would occur after $w_2$, instead of in between $w_1$ and $w_2$. However, when the other thread would execute $w_2$ is unpredictable. Consequently, it might take multiple re-executions of $r$ for the recovery to succeed.

Note that, BugTM$_S$ cannot recover from failures caused by WAR atomicity violations, and hence WAR atomicity violations are not part of the algorithm in Figure 3.10b.

For BugTM$_{HS}$, how much diagnostic information can be provided depends on whether a failure is recovered through HTM retries or setjmp/longjmp. When the recovery is through HTM retries, the root-cause inference routine in Figure 3.10a applies; on the other hand, the inference routine in Figure 3.10b applies.

Comparison and discussion As we can see, BugTM$_H$ offers less diagnostic information than BugTM$_S$, mainly because there are a wide variety of reasons behind its transaction aborts (Line 7 in Figure 3.10a).
All schemes of BugTM further support failure diagnosis by logging memory access type (read/write), addresses, values, and synchronization operations during re-execution, which helps failure diagnosis with no run-time overhead and only slight recovery delay.

Some real-world concurrency bugs are complicated and may go beyond the categories we discussed above (e.g., multi-variable atomicity violations). However, complicated bugs can often be decomposed into simpler ones. Furthermore, some principles still hold. For example, if the re-execution succeeds with just one attempt, it is highly likely that an atomicity violation happened to a code region inside the re-execution region.

3.9 Methodology

BugTM is implemented using LLVM infrastructure (v3.6.1). We obtained the source code of ConAir, also built upon LLVM. All the experiments are conducted on 4-core Intel Core i7-5775C (Broadwell) machines with 6MB cache, 8GB memory running Linux version 2.6.32, and O3 optimization level.

Benchmark suite We have evaluated BugTM on 29 bugs, including all the real-world bug benchmarks in a set of previous papers on concurrency-bug detection, fixing, and avoidance [27, 29, 67, 82, 83, 84]. They cover all common types of concurrency-bug root causes and failure symptoms. They are from server applications (e.g., MySQL database server, Apache HTTPD web server), client applications (e.g., Transmission BitTorrent client), network applications (e.g., HawkNL network library, HTTrack web crawler, Click router), and many desktop applications (e.g., PBZIP2 file compressor, Mozilla JavaScript Engine and XPCOM). The sizes of these applications range 50K — 1 million lines of code. Finally, our benchmark suite contains 3 extracted benchmarks: Moz52111, Moz209188, and Bank.

The goal of BugTM is to recover from production-run failures, not to detect bugs. Therefore, our evaluation uses previously known concurrency bugs that we know how to trigger failures. In all our experiments, the evaluated recovery tools do not rely on any knowledge
about specific bugs in their failure recovery attempts.

**Setups and metrics** We will measure the recovery capability and overhead of three BugTM designs. We will also evaluate and compare with ConAir [82], the state of the art concurrency-bug recovery technique.

To measure recovery capability, we follow the methodology of previous work [28, 82], and insert *sleeps* into software, so that the corresponding bugs will manifest frequently. We then run each bug-triggering workload with each tool applied for 1000 times.

To measure the run-time overhead. We run the original software **without** any *sleeps* with each tool applied. We report the average overhead measured during 100 failure-free runs, reflecting the performance during **regular** execution. We also evaluate alternative designs of BugTM, such as not conducting inter-procedural recovery, not excluding system calls from Txs, not excluding loops, etc. Due to space constraints, we only show this set of evaluation results on Mozilla and MySQL benchmarks, two widely used client and server applications.

### 3.10 Experimental results

Overall, three BugTM schemes all have better recovery capability than ConAir, with BugTM$^\text{HS}$ being the best and BugTM$^H$ a close second. The three schemes also all have good performance, with BugTM$^S$ being the best and BugTM$^\text{HS}$ the second best. BugTM$^\text{HS}$ provides the best combination of recovery capability and performance.

#### 3.10.1 Failure recovery capability

Among all the 29 benchmarks, 9 cannot be recovered by any of the evaluated techniques, no matter ConAir or BugTM, and the remaining 20 can be recovered by at least one of the techniques (BugTM$^\text{HS}$ can recover all of these 20).
Table 3.2
Recovery capability comparison

<table>
<thead>
<tr>
<th>RootCause</th>
<th>ConAir</th>
<th>BugTM_S</th>
<th>BugTM_H</th>
<th>BugTM_HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL2011</td>
<td>AV_RAR</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MySQL38883</td>
<td>AV_RAR</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Apache21287</td>
<td>AV_RAW</td>
<td>–</td>
<td>✓*</td>
<td>✓</td>
</tr>
<tr>
<td>Moz-JS18025</td>
<td>AV_RAW</td>
<td>–</td>
<td>✓*</td>
<td>✓</td>
</tr>
<tr>
<td>Moz-JS142651</td>
<td>AV_RAW</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Bank</td>
<td>AV_WAR</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Transmission</td>
<td>OV</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

Total 1 3+ 6 7

*: failures partly recovered; Moz-JS: Mozilla JavaScript Engine.

Table 3.2 shows the result of 7 benchmarks where different tools show different recovery capability. For the other 13 benchmarks, ConAir and BugTM can all help recover from all of them.

ConAir fails to recover from 6 out of 7 failures in Table 3.2, mainly because it does not allow shared-variable writes in re-execution regions. As a result, it cannot recover from any RAW or WAR atomicity bugs, and some RAR bugs, including the one in Figure 3.1.

BugTM_S has better recovery capability than ConAir. Its deferred write technique helps it to successfully recover from the two RAR violation failures in the table. The undo log technique of BugTM_S allows it to partly recover from two out of three RAW benchmarks. BugTM_S does not apply undo log to Moz-JS142651 because the bug involves complicated control flows. Moz-JS18025 is demonstrated in Figure 1.3. As discussed earlier, BugTM_S can help recover from the failure shown in the figure, but cannot prevent subsequent failures caused by the NULL value of s->table. Apache21287 can be recovered by BugTM_S with about 50% probability, depending on which bug-related thread fails first. Finally, BugTM_S fundamentally cannot handle WAR violations, as discussed at the end of chapter 3.6.

BugTM_H can successfully recover from all the 6 failures that ConAir cannot in Table 3.2. BugTM_H cannot recover from the Transmission bug, because recovering this bug requires re-executing malloc, a trapping operation for Intel TSX but handled by ConAir. In fact,
malloc is allowed in some more sophisticated TM designs [61, 72].

BugTM_{HS} combines the strengths of BugTM_H and ConAir, and hence can successfully recover from all 7 benchmarks in Table 3.2. It recovers the first 6 failures through HTM retries. It recovers from the Transmission failure through longjmp (it rolls back the malloc that cannot be handled by HTM-retry through free).

Unrecoverable benchmarks There are 9 benchmarks that no tools can help recover for mainly three reasons. Some of these issues go beyond the scope of failure recovery, yet others are promising to address in the future. First, two order violation benchmarks cause failures when the failure thread is unexpectedly slow. Therefore, re-executing the failure thread would not help correct the timing. Fortunately, both failures can be prevented by delaying resource deallocation, a prevention approach proposed before for memory-bug failures [52, 60]. Second, three benchmarks, Cherokee326, Apache25520, and MySQL169, cause failures that are difficult to detect (i.e., silent data corruption). Tackling them goes beyond the scope of failure recovery. Third, the remaining four failures cannot be recovered due to un-reexecutable instructions, which are promising to address. For example, Intel TSX does not support putting memcpy, cond_wait, or I/O into its Txs. More sophisticated TMs with OS support [61, 72] could help recover these failures.

Hardened failure sites BugTM can help recover from failures caused by concurrency bugs because it statically identifies potential failure sites and inserts rollback-reexecution code surrounding them.

Table 3.3 shows the number of failure sites that are hardened by BugTM_{HS} in each benchmark software. Naturally, BugTM_{HS} identifies and hardens the fewest failure sites in the smallest programs (Bank and HawkNL) and the most failure sites in the largest programs (MySQL and Apache). Benchmarks from the same software have slightly different numbers of failures sites in Table 3.3, because they come from different versions of the software. We do full instrumentation for each version.
## Table 3.3
Static failure sites hardened by BugTM$_{HS}$

<table>
<thead>
<tr>
<th></th>
<th>Assertion Violation</th>
<th>Wrong Output</th>
<th>Seg. Fault</th>
<th>Deadlock</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL2011</td>
<td>151</td>
<td>3128</td>
<td>15498</td>
<td>21</td>
<td>18798</td>
</tr>
<tr>
<td>MySQL3596</td>
<td>170</td>
<td>3297</td>
<td>15791</td>
<td>19</td>
<td>19277</td>
</tr>
<tr>
<td>MySQL38883</td>
<td>170</td>
<td>3276</td>
<td>15820</td>
<td>19</td>
<td>19285</td>
</tr>
<tr>
<td>Apache21287</td>
<td>5</td>
<td>503</td>
<td>16834</td>
<td>89</td>
<td>17431</td>
</tr>
<tr>
<td>Moz-JS18025</td>
<td>35</td>
<td>34</td>
<td>1802</td>
<td>10</td>
<td>1881</td>
</tr>
<tr>
<td>Moz-JS142651</td>
<td>1</td>
<td>31</td>
<td>1812</td>
<td>13</td>
<td>1857</td>
</tr>
<tr>
<td>Bank</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Moz-ex52111</td>
<td>1</td>
<td>1</td>
<td>23</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Moz-ex209188</td>
<td>1</td>
<td>2</td>
<td>34</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>MySQL791</td>
<td>129</td>
<td>2421</td>
<td>17890</td>
<td>43</td>
<td>20483</td>
</tr>
<tr>
<td>MySQL16582</td>
<td>134</td>
<td>2943</td>
<td>14592</td>
<td>20</td>
<td>17689</td>
</tr>
<tr>
<td>Click</td>
<td>2134</td>
<td>32</td>
<td>2234</td>
<td>0</td>
<td>4400</td>
</tr>
<tr>
<td>FFT</td>
<td>5</td>
<td>32</td>
<td>14</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>HTTrack</td>
<td>657</td>
<td>504</td>
<td>3146</td>
<td>0</td>
<td>4307</td>
</tr>
<tr>
<td>Moz-xpcom</td>
<td>1</td>
<td>117</td>
<td>6791</td>
<td>0</td>
<td>6909</td>
</tr>
<tr>
<td>Transmission</td>
<td>430</td>
<td>190</td>
<td>2151</td>
<td>0</td>
<td>2771</td>
</tr>
<tr>
<td>zsnes</td>
<td>1</td>
<td>50</td>
<td>331</td>
<td>0</td>
<td>382</td>
</tr>
<tr>
<td>HawkNL</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Moz-JS79054</td>
<td>0</td>
<td>5</td>
<td>134</td>
<td>6</td>
<td>145</td>
</tr>
<tr>
<td>SQLite1672</td>
<td>0</td>
<td>25</td>
<td>47</td>
<td>1</td>
<td>73</td>
</tr>
</tbody>
</table>

There are four types of potential failure sites: every assertion checking is a potential site for assertion-violation failure; every output function call is a potential site for wrong-output failure; every dereference of a heap or global pointer is a potential site for segmentation-fault failures; every lock acquisition that is enclosed by another lock acquisition is considered as a potential deadlock site. Among these four, potential segmentation-fault sites are the most common, and potential deadlock sites are the least common.

