THE UNIVERSITY OF CHICAGO

DETECTING AND FIXING CONCURRENCY BUGS IN CLOUD SYSTEM

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
GUANGPU LI

CHICAGO, ILLINOIS
DECEMBER 2020

Copyright © 2020 by Guangpu Li

All Rights Reserved

To My Family

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES viii

ACKNOWLEDGMENTS ix

ABSTRACT x

1 INTRODUCTION 1
1.1 Distributed Timing Bug Fixing 1
1.1.1 Introduction 1
1.1.2 Examples 1
1.1.3 Challenges & Goals 3
1.1.4 Contributions 5
1.2 Salable Thread-safety-violation Detection 7
1.2.1 Introduction 7
1.3 Unsupervised Synchronization Inference 11
1.3.1 Introduction 11

2 AUTOMATE DISTRIBUTED CONCURRENCY BUG FIXING 16
2.1 Background 16
2.2 Fixing Message-Timing Bugs 18
2.2.1 Overview 18
2.2.2 Where does the Re-execution Start? 19
2.2.3 Where to Check Timing and Initiate Rollback? 20
2.2.4 How to Rollback 22
2.2.5 How to Observe the Buggy Timing 23
2.2.6 Patch Correctness Analysis 25
2.3 Fixing Fault-Timing Bugs 27
2.3.1 Overview 27
2.3.2 How to Observe the Buggy Timing 29
2.3.3 Fast-Forward Design 30
2.3.4 Rollback Design 32
2.3.5 Patch Correctness Analysis 34
2.4 Implementation Details & Limitations 35
2.5 Evaluation 37
2.5.1 Methodology 37
2.5.2 Overall Result 38
2.5.3 Patch details 41
2.5.4 More 43

3 EFFICIENT AND SCALABLE THREAD-SAFETY VIOLATION DETECTION 45
3.1 Motivation and Background 45
3.1.1 Why Concurrency Testing in the Large? 45
3.1.2 Why Thread-Safety Violations? 45
3.1.3 Why Not Happens-Before (HB) Analysis? 47
3.2 Algorithm 50
3.2.1 Overview 50
3.2.2 TSVDbase 53
3.2.3 DataCollider 53
3.2.4 TSVD 53
3.2.5 TSVD with happens-before analysis 59
3.3 Implementation 61
3.4 Evaluation 63
3.4.1 Methodology 63
3.4.2 Overall Results 64
3.4.3 Comparison with other detection techniques 67
3.4.4 Evaluating TSVD parameters 71
3.4.5 TSVD CPU/Memory Consumption 74
3.4.6 Examples of bugs found by TSVD 75
3.4.7 TSVD on Open Source Projects 76
3.4.8 More 76

4 UNSUPERVISED SYNCHRONIZATION-OPERATION INFERENCE 78
4.1 What are synchronization behaviors? 78
4.2 How to facilitate interesting behaviors? 80
4.3 SherLock 82
4.3.1 Observer 83
4.3.2 Solver 84
4.3.3 Perturber and Feedback across Runs 88
4.4 Evaluation 90
4.4.1 Methodology 90
4.4.2 Overall results 91
4.4.3 What synchronizations are inferred? 92
4.4.4 How helpful are inferred synchronizations? 95
4.4.5 What caused false positives and false negatives? 97
4.4.6 More detailed results 98

5 RELATED WORK 102
5.1 Related Work 102

6 FUTURE WORK 108

7 CONCLUSION 110

REFERENCES 111

LIST OF FIGURES

1.1 A message-timing bug in MapReduce[8]. 2
1.2 A fault-timing bug in HBase [7] (red explosion: bug-triggering crash; green explosion: tolerable crash). 3
1.3 Distributed message-timing bugs ((a) and (b)) and distributed fault-timing bugs ((c) and (d)). 6
1.4 A thread-safety violation (TSV) bug 8
1.5 The design space of active testing 9
1.6 SherLock workflow 14