BugTM$_H$ and BugTM$_S$ have hardened similar, just slightly fewer, failure sites as BugTM$_{HS}$, as some failure sites hardened by BugTM$_{HS}$ are identified as un-recoverable for BugTM$_H$ and BugTM$_S$, and hence not hardened by them.
### Table 3.4
Overhead during regular execution and detailed performance comparison

<table>
<thead>
<tr>
<th></th>
<th>Run-time Overhead</th>
<th>#setjmp</th>
<th>#StartTx</th>
<th>#StartTx per 10µs</th>
<th>Abort%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>HS</td>
<td>S</td>
</tr>
<tr>
<td>MySQL2011</td>
<td>0.05%</td>
<td>0.04%</td>
<td>0.13%</td>
<td>0.08%</td>
<td>642974</td>
</tr>
<tr>
<td>MySQL3596</td>
<td>0.40%</td>
<td>0.43%</td>
<td>3.10%</td>
<td>1.12%</td>
<td>144119</td>
</tr>
<tr>
<td>MySQL38888</td>
<td>0.40%</td>
<td>0.41%</td>
<td>3.08%</td>
<td>1.11%</td>
<td>144109</td>
</tr>
<tr>
<td>Apache21287</td>
<td>0.55%</td>
<td>0.73%</td>
<td>3.77%</td>
<td>3.00%</td>
<td>39918</td>
</tr>
<tr>
<td>Moz-JS18025</td>
<td>0.57%</td>
<td>0.86%</td>
<td>9.03%</td>
<td>2.62%</td>
<td>3987</td>
</tr>
<tr>
<td>Moz-JS142651</td>
<td>0.76%</td>
<td>0.86%</td>
<td>11.9%</td>
<td>5.30%</td>
<td>2269</td>
</tr>
<tr>
<td>Bank</td>
<td>0.15%</td>
<td>0.23%</td>
<td>2.18%</td>
<td>2.95%</td>
<td>6</td>
</tr>
<tr>
<td>Moz-ex52111</td>
<td>0.47%</td>
<td>0.65%</td>
<td>0.53%</td>
<td>0.41%</td>
<td>4</td>
</tr>
<tr>
<td>Moz-ex209188</td>
<td>0.12%</td>
<td>0.12%</td>
<td>0.58%</td>
<td>0.77%</td>
<td>2</td>
</tr>
<tr>
<td>MySQL791</td>
<td>0.35%</td>
<td>0.84%</td>
<td>1.98%</td>
<td>0.24%</td>
<td>48933</td>
</tr>
<tr>
<td>MySQL16582</td>
<td>0.15%</td>
<td>0.33%</td>
<td>3.03%</td>
<td>0.99%</td>
<td>269230</td>
</tr>
<tr>
<td>Click</td>
<td>0.57%</td>
<td>0.80%</td>
<td>8.11%</td>
<td>3.60%</td>
<td>4893</td>
</tr>
<tr>
<td>FFT</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.03%</td>
<td>0.14%</td>
<td>23</td>
</tr>
<tr>
<td>HTTrack</td>
<td>0.15%</td>
<td>0.16%</td>
<td>0.64%</td>
<td>0.04%</td>
<td>9004</td>
</tr>
<tr>
<td>Moz-xpcom</td>
<td>0.38%</td>
<td>0.40%</td>
<td>0.45%</td>
<td>0.03%</td>
<td>313</td>
</tr>
<tr>
<td>Transmission</td>
<td>0.11%</td>
<td>0.20%</td>
<td>0.22%</td>
<td>0.07%</td>
<td>1088</td>
</tr>
<tr>
<td>zsses</td>
<td>0.05%</td>
<td>0.11%</td>
<td>0.03%</td>
<td>0.44%</td>
<td>10684</td>
</tr>
<tr>
<td>HawkNL</td>
<td>0.09%</td>
<td>0.08%</td>
<td>0.00%</td>
<td>0.15%</td>
<td>10</td>
</tr>
<tr>
<td>Moz-JS7954</td>
<td>0.84%</td>
<td>0.99%</td>
<td>11.7%</td>
<td>4.20%</td>
<td>340</td>
</tr>
<tr>
<td>SQLite1672</td>
<td>0.05%</td>
<td>0.01%</td>
<td>0.98%</td>
<td>0.50%</td>
<td>6</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.31%</td>
<td>0.42%</td>
<td>3.08%</td>
<td>1.39%</td>
<td>-</td>
</tr>
</tbody>
</table>

C: ConAir; S: BugTM_S; H: BugTM_H; HS: BugTM_HS; red font denotes >3% overhead; #: count of dynamic instances; Abort%: percentage of aborted dynamic Txs
3.10.2 Performance

Table 3.4 shows the regular-run overheads of applying BugTM schemes to 20 benchmarks, all the benchmarks that are recoverable by BugTM$_S$.

Overall, BugTM$_S$ has the best performance, incurring less than 1% overhead for all benchmarks at run time, almost a free lunch for production failure recovery. Considering that each reexecution point only takes a few nanoseconds to execute (a `setjmp`), the low overhead of BugTM$_S$ is understandable.

BugTM$_H$ incurs more overhead, about 3% on average, than ConAir does, about 0.3% on average, mainly because a Tx is much more expensive than a `setjmp`.

Fortunately, BugTM$_{HS}$ wins most of the lost performance back, incurring 1.4% overhead on average and less than 3% for all but 3 benchmarks. In the worst cases, it incurs 4.2% and 5.3% overhead for two benchmarks in Mozilla JavaScript Engine (JSE), a browser component with little I/O. If we apply BugTM$_{HS}$ to the whole browser, the overhead would be much smaller, as JSE never takes >20% of the whole page-loading time based on our profiling and previous work [54].

Comparing BugTM$_{HS}$ with BugTM$_H$, BugTM$_{HS}$ is faster mainly because it has greatly reduced the number of hardware transactions at run time. For example, for the four benchmarks that incur the largest overhead under BugTM$_H$ (Moz-JS18025, Moz-JS142651, Click, and Moz-JS79054), BugTM$_{HS}$ reduces the \#startTx per 10$\mu$s from 9.4 — 30.4 to 2.6 — 12.6, and hence dropping the overhead from 8.11—11.9% to 2.6—5.3%.

Tx abort rate is less than 1% for all benchmarks, with more than 95% of all aborts being unknown aborts (timer interrupts, etc.). As chapter 3.10.4 will show, abort rates and overhead are much worse in alternative designs.

We do not show the execution frequencies of failure sites. In our performance experiments behind Table 3.4, failures are never triggered. Of course, code regions surrounding failure sites have been executed during our experiments. Their execution frequencies are reflected
Table 3.5
Failure recovery details by BugTM$_{HS}$

<table>
<thead>
<tr>
<th>Root Cause</th>
<th>Recovery ($\mu$-seconds)</th>
<th># Retries</th>
<th>Restart ($\mu$-seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL2011 AV$_{RAR}$</td>
<td>10</td>
<td>1</td>
<td>5069944</td>
</tr>
<tr>
<td>MySQL3596 AV$_{RAR}$</td>
<td>21</td>
<td>1</td>
<td>5982713</td>
</tr>
<tr>
<td>MySQL38883 AV$_{RAR}$</td>
<td>21</td>
<td>1</td>
<td>5982720</td>
</tr>
<tr>
<td>Apache21287 AV$_{RAW}$</td>
<td>98</td>
<td>1</td>
<td>4824363</td>
</tr>
<tr>
<td>Moz-JS18025 AV$_{RAW}$</td>
<td>42</td>
<td>1</td>
<td>5508</td>
</tr>
<tr>
<td>Moz-JS142651 AV$_{RAW}$</td>
<td>74</td>
<td>1</td>
<td>1143</td>
</tr>
<tr>
<td>Bank AV$_{WAR}$</td>
<td>5</td>
<td>1</td>
<td>139</td>
</tr>
<tr>
<td>Moz-ex52111 AV$_{RAW}$</td>
<td>183</td>
<td>1245</td>
<td>149</td>
</tr>
<tr>
<td>Moz-ex209188 AV$_{WAR}$</td>
<td>92</td>
<td>741</td>
<td>107</td>
</tr>
<tr>
<td>MySQL791 AV$_{WAW}$</td>
<td>10732</td>
<td>11234</td>
<td>66724</td>
</tr>
<tr>
<td>MySQL16582 AV$_{WAW}$</td>
<td>24321</td>
<td>23108</td>
<td>7424224</td>
</tr>
<tr>
<td>Click OV</td>
<td>1211</td>
<td>7662</td>
<td>900452</td>
</tr>
<tr>
<td>FFT OV</td>
<td>11719</td>
<td>9725</td>
<td>131494</td>
</tr>
<tr>
<td>HTTrack OV</td>
<td>4625</td>
<td>18244</td>
<td>11776</td>
</tr>
<tr>
<td>Moz-xpcom OV</td>
<td>37388</td>
<td>124732</td>
<td>217041</td>
</tr>
<tr>
<td>Transmission OV</td>
<td>22445</td>
<td>1234</td>
<td>553298</td>
</tr>
<tr>
<td>znes Deadlock</td>
<td>109</td>
<td>1</td>
<td>354640</td>
</tr>
<tr>
<td>HawkNL Deadlock</td>
<td>78</td>
<td>1</td>
<td>14493</td>
</tr>
<tr>
<td>Moz-JS79054 Deadlock</td>
<td>35</td>
<td>1</td>
<td>472</td>
</tr>
<tr>
<td>SQLite1672 Deadlock</td>
<td>89</td>
<td>1</td>
<td>1570</td>
</tr>
</tbody>
</table>

The experiments are conducted with small amount of noises inserted to help trigger the concurrency-bug failures by the numbers of setjmp and StartTx in Table 3.4.

**Recovery time & Comparison with whole-program restart** As shown in Table 3.5, a successful BugTM failure recovery takes little time.

The recovery of atomicity violations, except for WAW atomicity violations, and deadlocks mostly takes less than 100 $\mu$-seconds. For these bugs, one run of retry is sufficient to avoid unexpected interleaving, because the failing thread does not need to wait for any other threads.

The recovery of order violations and WAW atomicity violations takes slightly longer time, as it highly depends on how much sleep is inserted to trigger the failure, as discussed in chapter 3.8.

The reexecution regions are always small, ranging from 5-224 instructions at byte-code
BugTM recovery is much faster than a system restart, which could take a few minutes or even more for complicated systems, as shown in the table. BugTM recovery also avoids wasting already conducted computation and crash inconsistencies. For example, without BugTM, MySQL791 would crash the database after a table is changed but before this change is logged, leaving inconsistent persistent states.

Table 3.5 only shows the recovery details of BugTM_{HS}. The details of BugTM_{H} and BugTM_{S} are similar — they took negligible amount of time to recover from most atomicity violations and deadlocks, and slightly longer time to recover from order violations and WAW atomicity violations.

**Understanding BugTM_{H} overhead** The overhead of BugTM_{H} differs among benchmarks, ranging from 0.00% to 11.9%. As TM researchers found before, performance in TM systems is often complicated [7, 58]. An indicating metrics for our benchmarks is the frequency of dynamic StartTx. As shown in the #StartTx per 10μs column of Table 3.4, BugTM_{H} executes more than 1 StartTx per 10 micro second on average for 10 benchmarks, and incurs more than 1% overhead for 9 of them.

### 3.10.3 Diagnosis

Table 3.6 shows the different diagnosis capability of BugTM.

BugTM_{S} provides useful diagnosis information for all of the 18 benchmarks that it can help recover from. For 10 out of these benchmarks whose root causes are order violations or WAW atomicity violations, BugTM_{S} reports that the root cause could be either one of these two. For the other 8 benchmarks, BugTM_{S} accurately pin-points the exact root cause.

There are many benchmarks in Table 3.6 that can be recovered by BugTM_{H} and BugTM_{HS}, but cannot be diagnosed by them. The reason is that these bugs’ manifestation leads to HTM data conflict aborts, while BugTM_{H} and BugTM_{HS} cannot decide whether such aborts are
Table 3.6
Diagnosis capability comparison

<table>
<thead>
<tr>
<th>RootCause BugTM_S</th>
<th>BugTM_H</th>
<th>BugTM_HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL2011 AVRAR</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>MySQL3596 AVRAR</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>MySQL38883 AVRAR</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Apache21287 AVRAW</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Moz-JS18025 AVRAW</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Moz-JS142651 AVRAW</td>
<td>NA</td>
<td>-</td>
</tr>
<tr>
<td>Bank AVWAR</td>
<td>NA</td>
<td>-</td>
</tr>
<tr>
<td>Moz-ex52111 AVWAW</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>Moz-ex209188 AVWAW</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>MySQL791 AVWAW</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>MySQL16582 AVWAW</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>Click OV</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>FFT OV</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>HTTrack OV</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>Moz-xpcom OV</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>Transmission OV</td>
<td>✓*</td>
<td>NA</td>
</tr>
<tr>
<td>zsnes OV</td>
<td>✓*</td>
<td>✓*</td>
</tr>
<tr>
<td>HawkNL Deadlock</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Moz-JS79054 Deadlock</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SQLite1672 Deadlock</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>8(18)</td>
<td>3(12)</td>
</tr>
</tbody>
</table>

NA: can’t help recover from them
*: report two possible root causes, order violation and WAW violation.
-: no affirmative diagnosis
Table 3.7
BugTM<sub>H</sub> vs. alternative designs

<table>
<thead>
<tr>
<th></th>
<th>BugTM&lt;sub&gt;H&lt;/sub&gt;</th>
<th>Intra-proc</th>
<th>Trapping-Ins</th>
<th>Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moz-xpcom</td>
<td>0.45% ✓</td>
<td>0.44% ×</td>
<td>0.54% ✓</td>
<td>0.20% ✓</td>
</tr>
<tr>
<td>Moz-JS18052</td>
<td>9.03% ✓</td>
<td>7.01% ✓</td>
<td>16.8% ✓</td>
<td>11.3% ✓</td>
</tr>
<tr>
<td>Moz-JS79054</td>
<td>11.7% ✓</td>
<td>11.4% ×</td>
<td>14.0% ✓</td>
<td>11.1% ✓</td>
</tr>
<tr>
<td>Moz-JS142651</td>
<td>11.9% ✓</td>
<td>7.6% ×</td>
<td>19.6% ✓</td>
<td>12.2% ✓</td>
</tr>
<tr>
<td>MySQL791</td>
<td>1.98% ✓</td>
<td>1.50% ✓</td>
<td>11.4% ✓</td>
<td>11.5% ✓</td>
</tr>
<tr>
<td>MySQL2011</td>
<td>0.13% ✓</td>
<td>0.13% ×</td>
<td>1.50% ✓</td>
<td>0.06% ✓</td>
</tr>
<tr>
<td>MySQL3596</td>
<td>3.10% ✓</td>
<td>3.05% ✓</td>
<td>108% ×</td>
<td>2.63% ✓</td>
</tr>
<tr>
<td>MySQL16582</td>
<td>3.03% ✓</td>
<td>0.16% ✓</td>
<td>93.1% ✓</td>
<td>1.89% ✓</td>
</tr>
<tr>
<td>MySQL38883</td>
<td>3.08% ✓</td>
<td>3.04% ✓</td>
<td>106% ×</td>
<td>2.52% ✓</td>
</tr>
</tbody>
</table>

%: the overhead over baseline execution w/o recovery scheme applied; ✓: failure recovered; ×: failure not recovered.

caused by software bugs or benign races.

BugTM also conducts memory-access logging during failure recovery attempts. Evaluation shows that this extra logging incurs 1.01X – 2.5X slowdowns to failure recovery with no overhead to regular execution. The 2.5X slowdown happens during a fast half-microsecond recovery.

3.10.4 Alternative designs of BugTM

Table 3.7 shows the performance and recovery capability of three alternative designs of BugTM<sub>H</sub>. Due to space constraints, we only show results on benchmarks in MySQL database server and Mozilla browser suite (non-extracted). Since BugTM<sub>H</sub> is the foundation of BugTM<sub>HS</sub>, an alternative design that degrades the performance or recovery capability of BugTM<sub>H</sub> will also degrade BugTM<sub>HS</sub> accordingly as discussed below.

Inter-procedural vs. Intra-procedural BugTM<sub>H</sub> uses the inter-procedural algorithm discussed in chapter 3.4.5. This design adds 0.00 – 4.3 % overhead to its intra-procedural alternative, as shown in Table 3.7. In exchange, there are 4 benchmarks in Table 3.7 that require inter-procedural re-execution of BugTM<sub>H</sub> to recover from. Among them, two can be recovered by ConAir and hence can still be recovered by intra-procedural BugTM<sub>HS</sub>;
the other two require inter-procedural BugTM\textsubscript{HS} to recover. Recovering MySQL2011, Moz-xpcom, Moz-JS79054 has to re-execute not only function \( F \) where failures occur, but also \( F \)'s caller. As for Moz-JS142651, we need to re-execute a callee of \( F \) where a memory access involved in the atomicity violation resides.

**Including trapping instructions in Txs** Clearly, if BugTM\textsubscript{H} did not intentionally exclude system calls from its Txs, more Txs will abort. This alternative design hurts performance a lot, incurring around 100% overhead for three MySQL benchmarks shown in Table 3.7. Such design also causes BugTM\textsubscript{HS} to incur more than 20% overhead on these benchmarks. Furthermore, these aborts may hurt recovery capability, as they will cause corresponding Tx regions to execute in non-transaction mode to avoid endless aborts and hence lose the opportunity of failure recovery. This indeed happens for two benchmarks in Table 3.7. One of them will also fail to be recovered by BugTM\textsubscript{HS} under this alternative design.

**Including loops in Txs** could lead to more capacity aborts, which are indeed observed for all benchmarks in Table 3.7. The overhead actually does not change much for most benchmarks. Having said that, it raises the overhead of MySQL791 from 1.98% to 11.5%.

**More Txs** We also tried randomly inserting more \texttt{startTx}. The overhead increases significantly. For Moz-JS142651, when we double, treble, and quadruple the number of dynamic Txs through randomly inserted Txs, the overhead goes beyond 30%, 100%, and 800%. The impact to BugTM\textsubscript{HS} would also be huge accordingly.

### 3.11 Discussion

BugTM has explored three designs that all have better failure-recovery capability than the state of the art technique ConAir, at the cost of slightly worse performance than ConAir. These three designs have different trade-off combinations between performance and failure-recovery capability. Among them, BugTM\textsubscript{S} has the best performance, as well as the best
failure diagnosis capability, but has the worst recovery capability. BugTM\textsubscript{H} has better recovery capability than BugTM\textsubscript{S}, but the worst performance. BugTM\textsubscript{HS} has the best recovery capability, and also good performance that is much better than BugTM\textsubscript{H} but slightly worse than BugTM\textsubscript{S}. On commodity machines that support HTM, BugTM\textsubscript{HS} is likely the best BugTM design. For the multi-threaded software systems which are performance-sensitive or I/O intensive, like browser, BugTM\textsubscript{S} and BugTM\textsubscript{HS} are recommended instead of BugTM\textsubscript{H} considering BugTM’s higher overhead and HTM constraints. For other multi-threaded software systems which emphasize availability, e.g. banking/business systems, they may prefer BugTM\textsubscript{H} and BugTM\textsubscript{HS} for better recovery capabilities, which means fewer failure occurrences during production runs. Overall, all hardened programs of these three designs can perform as the intermediate patches before the final patches generated, which improves software availability and could facilitate the debugging and patching process.

As the evaluation and our earlier discussion show, BugTM does not guarantee to recover from all concurrency bug failures, particularly if the bug has a long error propagation before causing a failure. However, we believe BugTM, particularly BugTM\textsubscript{HS}, would provide a beneficial safety net to most multi-threaded software with little deployment cost or performance loss. Even if not successful, the recovery attempts only delays the failure manifestation by a negligible amount of time (less than 40 micro-seconds in our experiments).

Several practices can help further improve the benefit of BugTM. First, as discussed in chapter 3.10.1, some improvements of HTM design would greatly help BugTM to recover from more concurrency-bug failures. Second, developers’ practices of inserting sanity checks into software would greatly help BugTM. With more sanity checks, fewer concurrency bugs would have long error propagation and hence more concurrency-bug failures would be recovered by BugTM. Third, different from locks, which protect the atomicity of a code region only when the region and all its conflicting code are all protected by the same lock, BugTM can help protect a code region regardless how other code regions are written. Consequently,
developers could choose to selectively apply BugTM to parts of software where he/she is least certain about synchronization correctness.

Finally, BugTM can be applied to software that is already using HTMs. BugTM will choose not to make its HTM regions nesting with existing HTM regions.

### 3.12 Conclusion

Concurrency bugs severely affect system availability. We present three TM-inspired techniques to help automatically recover concurrency-bug failures during production runs. BugTM$_H$ automatically inserts HTM APIs into software. It is capable of recovering failures caused by all major types of concurrency bugs and incurs 3.08% overhead on average in our evaluation. BugTM$_S$ uses STM inspired techniques to enhance the recovery capability of state-of-the-art ConAir. Although it cannot recover as many failures as BugTM$_H$, it incurs less than 1% overhead and can provide useful diagnostic information. BugTM$_{HS}$ combines ConAir and BugTM$_H$, achieving a good combination of recovery capability and performance (1.39% overhead). Overall, BugTM improves the state of the art of failure recovery, and presents novel ways of using TM techniques.
Along the direction of efficient concurrency bug, we present BFix, an automatic tool to bypass parts of original computation under buggy context for concurrency bug fixing during code maintenance. The patches generated by BFix have better performance and code readability than most existing auto-fixing tools, since it skips some existing computations via simple conditional checking without using timing-consuming synchronization primitives (e.g. locks). In this chapter, we will introduce the details of BFix. Chapter 4.1 introduces the challenges BFix is facing. Chapter 4.2 discusses the background of auto-fixing tools, while chapter 4.3 shows concurrency bug fixing framework that BFix reuses. In chapter 4.5 talks the work flow of BFix in details, including feasibility checking, patch generation and patch merging. Chapter 4.6 introduces the methodology of our experiment. Chapter 4.7 shows the detailed experimental result. Chapter 4.8 discusses what bugs BFix fails to fix. Chapter 4.9 concludes this chapter.

4.1 Challenges for automatically generating correct bypassing patches for concurrency bugs

As mentioned, bypassing patch generation tools should decide which computation to bypass, what the condition is and prove the patch correctness. The latter two challenges are highly related to the solution to which computation to bypass.

Ideally, bypassing patches should try to skip the execution of some operations along the buggy interleaving for bug fixing. Because when they are skipped, the bug might disappear.