2.1 DFix patch for a bug simplified from figure 1.1 (green part illustrates the patch; identities of A and B, including their bug-triggering causality stacks, are the inputs to DFix). 17
2.2 Pre-computation of race-object hash-code. 24
2.3 DFix fast-forward patch for the bug in figure 1.2 (this patch, in green, is essentially the same with developers' patch; identities of A_1, A_2, and B are the inputs to DFix). 24

3.1 Example of task parallelism in C#. TSVs can occur due to concurrent accesses of `dict`. 49
3.2 Happens-before graph for Figure 3.1, assuming neither `a` nor `b` is in the dict. The nodes show the executed line#; subscripts distinguish multiple executions of the same line. 49
3.3 The trap mechanism used by TSVD and its variants 51
3.4 Happens-before inference in TSVD (the thick arrows indicate inferred happens-before relationship) 56
3.5 TSVD instrumentation 62
3.6 Number of bugs found after more runs 69
3.7 Sensitivity analysis of various parameters of TSVD. 71
3.8 Examples 74

4.1 Acquire/release windows for synch. inference 81
4.2 Inferred synchronization examples 93
4.3 The numbers of correctly inferred unique synchronizations under different Perturber and feedback settings. 99

LIST OF TABLES

2.1 DFix benchmarks. Those 4/5-digit numbers are bug-IDs in bug databases. . . . 35
2.2 Overall results. (*: manual patching takes so long that the software has changed too much for performance comparison; $T_{correct}$: no sleep inserted, baseline is original software; T_{buggy}: sleep inserted to trigger the bug, baseline is manual patch; #: goes down with shorter sleep.) 39
2.3 DFix fault-timing patches. 42

3.1 Summary of bugs found by TSVD tools. 65
3.2 Comparing TSVD with other detection techniques. 68
3.3 Removing one technique at a time from TSVD 73
3.4 TSVD results on open source projects. 76

4.1 Applications in benchmarks 90
4.2 SherLock inferred results after 3 rounds. The sum in the parentheses is the unique synchronizations across applications. 91
4.3 SherLock vs. manual annotation in race detection 95
4.4 Breakdown of false positives/negatives. 97
4.5 Inference with or without certain hypothesis 98
4.6 Sensitivity of λ (numbers are the unique sums across 8 applications after running each test 3 times.) 99

ACKNOWLEDGMENTS

I want to thank many people during my phd life.

First of all, my advisor Prof. Shan Lu. From both technical and non-technical perspective, I learned a lot from Shan. The most important contribution in system research is the understanding instead of solution due to the complexity of both system and tool. Shan is the most patient person I have ever met. She was willing to teach me the obvious stuff many times if I cannot learn them ”immediately”. In addition, her consistent encouragement to resist research failure supports me to go through four-year of no paper life.

I want to thank two researcher in Microsoft, Madan Musuvathi and Suman Nath. I spent two summers with them and finished two thirds of my thesis. I really appreciate their help on the projects with their professional sense of software engine in industry. Their research taste and experience also encourage me to work on hard problem so that people will be interested in. I also want to thank Prof. Haryadi Gunawi for his advice of paper writing and handling paper rejection.

It is an amazing experience to spend five years in the computer science department in University of Chicago. I made many friends with the students here to have fun with. I really want to thank everyone in Shan's group. Sharing research difficulty with you guys relied me a lot in the past years. I also stayed in the same office five years with Prof. Haryadi's group. I want to thanks their funny conversation during daily life. I also want to the student outside these two groups: basketball friend Dixin Tang, fresh-year friend Brian Hempel. Moreover, I want to thank the stuff in our department for their generous help.

In the end, I want to thank my parent, Jingying Zhao and Shuicai Li , brother, Dongyang Li, and girlfriend, Qinpu He, for their encouragements and supports.

ABSTRACT

Distributed systems nowadays are the backbone of computing society, and are expected to have high availability. Unfortunately, distributed timing bugs, a type of bugs triggered by non-deterministic timing of messages and node crashes, widely exist. They lead to many production-run failures, and are difficult to reason about and patch. Although recently proposed techniques can automatically detect these bugs, how to automatically and correctly fix them still remains as an open problem. I designed DFix, a tool that automatically processes distributed timing bug reports, statically analyzes the buggy system, and produces patches. Our evaluation shows that DFix is effective in fixing real-world distributed timing bugs.