Skipping failure-triggering operation is a straightforward option, since it is the explicit reflection of bug. Taking write-after-write AV bug as an example, as shown in (i) of Figure
4.1, assuming it would throw assertion as failure to catch unexpected program state when Thread-2’s $R$ unexpectedly reads the intermediate result of Thread-1’s two consecutive writes $P$ and $C$. In the bug-triggering timing, if assertion invocation is skipped, as shown in (ii) of Figure 4.1, the failure symptom for this bug would disappear for sure. But this option has two main problems. Firstly, the bug is already propagated among the code region between $R$ and the failure location where the assertion is called. Secondly, since assertion here is used to catch unexpected program state, skipping it and continuing executing the subsequent code region would violate the original intention, which assumes the subsequent code only executes when program goes into the expected state.

Skipping the earlier operations is another option, like buggy operation $R$ is skipped, as shown in (iii) of Figure 4.1. It has several advantages over skipping failure-triggering operation. Firstly, unexpected execution order of buggy operations directly means buggy timing, it is the sufficient condition of failure symptom and could be used as the indicator of bug occurrence. Secondly, skipping them might help eliminate the bug and avoid bug propagation problem introduced by skipping failure-triggering operation. Thirdly, their information can be easily got from bug detectors.

However, only skipping buggy operation might lead to inconsistency problem given dependency among operations. Hence, the dependency relations among operations should be explored. We can get these by conducting heavy static analysis or asking the help from developers. But none of them can help propose a simple design.

As for the synthesized condition, how to choose the variables used in extra conditional expression is also not trivial. There are two options to choose variables. If leveraging existing variables, bypassing patch generation tools should pick the suitable variables among tons of used variables in the original program. Extra semantics information might be needed. These will make the design complex. If introducing new variables, how and where these variables are updated and checked is also not easy.
Figure 4.1: Why bypassing works for AV bugs: (i) bug context for write-after-write AV bug; (ii) skipping failure-triggering operations; (iii) skipping buggy operations

Correctness proof is also very challenging. Even for the developers who have rich background about the bugs they are working on, it is not easy to reason the correctness patch (e.g., since bypassing patch usually change the sequential semantics, only focusing on the thread where to-be-skipped operations are on can not help reason the patch correctness, developers usually need check more threads and code regions). It is even harder for the automatic tools, because they lack of rich semantics information.

### 4.2 Existing auto-fixing tools

Most auto-fixing tools [27, 28, 40, 42, 74] prevent buggy timing occurrence via adding synchronizations operations into software. They do not get involved with any computation semantics. The correctness is obviously guaranteed, since the correct computation semantics already exist in software and they only add constraints to enforce occurrence of the correct computation semantics.

Figure 4.2 is a simplified MySQL bug from CFix. If \texttt{NULL} assignment of \texttt{thd\rightarrow proc} in
Figure 4.2: MySQL example from CFix. Interleaving ‘A’ and ‘B’ are two bug-free interleavings, ‘+’ indicates the generated patch by CFix

Thread-2 happens between if checking and string concatenation in Thread-1, the concurrency bug would manifest. After adding locks by CFix, there are only two possible interleavings for these code snippets, marked as A and B in Figure 4.2. Actually, these two interleavings can originally happen due to different context switch even without adding lock/unlock synchronizations. That is the main correctness reasoning for synchronization-adding tools, like CFix.

There are other auto-fixing tools which do not belong to synchronization-adding category. Data privatization [25] creates a temporary variable to buffer the result of an earlier memory access in order to prevent the buggy shared memory access. HFix [39] tries to move around existing synchronizations to extend the critical regions or enforce order relations. But similarly, none of them get involved with computation semantics.

### 4.3 Bug-fixing framework

BFix reuses the general bug-fixing framework proposed by CFix. The inputs to the bug-fixing framework are bug reports, which can be automatically generated by bug detection tools. Given a bug report, some feasibility checks are conducted to determine whether the designed fixing strategy might be suitable for the bug or not. If yes, it goes through patch
generation. Finally, patch merging is conducted for better performance and readability.

BFix reuses an important philosophy of CFix – bug fixing works only when correct bug reports are provided. It reuses the bug report format requirement of CFix. An atomicity violation bug report needs to specify three operations $P$, $C$, and $R$, referred to as buggy operations. An AV bug manifests when one thread unexpectedly executes $R$ between another thread’s execution of $P$ and $C$. For AV bugs, based on the read/write access difference, they can be categorized into four types [48], as shown in Figure 4.4. An order violation bug report needs to specify two operations $A$ and $B$, referred to as buggy operations too. An OV bug happens when $A$ unexpectedly executes after, instead of before $B$. In the remaining parts of this dissertation, we will use the same symbols as shown in Figure 4.3 to mark the code regions for AV and OV bugs. Bug reports also provide (1) a call stack that executes the buggy operation in a thread, and (2) a chain of call stacks indicating how that thread has been created.

Figure 4.3: Notation for the buggy code snippet. Arrows indicate the bug-triggering timing
4.4 Three bypassing patterns

As discussed before, given a concurrency bug, it is easy to eliminate the original failure symptom by skipping one of the buggy operations (i.e., \( P \), \( C \), or \( R \) in an atomicity violation or \( A \) or \( B \) in an order violation), but it is challenging to prove that such skipping does not introduce new bugs.

In this chapter, we present three common patterns among buggy operations. As we will see, if a concurrency bug falls into one of these patterns, the correctness of a corresponding bypassing patch will be easy to prove.

4.4.1 Overwritten

The overwritten pattern involves two buggy operations that come from different threads and write to the same memory location. This pattern utilizes data flow information from the original program. If two buggy operations are two consecutive writes (\( W_1 \) and \( W_2 \)) to the same memory location, the side effect of the \( W_1 \) would be overwritten by \( W_2 \) during \( T' \). In this pattern, skipping operations \( O \) in \( P' \) is \( W_1 \). Since both \( P_{T'} \) and \( P'_{T'} \) skip \( W_1 \), it is easy
Figure 4.5: A simplified bug in Jscript module of Mozilla, `'+'` indicates the manual patch to prove they generate the same semantics.

Figure 4.5 shows a real-world concurrency bug from Mozilla, which belongs to the overwritten pattern. Thread-1 tries to insert an entry into a table `table`, while Thread-2 conducts job cleaning, it firstly fills `table` with zero, then checks each entry in the `table`. If the insertion of Thread-1 happens in between, an assertion failure would show up and the system would crash. The manual patch would skip the insertion in `PROPERTY_CACHE_FILL` when Thread-2 conducts cleaning. Specifically the manual patch introduces a context-associating flag, which would be set as FALSE before Thread-2’s cleaning and reset as TRUE afterward. Depending on the value of this flag, Thread-1 will either skip the insertion when bug timing happens or conduct the insertion under the correct timing.

Patch correctness reasoning utilizes the overwritten property without asking extra semantics from the developers. The patched program of Figure 4.5 in bug-triggering timing will skip write access to `table` in `PROPERTY_CACHE_FILL`. This execution behavior $P'_T$ is actually equivalent with the original program under a correct timing ($P_T$), where this write access is conducted before `memset` of `jsFlushPropertyCache`, as shown in Figure 4.6, since the side effect of this insertion will be overwritten by `memset`.

This pattern exist widely in AV and OV bugs. Look back four types of AV bugs in Figure 4.4. For `RAW`, considering a correct timing where $P$, $C$ and $R$ follow the order $RPC$, since Write$R$ and Write$P$ write to same memory location $x$, the side effect of Write$R$ is...
PROPERTY_CACHE_FILL:
if (js_context.enable)
    (table.s)[0] = obj;

js_FlushPropertyCache:
memset(table, 0, sizeof table);
assert(table.s[0] == 0);

// Thread-1 // Thread-2

(a) Execution timeline for the patched version when bug happens
(b) Execution timeline when PROPERTY_CACHE_FILL is conducted early

Figure 4.6: Execution timeline for the patched version in bug-triggering timing and a bug-free run in Figure 4.5

1 //Thread-1
2
3 Write_P;
4
5 Read_C;

Figure 4.7: Simplified overwritten pattern for read-after-write AV bugs and their patch pattern

would be overwritten by Write_P, other threads would only get the result written by Write_P. Hence, Write_R might could be skipped. For WAR, considering a correct timing where the order PCR is followed, similarly, Write_C would be overwritten by Write_R. As for WAW, in the same thread, Write_P can be overwritten by Write_C. However bypassing Write_P is not considered by BFix, since it is hard to generate a reasonable conditional checking (we will revisit this issue later). For OV, if A and B are both write operations (or if A is the write operation and along the call stack to B, there is an overwritten operation for the same address), A might could be bypassed.

Implication to patch generation  Figures 4.7, 4.8 and 4.9 show all simplified overwritten patterns for the AV and OV types that BFix handles and their patch patterns motivated by Figure 4.5. In terms of what operations to bypass, the overwritten buggy write operation (the former one of two consecutive writes in $T'$) should be skipped. In terms of what bypassing
predicate the patch should use (e.g., \texttt{bug\_context} in Figures 4.7, 4.8 and 4.9), a variable whose status change can reflect the occurrence of the buggy timing should be selected. Motivated by Figure 4.5, this variable should be context or address sensitive, since multiple instances might be involved. The bypassing patch should take effect only when buggy operations touch the same memory location. Additionally if it is a newly-introduced variable as illustrated in Figure 4.5, we need to carefully update it around the buggy/overwritten operations. In terms of where to insert extra conditional checking, it should be directly inserted before the overwritten write access. In terms of correctness proof, we can try to find the equivalence between $P_T$ and $P'_T$, utilizing the data overwritten property.

\subsection*{4.4.2 Conditional}

The conditional pattern involves buggy operations that lie in the existing conditional branch. This pattern utilizes control flow information from the original program. This pattern has two features: (1) $C$, $R$ or $A$ originally lie in a conditional branch (referred to as \texttt{cond}) of the original program; (2) the value of the conditional expression (referred to as \texttt{global\_flag}) is updated around other buggy operations in different threads. In this pattern, skipping operation $O$ in $P'$ includes all operations surrounded by \texttt{cond}. Since the execution behavior
```c
1 // Thread-1
2 Token* instance(){
3     ...
4     if(!fInst){
5         XML_Lock();
6         + if(fInst == NULL){
7             fInst = new TokenMap();
8             ...
9         } + }
10         XML_Unlock();
11 } +
12 }
13 return fInst;
14 }
```

Figure 4.10: A simplified bug from Apache Xerces. ‘+’ indicates the manual patch of skipping such operations could be found both in $P_T'$ and $P_T''$ due to the different paths taken, it is not difficult to prove they can generate the same semantics that skips $O$.

Figure 1.4 belongs to the conditional pattern. The reset procedure is inside a conditional branch in the original program. When reasoning the correctness of the manual bypassing patch, lookup→mStatus == COMPLETE checking plays a critical role. As shown in Figure 1.6, in a correct timing where lookup→mStatus == COMPLETE checking happens before lookup→mStatus = COMPLETE, lookup→mStatus == COMPLETE should be FALSE, and as a result, the reset of mStatus is skipped. This skipping reset of mStatus execution behavior $P_T'$ is the same as the patched version in buggy timing $P_T''$.

Figure 4.10 shows another conditional pattern example from Apache Xerces. When two threads concurrently execute the function instance, it is possible that fInst would be updated twice. If Thread-2 already reads and locally copies the returned fInst before the second update by Thread-1, it would lead to inconsistency issue. Similarly, the manual patch uses existing variable fInst for conditional checking to skip the second update in Thread-1.

The correctness of the bypassing patch of this example is also related to the existing condition branch in Line 4 of Thread-2. As shown in Figure 4.11, the patched program will skip the update to fInst in Thread-1 when the bug is about to happen. This execution behavior $P_T''$ is actually equivalent with the original program under a different timing ($P_T'$),
where Thread-2 conducts instance earlier and the update to fInst in Thread-1 is skipped since fInst is not NULL after Thread-2’s update.

(a) Execution timeline for the patched version when bug happens

(b) Execution timeline when Thread-2 conducts instance later

Figure 4.11: Execution timeline for the patched version in bug-triggering timing and a bug-free run in Figure 4.10

Figures 4.12, 4.13, 4.14 show all conditional patterns that BFix handles and their patch patterns motivated by Figure 1.4 and Figure 4.10. They have several cases, since global_flag can be updated to different values in different locations. For example, for the case where R is in an existing conditional checking, global_flag can be updated before P, after C and between P and C. Additionally, the update has two types: (1) Upglobal_flag: global_flag is updated to the same value as conditional branch checks; (2) !Upglobal_flag: global_flag is updated to the opposite value as conditional branch checks.

**Implication to patch generation** In terms of what operations to bypass, it is straightforward that all operations surrounded by the existing conditional branch cond should be skipped. In the conditional pattern, only skipping buggy operations is not enough, operations that depend on buggy operations should be skipped too. Such information is already encapsulated by the existing control flow structure cond. In terms of what bypassing predicate that patch should use, similar to the overwritten pattern, a variable bug_context whose status change can reflect the occurrence of buggy timing is introduced. Buggy variable itself can also serve as extra conditional checking, as shown in Figure 4.10, since the remote write
from other threads can cause value difference at location $P$ and $C$. This finding provides a way of choosing existing variable for condition synthesis in bypassing patch generation. In terms of where to insert extra conditional checking, we need to use \textbf{AND} to connect newly-introduced checking with existing conditional branch $cond$. In terms of correctness proof, we can try to find the equivalence between $P_T'$ and $P_T''$ utilizing existing control flow information.

\textbf{4.4.3 Exclusive}

The exclusive pattern involves two buggy operations that come from different threads and conduct the same exclusive function. This pattern utilizes implicit semantics of exclusive functions. Certain functions have exclusive property, that is – they can only successfully execute once for a specific memory location, further invocation on the same memory location is redundant and would fail. These functions always destroy resources, like \texttt{free} for memory resource and \texttt{shutdown} for socket resource. In this pattern, skipping operations $O$ in $P'$ should be the repeated invocation of the same exclusive function in $T$. Since the repeated execution of same exclusive function can always be skipped without violating the correctness, the
Figure 4.14: Simplified conditional pattern for bypassing A and the patch pattern

(a) Updating is opposite to the conditional check

(b) Updating is same as the conditional checking

Figure 4.15: All conditional patterns that BFix handles. global_flag is a shared memory location, it can be a variable or expression. For each case, the inferred correct timing and bypass-able operation are also shown

correctness proof for this pattern is very simple and it is done by checking the existence of exclusive functions.

Figure 4.16 shows a modified version of a bug in Apache belonging to the exclusive pattern. Thread-1 creates and inserts an entry obj_1 into a queue at the beginning and finally removes it from the queue and frees it by invoking cache_remove. Thread-2 tries to pop an entry obj_2 from the same queue and frees obj_2. When obj_1 and obj_2 point to the same memory location, and Thread-2’s cache_pop happens between Thread-1’s insertion and removal, the same object might be freed twice. The manual patch uses an existing function cache_find as conditional checking before Thread-1’s removal. When buggy timing happens, the entry would not be found in the queue, cache_find returns FALSE and as a
Figure 4.16: A modified version of a concurrency bug in Apache, ‘+’ indicates the manual patch result, Thread-1 would skip removal for this object.

The correctness reasoning for this patch is different from the former two patterns. It utilizes the semantics provided by exclusive functions. Firstly, since `free` operation has exclusive property, the repeated invocation of `free` for the same memory location is always redundant and could be skipped. Secondly, `free` will deallocate the memory address, which means it is the ending of the life cycle for this entry object and no further valid operations should access the same entry object (the pointed space would be deallocated, and the pointer would point to an invalid location). Hence the repeated `free` could be skipped here without violating the correctness.

**Implication to patch generation** In terms of what operations to bypass, the bypassing patch should skip the repeated invocation of same exclusive function in $T$. For Figure 4.16, the second invocation of `free` for the same memory location is skipped in bug-triggering timing 1. In terms of what bypassing predicate that patch should use, even developers directly leverage an existing function `cache_find` to check whether the object is removed or not, this function has the same property as previously introduced new variables – the return value of this function can reflect the occurrence of the bug timing. If automatic tools also

---

1. In bug-triggering timing, the removal procedure inside `cache_remove` is a silent behavior. It is read-only without changing any environment. Skipping `cache_remove` is actually semantically the same as skipping `free`. 

---
want to leverage this function, they need to explore the semantics of `cache_find` or ask the help from developers, which is different from Figure 4.10. And for this bug, the patch also cares about multi-instance or multi-thread problems. In terms of where to insert extra conditional checking, it should be inserted directly before the repeated invocation of the same exclusive function in $T$. In terms of correctness proof, we only need to check the existence of exclusive functions and whether their parameters touch the same memory location or not.

4.5 BFix

Guided by these three common bypassing patterns, BFix is designed to facilitate applying computation-bypassing strategy for concurrency bug fixing. In this chapter, we will firstly briefly introduce the workflow of BFix, and then introduce the design details about BFix, showing how BFix answers the questions about bypassed operations, synthesized conditions and correctness proof.

4.5.1 Workflow of BFix

The framework of BFix is depicted in Figure 4.17. Given the bug report from bug detectors, BFix will conduct several checks to determine whether it is suitable to apply computation-bypassing fixing strategy during the feasibility checking stage. Specifically, three checks are conducted.

**Pattern checking:** BFix checks whether the reported bug fit any one of three common bypassing patterns. Specifically, BFix checks (1) whether the buggy operations can be overwritten; (2) whether the buggy operations are conditional in the original program; (3) whether the buggy operations are used as arguments for exclusive functions. This step will identify which pattern the current program belongs to and the correct timing BFix infers.

**Synchronization checking:** BFix checks whether the correct timing it infers in the pattern checking can really happen. Specifically, BFix would check whether there are existing
synchronizations around the buggy code region, which hinder the occurrence of the identified correct timing. BFix focuses on condition-wait operations, barriers, thread join operations.

**Equivalence checking:** Pattern checking mainly focuses on the buggy operations or operations encapsulated with buggy operations. During equivalence checking, BFix considers all operations. BFix takes all operations’ execution order in the correct timing got from the pattern checking step and all operations’ execution order in the bug-triggering timing with certain code region skipped. Then BFix checks whether two series are semantically equivalent or not. This is mainly for the overwritten and conditional patterns.

Only when all checks are passed, BFix goes to patch generation, synthesizing conditional checking for the to-be-bypassed operations. Finally, BFix merges the patches for better performance and readability. The computation that will be bypassed depends on which pattern the original program belongs to, and this is answered in the pattern checking step. The patch generation step shows the details about what condition is synthesized. As for the correct proof, BFix relies on all checks conducted in the feasibility checking step.

4.5.2 Feasibility checking

Firstly, BFix will conduct **pattern checking**, where it focuses on three common bypassing patterns (Overwritten, Conditional and Exclusive patterns). BFix can only help generate
correct bypassing patch for bugs that fit any one of these three patterns. Here we will introduce how to check original program matches them.

Checking algorithm for Overwritten Pattern  Given the bug report for a concurrency bug, BFix determines it is an AV or OV bug by checking the number of input buggy operations. For an AV bug, BFix checks the access types of $P$, $C$ and $R$. If it belongs to RAW or WAR cases, BFix marks it as an overwritten pattern, and records the bypassing candidate $R$ or $C$ (BFix does not need to check whether they write to the same memory location or not, since the bug detector has verified that in the buggy timing.). For an OV bug, similarly, BFix checks the access type of $A$ and $B$, if (1) $A$ and $B$ are both write operations, BFix marks this case as overwritten, or (2) $A$ is a write operation, and along the call stack of $B$, there is a overwritten operation to the same address, BFix marks this case as an overwritten pattern too.

BFix covers complex overwritten cases, like the relation of writing to a filed of a struct and writing to the whole struct and some commonly-used overwritten library functions, like memset etc.

Checking algorithm for Conditional Pattern  BFix checks whether the reported bug belongs to the conditional pattern or not based on the control flow structure. As mentioned in chapter 4.4.2, because of different updating values and locations for $\text{global\_flag}$, there are 12 conditional cases that BFix can handles, as shown in Figure 4.15. The differences between updating value and location would affect what type of correct timing BFix will search. For example, for the case where $R$ is in an existing conditional checking, if $\text{global\_flag}$ is set to the opposite value of the conditional branch checks, BFix will mainly search PCR correct timing. Actually as long as the branch checking happens after $\text{global\_flag}$ updating, no matter PCR or RPC, they can generate the semantics where $R$ is skipped. However, it is more natural to search PCR since $RPC$ happens in a stricter timing. Which correct timing
BFix infers does not affect the correctness of generated patch.

The checking algorithms for these 12 cases are similar. Due to the space limit, we only show details for (a)(1) and (b)(1), others can be easily inferred.

The checking algorithm is straightforward. Given the location of $P$, $C$, $R$ and their call stacks, firstly BFix backwardly searches the branches and their corresponding conditions from $R$ along the call stacks. Currently, BFix searches at most 5-level callers along the call stacks. This threshold is configurable. For the compound condition, the current version of BFix only considers \textbf{AND} and \textbf{NOT} binary operators for simplicity. For other cases, BFix simply ignores them.

Secondly, for each recorded simple branch condition, BFix filters it out if it (1) has function calls (to avoid the expensive inter-procedural analysis); (2) only has local variable read; (3) has write operations. For each remaining condition $\text{cond}$ after filtering, BFix records global/heap variable read ($\text{global\_flag}$), whose value can be affected by other threads. For the compound condition, BFix decomposes it and analyzes each part separately.

Thirdly, BFix backwardly searches $!U_{\text{p\_global\_flag}}$ or $U_{\text{p\_global\_flag}}$ (referred to as $\text{cond\_point}$) from $P$ ($P$ is included) for each $\text{global\_flag}$. Similarly, BFix searches at most 5-level callers along $P$’s call stack. And BFix constructs post-dominate tree to make sure this $!U_{\text{p\_global\_flag}}$ or $U_{\text{p\_global\_flag}}$ is executed with $P$ and $C$ every time. Also BFix checks there is no other updating to $\text{global\_flag}$ between $\text{cond\_point}$ and $P$. BFix uses LLVM built-in alias analysis. Only when it is reported as \textbf{MUST} alias, BFix considers two variables touching the same memory location. For the case of updating to the same value as branch checking checks, as in (b)(1), BFix conducts safety checking, to make sure the value before the updating point in Thread-1 is actually FALSE along the call stack of $P$. Only when it is actually FALSE, BFix marks it as a conditional pattern. Similarly, BFix searches at most 5-level callees along $P$’s call stack.

Taking Figure 1.4 as an example, BFix firstly finds reset of $\text{m\_Status}$ surrounded by a
if checking in Line 5 of FindLookup. BFix then extracts the checking conditions from that if checking, where it checks heap variable mState equals COMPLETE or not. After that, BFix backwardly searches updating location for this heap variable from Line 27 in FireStop, which is the other buggy location reported by the bug detection. Along the call stack, BFix finds the statement in Line 8 of ProcessLookup (the caller of FireStop function). At this point, mState is updated to COMPLETE, and this would serve as the evidence that resetting procedure is bypassable.

**Checking algorithm for Exclusive Pattern**  Since the correctness reasoning of exclusive pattern heavily relies on the implicit semantics of exclusive functions, BFix carefully chooses exclusive functions.

Currently, BFix only considers free for memory operation and shutdown for socket operation. Given the bug report, BFix checks whether buggy operations served as arguments for such functions in the buggy code region. If yes, BFix considers it as an exclusive pattern. Developers can add more exclusive functions into the configure files of BFix, like adding vio_close for MySQL application.

The second checking in feasibility checking stage is synchronisation checking. For the correct timing from which BFix infers whether certain buggy operation is bypass-able, BFix also need to check whether it can really happen or not. Basically, BFix determines whether there are existing synchronisation operations (condition-wait operations, barriers, thread join operations, etc.) around the buggy code region.

For AV bugs, BFix checks whether there are mentioned synchronisation operations between P and C 2. If yes, the inferred correct timing PCR or RPC might not actually happen, and BFix conservatively stops.

For OV bugs, such check is not needed. Since BFix always infers from the correct timing AB, if this order can not happen, the bug could always manifest.

2. It also includes the region between P/C and !Upglobal_flag/Upglobal_flag.
The third checking **equivalence checking** is directly with correctness proof. To prove the correctness of bypassing patch, basically, BFix checks the equivalence between the original program $P$ in a correct timing $T'$ and the patched program $P'$ in bug-triggering timing $T$. During the pattern checking, BFix mainly focuses on buggy operations or operations encapsulated with buggy operations, comparing their equivalent execution behavior in $T$ and $T'$. That is not enough. Since the execution order among other unchecked operations might be different in $T$ and $T'$, new semantics might be introduced. But if there are no dependencies among them, their execution could be reordered without violating correctness. As a result, equivalent execution behavior between them can be found. But if there are no dependency among them, their execution could be reordered without violating correctness. As a result, equivalent execution behavior between them can be found. In general, the independency relations among certain regions should be checked as well for correctness proof.

![Diagram](image)

(a) The execution of region $N$ is advanced in bypass-patched program for (a)(1) in Figure 4.15

(b) The execution of region $M$ is postponed in bypass-patched program for (b)(1) in Figure 4.15

Figure 4.18: Bypass-patched program may change the execution order of original program; dotted arrows means execution order is changed; same notation is used as Figure 4.3

Depending on the operations that are to be bypassed and the correct timing BFix infers, different code regions should be checked. Taking (a)(1) and (b)(1) in Figure 4.15 as examples, different region relations should be verified for the full correctness proof, as shown in Figure 4.18. For (a)(1), in $T'$, all operations $I_N$ of region $N$ are conducted after $C$, while $I_N$ are conducted after $P$ in $T$, which means the execution of $I_N$ is advanced, if we can prove all
operations $I_Y$ are independent with $I_N$ and $C$ is independent with $I_N$, then even we postpone the execution of $I_N$, it does not affect the correctness because of the accepted reordering by memory model \(^3\). As a result, the serialized interleaving of the patch program in $T$ is proven as a subset of the serialized interleaving of original program in $T'$, the correctness of bypassing patch is hence proven. Using the same analysis method, for (b)(1) we need to prove the independency between region $M$ and $P$ \(^4\), region $M$ and region $Y$.

In the remaining part of this dissertation, if two operation sequences $J$ and $K$ are data and control independent, this relation would be denoted as $J \perp K$.

The basis philosophy of equivalence checking is: for two regions $J$ and $K$ from different threads, in $T'$, their execution orders are determined (e.g. all operations in $T'$ are executed after operations of $K$). While in $T$, the operations inside these two regions can interleave with each other, BFix tries to check whether the relation $J \perp K$ is held or not.

**What to check** Based on which correct run BFix infers, different code regions should be checked \(^5\).

- **AV-bypassing $R$**: If the correct timing that BFix infers belongs to $PCR$, BFix checks whether $C' \perp N$ and $Y \perp N$ or not. If the correct timing that BFix infers belongs to $RPC$, BFix checks whether $M \perp P'$ and $M \perp Y$ or not.

- **AV-bypassing $C$**: If the correct timing that BFix infers belongs to $PCR$, BFix checks $Y \perp R'$ and $Y \perp N$. If the correct timing that BFix infers belongs to $RPC$, BFix checks $M \perp P'$, $M \perp Y$, $R' \perp Y$ and $R' \perp P'$.

---

3. BFix uses PSO memory model for reordering transformation. For any read and write(read) in the same thread, if they don not touch the same memory location, their order can be flipped. It is also applied to the write and write operations.

4. Actually it should be the region between $Up\_global\_flag$ and $P$.

5. $P'$, $C'$, $R'$ and $B'$ refer to the operations between updating point $\text{cond\_point}$ and $P$, $C$, $R$ and $B$. 

79
• OV-bypassing A: The correct timing BFix infers can only belong to AB, BFix checks $C' \perp B'$ and $C'' \perp F$.

How to check  If at least one of region $J$ and $K$ is empty, then $J \perp K$. If $J$ and $K$ are both the regions that contain at least one statement and are bounded 6, BFix checks $J$’s and $K$’s read and write sets along their call stack separately, then check whether they have read-and-write, write-and-read, or write-and-write dependency. All shared variable accesses in synchronization primitives are considered as read. In the current prototype, BFix uses the default MAY-pointer-alias in LLVM to decide whether two sets of operations read/write the same memory location. If at least one region is unbounded, BFix uses the following rules to conduct dependency checking (for conditional case, global_flag expression cond would be removed from the comparison, since its value is TRUE or FALSE, and BFix knows its value in bug-triggering timing and the correct timing BFix infers).

1. If at least one of $J$ and $K$ only access the local variables, then $J \perp K$.

2. If for all heap/global variable accesses in $J$ (or $K$), before $K$ (or $J$), they are destroyed (currently BFix focuses on destroying functions free, gc, etc; developers can also provide such information through configure file), then $J \perp K$.

3. If there are no reported concurrency bugs in region $J$ and region $K$, then $J \perp K$. (This means operations in $J$ and $K$ may touch the same memory location, but their order is enforced by existing synchronizations and can not interleave arbitrarily).

4. Either current thread is terminated in $J$ or $K$ (e.g. kill and cancel of pthread library, these are configurable for BFix) or $J$ or $K$ is directly in the main function. In this

---

6. A region is bounded if its range can be determined statically. If the starting or the ending point of a region could be as far as the start or the end of the whole thread, then this region is unbounded. Except the region $P, Y, C, R$ for AV bugs and $A, B$ for OV bugs, all other regions are unbounded for BFix.
case, BFix can infer that the ending point of the region \( J \) or \( K \) is bounded, and uses the algorithm described above to check \( J \perp K \).

5. Current thread is created in \( J \) or \( K \) (e.g. \texttt{pthread_create} in \texttt{pthread} library). In this case, BFix can infer the the starting point of the region \( J \) or \( K \) is bounded, and uses the algorithm described above to check \( J \perp K \).

After passing the equivalence checking, the same semantics generated by the threads involved with bug manifestation are always found in the correct timing space. As for other threads, since we can always advance or delay them in the correct timing space to generate the same schedule as in the bug-triggering timing space, it follows that their equivalent semantics can be found in the correct timing space too. Then, BFix finds the equivalent execution behavior for all operations. The full correctness proof is completed.

\textit{4.5.3 Patch generation}

The patch generation for BFix is motivated by the hints obtained from the manual patches shown in chapter 4.4. Firstly, it is possible that one operation has multiple instances. The patch should only take effect for the instance, that leads to the bug-triggering timing. Secondly, if introducing new variables as conditional checking, such variables should update around the buggy region. Thirdly, buggy variable itself sometimes can be used as conditional checking. Fourthly, regardless of whether the bypassing patch is using existing variables or introducing variables, their status change should reflect the occurrence of the bug-triggering timing.

In this chapter, we will introduce how to generate the bypassing patch when all feasibility checks are passed. We will discuss where to insert the conditional checking, what the conditional expression is and what the bypassing patch looks like. BFix uses the same function clone technique as CFix in order to make sure the generated patch only affects the execution of the buggy call stack.
**Which computation to bypass**  Benefited by the three adopted common bypassing patterns, which computation to bypass in the bug-triggering timing is straightforward for BFix. For the overwritten pattern, BFix would bypass the former write access of two consecutive write accesses. For the conditional pattern, the related operations have been encapsulated by if-else body, BFix would bypass such operations. For the exclusive pattern, BFix would bypass the repeated invocation of exclusive function of buggy interleaving, which takes buggy variables as arguments.

**What is the condition**  There are two ways to synthesize variables or expressions for conditional checking. One is to directly use existing variable/expression/function, which may take advantage of existing well-structured logic. However, it would need more semantics knowledge, making the automated patching complex or requiring greater effort from the developers, like Figure 4.16. The other way is to introduce a new variable/expression and carefully maintain it so that its status change can reflect the occurrence of the buggy timing. This is a much simpler and more general approach that does not require extra semantics. Mainly, BFix prefers to introduce a new variable to reduce the semantics requirements. But for certain cases, BFix uses existing variable/expression to generate simple patch instead.

**Introducing a new variable**  BFix designs a general patch pattern for this case, which only uses the nature of concurrency bugs without requiring any extra concrete semantics information.

Figure 4.19 and Figure 4.20 show what the patch patterns for bypassing $r$ and $c$. And Figure 4.21 shows what the patch pattern for bypassing $A$.

BFix designs a general patch pattern for this case, which only uses the nature of concurrency bugs without any extra concrete semantics information.

Figure 4.19 and Figure 4.20 show patch patterns for bypassing $R$ and $C$. Figure 4.21 shows the patch pattern for bypassing $A$.  

82
For each bug report, BFix records the run-time address and thread ID for each buggy operation via our `recorder` function. For AV bugs, BFix bypasses the bypassing-candidate operations only when: (1) \( P \), \( C \) and \( R \) touch the same memory location; (2) \( P \) and \( C \) are in the same thread, and \( R \) is in another thread; (3) they are about to construct the buggy order \( PRC \). Similarly, for OV bugs, only when (1) \( A \) and \( B \) touch the same memory location; (2) \( A \) and \( B \) are in different threads; (3) \( B \) is executed before \( A \), BFix bypasses the bypassing-candidate operations. Function `recorder` and `check_happen` maintain a protected map, which uses address information as key. `check_happen` function will return TRUE when the bug is about to happen, otherwise, it will return FALSE. It achieves this by analyzing the recorded sequences, checking whether a remote \( R \) exists between another thread’s \( P \) and \( C \) or whether a thread’s \( B \) is before another thread’s \( A \). For the conditional pattern, `check_happen` is inserted before the existing conditional expression using the AND operation to connect it with existing conditional expression.

As mentioned before, bypassing \( P \) for AV bugs is not a choice for BFix. Since when \( P \) is executed, \( C \) and \( R \) are not executed, it could be a bug when \( R \) is executed before \( C \), and could not be when \( C \) is executed before \( R \). Those are uncertain for BFix. Hence BFix would
if(!check_happen(pid1,'A',&A)) {
    // check whether it can
    // really construct OV bug
    A
    +recorder(pid1,'A',&A);
}

+recorder(pid2,'B',&B);

Figure 4.21: Patch pattern for bypassing A (pid is thread ID)

not try to bypass P for AV bugs.

Usually there is no need to predict address of buggy operations before their execution. However for the conditional pattern, BFix needs to predict memory address in advance, especially when if checking cond_point and buggy operations are in different functions. If prediction is needed, BFix backwardly checks how the address is calculated along the call stack, and records such information until encountering cond_point. Only when the calculation (the way of calculating the base and the offset) is idempotent, BFix copies the calculation (replacing and using the correct variable names) around the existing conditional checking (cond_point). Otherwise, BFix can not guarantee to predict the correct address and fails to generate the bypassing patch.

Using an existing variable Even when trying to use existing variables, BFix does not want to explore much extra semantics information as that would complicate our design; instead, it only tries to explore the chance of using buggy variable as conditional checking. For AV bugs, we may have chance in read-after-read, read-after-write, and write-after-read cases, since the remote access for the same shared memory is a write operation. The buggy variable itself might also satisfy the property that status change can reflect the occurrence of bug timing. If the bypassing-candidate operation is C, in the correct timing, the value of the buggy variable should be the same in location C and location P; in the bug-triggering timing, this equality is not held because the remote write R happens in between and changes the value of this buggy variable.

However, simply checking the equality in location P and location C might introduce new
bugs. Figure 4.22 shows a make-up example. The patch simply checks whether a’s value is changed or not. If yes, then buggy timing would happen, and it bypasses Line 5 of Thread-1. However, this pattern would introduce extra access for a in Line 4, hence leading to another AV bug triplet.

BFix only uses buggy variables as conditional checking, when (1) the bypassing-candidate operations include C; (2) C and R are protected by the same lock; (3) R is a write access; (4) P and C are in the same function (to avoid inter-procedural copy); (5) if P is a write access, its content should be idempotent (to avoid changing semantics during copy).

Figure 4.23 shows the patch pattern of using an existing variable. BFix would do local copy for P around P, and then compare the local value and the global value of P. This simple patch pattern has higher adoption priority than our general patch pattern, since it is much simpler. BFix would check the chance of applying a simple patch before applying a general patch for a given bug.

Discussion Without the protection of locks, there would be a small buggy timing window even with the bypassing patch applied. Moreover, our introduced functions recorder and check_happen totally get rid of the lock primitives, hence they could be another bug triplet.
Fortunately, BFix can sometimes take advantage of existing locks to fundamentally fix the reported bug, as demonstrated in Figure 1.4. This happens especially for AV bugs. If $P$, $C$ and $R$ are already protected by the same lock, then our patch can fundamentally fix this bug. BFix checks whether $P$, $C$ and $R$ are protected by the same lock along the call stacks. If they are found to be protected by the same lock, then the generated patch can fundamentally fix the reported bug and BFix reports this information to users. If not, our bypassing patch is still appealing, since it has good readability and can significantly decrease the chance of bug occurrence, leading to good performance, since usually the $P - C$ region is very long and may be crossing several call stacks.

4.5.4 Patch merging

For two given bug reports, BFix would generate the patch for them separately if applied. BFix would then conduct some simple merging if they are same type of bugs and have one of the following relationships:

1. If two given bugs share same buggy operations, then BFix would only generate one bypassing patch.

2. If two bypassing patches bypass two neighboring operations, and (1) these two to-be-skipped code regions have dominating and post-dominating relation; (2) no other operations exist in between (or those operations in between are not necessary, like log information for a non-log mode program running in MySQL, this is configurable by developers); (3) they share the same memory locations for other corresponding buggy operations, then BFix would generate one conditional checking for those operations instead of two conditional checking for each of them.

3. If two bypassing patches bypass the same operation at the same program location.

7. For simple patch case, only $C$ and $R$ are protected by the same lock is sufficient.
or they have inclusion relation, and they share the same memory locations for other corresponding buggy operations, BFix would generate only one condition checking for the bigger region.

4. If the bypassing operations of two patches lie in the same existing conditional checking (cond_point), and they share the same memory locations for other corresponding buggy operations, BFix would generate only one condition checking before cond_point.

4.6 Methodology

**Implementation** BFix is implemented using LLVM infrastructure (v3.6.1). All the experiments are conducted on 4-core Intel Core i7-5775C machines with 6MB cache, 8GB memory running 2.6.32, and O4 optimization level.

**Benchmark suit** We have compared manual bypassing patches, the patches generated by BFix, CFix and HFix on 8 real-world bug benchmarks. They are the bugs, whose manual patches are marked as bypassing in HFix [39]. In HFix, there are 14 bugs with bypassing manual patches in total. However, we only conduct experiments on those 8 bugs, because they are well-documented so that we could reproduce them. The remaining bugs were reported around 20 years ago and could not be reproduced based on the descriptions. More importantly, none of those 6 bugs belong to our three common bypassing patterns. Our benchmark suits include server applications (e.g., MySQL database server, Apache HTTPD web server), network applications (e.g., Apache Xerces-C++ XML parser), and desktop application (e.g., Mozilla JavaScript Engine). The sizes of these applications range 50K – 1 million lines of code.

**Setups and metrics** We evaluate the quality of BFix patches by comparing them with manual patches, and the patches generated by the state of the art concurrency-bug fixing techniques, including synchronization-adding tool CFix and synchronization-moving tool HFix. We compare them based on three criterion, correctness, performance and simplicity.
To measure the patch correctness, we follow the methodology of previous work [27, 28, 39, 82], and insert sleep into software, so that the corresponding bugs will manifest frequently. We then run each bug-triggering workload with the manual patches or each tool applied for 1000 times.

To measure the patch performance, we run the original software without any sleeps with the manual patches or each tool applied. We report the average overhead measured during 100 failure-free runs, reflecting the performance during regular execution.

To measure the patch simplicity, we mainly compare the number of synchronization operations introduced by the manual patches, BFix, CFix and HFix. We also count the lines of code modification at source code level for those patches (Even the patching happens at IR level, in order to compare with the manual patches, we try to convert them and count the lines of code modification at source code level.).

### 4.7 Experimental results

Although CFix patches can help eliminate all those 8 benchmarks, they usually have worst performance (around 20% for some of them). HFix patches has great performance, but can not be applied for 4 of them. BFix patches has the best performance and simplicity. As for correctness, BFix can help eliminate 5 of them, for the remaining 3 bugs, BFix can dramatically decrease the failure rates, similar to the manual patches. Unlike CFix patches, BFix patches do not introduce any new synchronizations. In general, BFix patches achieve the best balance among correctness, performance and code readability. Additionally, they have much performance and readability advantages over the patches generated by CFix and HFix.
Table 4.1
Failure Rates comparison

<table>
<thead>
<tr>
<th>Root Cause</th>
<th>Origin Manual</th>
<th>BF ix</th>
<th>CF ix</th>
<th>HF ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla85802</td>
<td>AV 100% 0% 12% 0% 0%*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mozilla72599</td>
<td>AV 97% 0% 0% 0% 0%*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache21287</td>
<td>AV 83% 0% 0% 0% 0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache38793</td>
<td>AV 86% 12% 10% 0% NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache21285</td>
<td>AV 98% 0% 0% 0% NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache17516</td>
<td>AV 100% 0% 0% 0% 0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL19437</td>
<td>AV 71% 0% 0% 0% NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mozilla73761</td>
<td>AV 91% 8% 9% 0% NA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NA: not applicable; *: original HF ix can not move lock/unlock across loops, since they do not want to change the number of times that lock executes. Here we change it to make it can move lock/unlock across loops.

4.7.1 Correctness

Table 4.1 shows the failure rates of the original software and the failure rates when applying manual patches, the patches generated by BF ix, CF ix and HF ix. The sleep s are inserted for bug manifestation.

For manual bypassing patches, they can help eliminate most of bugs except Apache38793 and Mozilla73761, since for these two bugs, manual patches can not leverage any existing synchronization operations (e.g., lock). Even inserting the conditional checking for bypassing, there is still a time window for bug manifestation, as discussed in chapter 4.5.3. It is similar to BF ix patches. For Mozilla85802, shown in Figure 1.4, the manual patch can even leverage the lock in $P - C$’s caller to prevent from bug occurrence. However, in current BF ix prototype, only when $P$, $C$ and $R$ (or $A$ and $B$) are protected by the same lock, BF ix patches can fundamentally fix the reported bugs. That is why BF ix fails to help eliminate the bug in Mozilla85802. BF ix correctly tells the cases that it can not fundamentally fix to users.

CF ix can help eliminate all bugs, considering its generality design goal. However, as shown in chapter 4.7.2 later, this aggressively sacrifices the performance. CF ix patches have the highest run-time overhead, since the introduced critical regions may include many
Table 4.2
Performance comparison

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>BFix</th>
<th>CFix</th>
<th>HFix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla85802</td>
<td>0.22%</td>
<td>0.31%</td>
<td>12.90%</td>
<td>11.41%</td>
</tr>
<tr>
<td>Mozilla72599</td>
<td>0.81%</td>
<td>0.75%</td>
<td>3.21%</td>
<td>3.62%</td>
</tr>
<tr>
<td>Apache21287</td>
<td>0.11%</td>
<td>0.23%</td>
<td>0.28%</td>
<td>0.26%</td>
</tr>
<tr>
<td>Apache38793</td>
<td>0.88%</td>
<td>0.94%</td>
<td>18.95%</td>
<td>NA</td>
</tr>
<tr>
<td>Apache21285</td>
<td>0.39%</td>
<td>0.42%</td>
<td>14.47%</td>
<td>NA</td>
</tr>
<tr>
<td>Apache17516</td>
<td>0.21%</td>
<td>0.32%</td>
<td>0.55%</td>
<td>0.43%</td>
</tr>
<tr>
<td>MySQL19437</td>
<td>0.91%</td>
<td>0.90%</td>
<td>2.81%</td>
<td>NA</td>
</tr>
<tr>
<td>Mozilla73761</td>
<td>0.71%</td>
<td>0.82%</td>
<td>17.6%</td>
<td>NA</td>
</tr>
<tr>
<td>Average</td>
<td>0.47%</td>
<td>0.57%</td>
<td>15.3%</td>
<td>5.92%</td>
</tr>
</tbody>
</table>

NA: not applicable;

time-consuming structures or operations, like loops, IO, etc.

Originally HFix can not help 6 out of those 8 bugs, because the synchronization movement in HFix has lots of constraints. Firstly, it would fail to help if it can not find existing synchronizations (locks, thread_join, thread_create) to move around, this happens to Apache38793 and Mozilla73761. Secondly, to avoid complex inter-procedural analysis and keep the original code encapsulation, it can not help move synchronizations across different functions. Hence even having existing locks to move around, HFix can not help for MySQL19437 and Apache21285. Thirdly, HFix would not move synchronizations across loops. However this design only affects the performance without violating correctness, so here we make it possible to move synchronization across loops when there are no other synchronizations along the moving path. After this change, HFix can help fix Aapache38793 and Mozilla73761.

Even not like CFix patches, which can fundamentally fix all those 8 bugs, BFix patches can eliminate most of them, similar to the manual patches and better than HFix. And as shown later, BFix has the best combination of correctness, performance and simplicity.

4.7.2 Performance

Table 4.2 shows the regular-run overheads of applying manual patches, the patches generated by BFix, CFix and HFix.
Figure 4.24: A patch pattern used by developers

Overall, the manual bypassing patches have the best performance, incurring less than 1% overhead for all benchmarks at run time. It is not surprising, since developers usually have rich semantics knowledge, they can make the best choice, like directly leveraging existing functions/expressions/variables for conditional checking.

BFix patches have slightly worse performance than the manual ones. The overhead mainly comes from address/thread_id recording and determining the bug is about to happen or not in the newly-introduced conditional checking. The overhead for all benchmarks is less than 1% too, similar to the manual patches and much better than CFix and HFix.

For Mozilla72599 and MySQL19437, the manual patches use the same pattern as Figure 4.24 (For MySQL19437, R is already in a if body.). This patch pattern can help eliminate bug, but it also prevent the original correct interleaving PCR. For BFix, it wants to maximally maintain the original correct interleavings (since they might get benefit of parallelism). BFix only fixes the buggy interleaving without preventing from any original correct interleaving. Hence, BFix adopts the patch patterns as Figure 4.19 for them. And as shown in Table 4.2, for these two bugs, BFix patches have slightly better performance than the manual patches.

As mentioned before, the goal of CFix is to automatically correct patches for most types of concurrency bugs. It is not doubt that performance is also its concerns. However, the introduced critical regions may contain time-consuming operations. Worse, for AV bugs, if P and C are in different functions, CFix would try to search the call stack to find a common function to create a critical region inside it surrounding P and C, this can lead to much larger critical region when P is far from C. These are the main reasons why CFix has the worst
Table 4.3
Simplicity comparison (#Sync: number of new synchronization operations added by the patch; #line: line of code modification; L: lock; U: unlock; S: signal; W: wait; +: add; -: remove; ↑: move)

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>BFix</th>
<th>CFix *</th>
<th>HFix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Sync</td>
<td>#line</td>
<td>#Sync</td>
<td>#line</td>
</tr>
<tr>
<td>Mozilla85802</td>
<td>0</td>
<td>+3</td>
<td>0</td>
<td>+4</td>
</tr>
<tr>
<td>Mozilla72599</td>
<td>1L,1U</td>
<td>+4</td>
<td>0</td>
<td>+4</td>
</tr>
<tr>
<td>Apache21287</td>
<td>0</td>
<td>+3</td>
<td>0</td>
<td>+4</td>
</tr>
<tr>
<td>Apache38793</td>
<td>0</td>
<td>+3</td>
<td>0</td>
<td>+4</td>
</tr>
<tr>
<td>Apache21285</td>
<td>0</td>
<td>+3</td>
<td>0</td>
<td>+4</td>
</tr>
<tr>
<td>Apache17516</td>
<td>0</td>
<td>+1</td>
<td>0</td>
<td>+2</td>
</tr>
<tr>
<td>MySQL19437</td>
<td>1L,1U</td>
<td>+3</td>
<td>0</td>
<td>+4</td>
</tr>
<tr>
<td>Mozilla73761</td>
<td>0</td>
<td>+4</td>
<td>0</td>
<td>+3</td>
</tr>
</tbody>
</table>

NA: not applicable; *: For CFix patches, we do not count the line for global lock initialization when counting #line;

performance, with 15.3% overhead on average for our benchmarks. The patches generated for Mozilla85802 and Mozilla72599 would include loops. For Apache38793, the introduced critical region would contain the 6-step proxy connection. For Apache21285, CFix patch would include the whole life cycle of dynamic object caching and garbage collection would be contained for Mozilla73761.

Except the benchmarks HFix fails, HFix patches for Mozilla85802 and Mozilla72599 would include existing loops as well. And since HFix would not introduce new synchronizations, its patches usually have better performance than CFix. But as described before, it can not help most of benchmarks given its fixing constraints.

4.7.3 Simplicity

For the manual patches or the patches generated by BFix, CFix and HFix, we count the newly-introduced synchronization operations and the lines of code modification at source code level 8. Table 4.3 shows the detailed result. Overall, BFix patches are as simple as the manual ones. And they are much more readable than CFix/HFix patches.

8. For the newly-introduced functions, we only count it as 1-line code modification, since they can serve as library call and be dynamically linked.
The manual bypassing patches and the patches generated by BFix usually do not introduce new synchronizations. They either leverage the existing locks or simply give a patch which can not fundamentally fix the bugs but dramatically decrease the failure rates. The bypassing manual patches and the patches generated by BFix usually needs 3-4 lines of code modification, including 3-line conditional variables modification for AV bugs (2-line conditional variables modification for OV bugs), and 1-line conditional checking. Usually the bypassing manual patches are different from the patches generated by BFix, since manual patches can greatly utilize the domain knowledge that developers have and hence are a little bit simple, such as Mozilla72599 and MySQL19437. However, they essentially share the same philosophy. Taking Figure 1.4 as an example, both our recorder function in BFix patch and mProcess flag in its manual patch are used to depict the buggy timing, while our check_happen function in BFix patch and mProcess == 0 in its manual patch check the occurrence of buggy timing similarly. Sometimes, the patch generated by BFix is same as the one designed by developers. For Apache 17516, both manual patch and BFix patch adopts the simple patch pattern as Figure 4.23, hence it only needs 2-line code modification.

CFix patches can help eliminate all 8 bugs because of its aggressive usage of synchronizations, as shown in the CFix column in Table 4.3. Usually for AV bug, it needs 1 lock/unlock pair to protect the region between P and C, and another lock/unlock pair to protect C. If there are complex control flows around the buggy snippet, then CFix needs more synchronizations to make sure synchronizations paired during run time, like Apache38793.

For the HFix patches, they usually need move the existing lock/unlock to extend the critical regions and when it is needed, they need also copy lock/unlock in the related path to guarantee the correctness (for Apache17516 case). For Mozilla85802 and Mozilla72599, since we change the original HFix design a little bit, we need also remove the extra 2-line lock/unlock to avoid the deadlock here.

In general, BFix patches are simple as the manual patches without introducing new
4.8 Discussion

As shown in chapter 4.7, not like CFix, BFix can not fix all benchmarks. Our goal is not to design a new tool that can help fix most concurrency bugs. We try to provide a new fixing strategy to help developers generate high-quality patches, not introducing new bugs with low overhead and great readability. And our bypassing strategy is fundamentally different with other fixing strategies which have been automated, it gets involved with software semantics and does not eliminate the buggy timing. Our tool can complement the existing auto-fixing tools, offering diverse choice to facilitate the fixing process for developers.

Even for the bugs whose patches are about bypassing, BFix can not help automatically fix all of them, due to the complex correctness reasoning, due to the complex correctness reasoning. BFix only instantiates three equivalent semantics patterns, which widely exist (they cover 8 out of 14 bugs, whose manual patches are about bypassing in HFix.) and can help easy correctness reasoning. For the bugs which do not belong to any of these three patterns, BFix fails. Figure 4.25 shows a simplified Mozilla bug which BFix can not help. When layouting text, the Mozilla browser uses mOffset and mLength together to mark the region of useful characters stored in mContent. Without synchronization protection, Thread-1 may read inconsistent value of mOffset and mLength. The patch would bypass the characters which would be displayed in the browser when bug manifests. Since the browser would keep
flushing and finally the correct content would be displayed for the end users, visually, the effect of this bypassing patch is acceptable. However, it is really hard for the automated tool to reason the correctness for this patch without such domain knowledge. We admit there are also other situations that BFix fails to help. However, we still believe our work is worthwhile and can facilitate the fixing process for developers.

For big multi-threaded software, it is infeasible to prove the correctness of patched software. Following the methodology of previous work [27, 28, 39, 82], we made our best effort in correctness checking through manual examination and anecdotal patch testing. More importantly, our tool statically determine the bypass-patched program does not introduce new semantics, and the bypassing semantics already exist in the software. This is very crucial for the semantic-related patch proof. We are confident in the correctness of BFix patches because of both the soundness guarantee of BFix and the similarity between BFix patches and manual patches.

4.9 Conclusion

We have described BFix, a framework for automatically fixing concurrency via bypassing strategy. It can help fix certain concurrency bugs without eliminating the buggy timing, fundamentally different with the existing auto-fixing tools. We have implemented the system and shown BFix can general high-quality patches for certain concurrency bugs, outperforming the state-of-the-art auto-fixing tools CFix and HFix. BFix conducts through static analysis to generate provably correct patches and reach a good balance among correctness, performance, and code readability. Overall, BFix complements the existing auto-fixing tools, offering more fixing choice to help developers fight with concurrency bugs during the late phases of software life cycle.
CHAPTER 5

CONCLUSION AND FUTURE WORK

Concurrency bugs are difficult to avoid, detect, diagnose, and fix. Even with lots of in-house testing, some concurrency bugs can still slip into production run, threatening software availability and reliability. In this dissertation, we propose two approaches to help developers fight with concurrency bugs during the late phases of software life cycle.

For the concurrency bugs which fail to be detected during in-house testing, we present BugTM that leverages HTM available on commodity machines to help automatically recover concurrency-bug failures during production runs. BugTM can recover failures caused by all major types of concurrency bugs and incurs very low overhead. BugTM does not require any prior knowledge about concurrency bugs in a program and guarantees not to introduce any new bugs. We believe BugTM improves the state of the art of failure recovery, presents novel ways of using HTM techniques, and provides a practical and easily deployable solution to improve the availability of multi-threaded systems with little cost.

BugTM can not recover from all concurrency bugs. In the future, firstly, we hope techniques such as xCall [72] can be applied into BugTM to further improve its recovery capability. Secondly, we hope to deploy BugTM into a commercial system to see its challenges and opportunities in a real commercial environment, since availability and reliability are very important for commercial systems, and they can directly affect thousands of end users. Thirdly, it would be very interesting to deploy BugTM into distributed systems. We know it would be much more challenging, but it is still worthwhile.

For the concurrency bugs which can are detected during in-house testing or those discovered by the diagnosis aspect of BugTM, we present BFix, which is the first automatic tool facilitating to apply bypassing strategy for concurrency bug fixing. Unlike most existing tools, it does not introduce new synchronizations. The BFix patches are performance-friendly and as simple as the patches generated by developers. BFix complements the existing tools,
providing more choice to help developers generate high quality patches.

In this direction, since our current three common bypassing patterns can still fail to cover lots of cases, in the future, we hope explore more patterns for bypassing and give a more general solution for correctness reasoning. Also, since in distributed systems, there are lots of code-bypassing scenario, such as when keeping trying to connect with master if slave encounters bad machine state, current retrial including the subsequent non-executed code region would be bypassed. And the system would just keep retrying without violating the correctness. If the buggy operations exist in such code region, it seems that they can be bypassed since some compensations would be done later. We would like to explore the chance of applying our bypassing idea for the concurrency bugs into distributed systems in the future. Similarly, for concurrency bugs in fast path and slow paths [26], if we bypass all operations in fast path/slow path and force program to execute the other path, it can guarantee the correctness and help fix the bug. This is also an interesting direction.

In conclusion, both of our solutions try to tackle detected or undetected concurrency bugs. They can greatly improve software availability and reliability during the late phases of software life cycle. Before final patch is released, BugTM solution can serve as tentative patch for availability purpose. The diagnosis aspect of BugTM can complement bug detection tools, offering input for BFix and helping generate correct patch. Together with existing solutions, they can provide effective and efficient multi-phase protections for multi-threaded programs. Moreover, their performance-friendliness and simplicity make them attractive to be adopted by developers.
REFERENCES


[16] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott Mahlke, and David August. Encore: Low-cost, fine-grained transient fault recovery. In *MICRO ’11*.


100