Concurrency bugs are hard to find, reproduce, and debug. They often escape rigorous in-house testing, but result in large-scale outages in production. Existing concurrencybug detection techniques unfortunately cannot be part of industry's integrated build and test environment due to some open challenges: how to handle code developed by thousands of engineering teams that uses a wide variety of synchronization mechanisms, how to report little/no false positives, and how to avoid excessive testing resource consumption. TSVD is a thread-safety violation detector that addresses these challenges through a new design point in the domain of active testing. Unlike previous techniques that inject delays randomly or employ expensive synchronization analysis, TSVD uses lightweight monitoring of the calling behaviors of thread-unsafe methods, not any synchronization operations, to dynamically identify bug suspects. It then injects corresponding delays to drive the program towards thread-unsafe behaviors, actively learns from its ability or inability to do so, and persists its learning from one test run to the next. TSVD is deployed and regularly used in Microsoft and it has already found over 1000 thread-safety violations from thousands of projects. It detects more bugs than state-of-the-art techniques, mostly with just one test run

Synchronizations are fundamental to the correctness and performance of concurrent

software. Unfortunately, correctly identifying all synchronizations has become extremely difficult in modern software systems due to the various types of synchronizations. Previous work either only infers specific type of synchronization by code analysis or relies on manual effect to annotate the synchronization. SherLock is a tool that uses unsupervised inference to identify synchronizations. SherLock leverages the fact that most synchronizations appear around the conflicting operations and form it into a linear system with a set of synchronization properties and hypotheses. To collect enough observations, SherLock runs the unit tests a small number of times with feedback-based delay injection. I applied SherLock on 8 C# open-source applications. Without any prior knowledge, SherLock inferred 122 unique synchronizations, with few false positives. These inferred synchronizations cover a wide variety of types, including lock operations, fork-join operations, asynchronous operations, framework synchronization, and custom synchronization.

CHAPTER 1

INTRODUCTION

1.1 Distributed Timing Bug Fixing

1.1.1 Introduction

Distributed systems such as scale-out storage systems [35, 46, 62, 127] and cloud computing frameworks [45, 123] are the backbone of our computing ecosystem. High availability of these systems is crucial, with minutes of outage costing millions of dollars [122, 159], but severely threatened by software bugs, particularly distributed timing bugs [67, 100, 169], a type of bugs triggered by non-deterministic timing of message communication or component failures (e.g., node crashes).

Distributed timing bugs impose a particularly large threat to system availability for several reasons. They are difficult to expose before code release, given their non-deterministic nature, and the limited scale and duration of in-house testing. Consequently, they widely exist in the field [21, 67, 69, 100] and easily manifest during large-scale and long-running production deployment, contributing to more than a quarter of cloud-system failures [169]. Although many recent tools [47, 68, 92, 98, 105, 107, 109, 152] can automatically detect these bugs, system availability does not improve until after these bugs are fixed. Unfortunately, correctly fixing distributed timing bugs is challenging for developers, as it involves global reasoning beyond one thread or one node, and often requires non-traditional synchronization, as we will elaborate below.

1.1.2 Examples

There are two types of distributed timing bugs: message-timing bugs and fault-timing bugs. Figure 1.1 illustrates a *message-timing bug* in MapReduce, triggered when a user kills a job

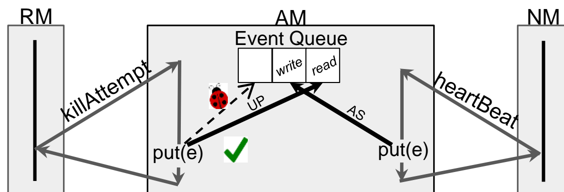


Figure 1.1: A message-timing bug in MapReduce[8].

unexpectedly early. From the figure, we can see two series of concurrent operations involved in this bug. Starting from the left-hand side of the figure, when a user issues a task-kill command, the resource-manager RM sends an RPC `killAttempt` to the application-manager AM; the RPC handler in AM then creates an event `UP` to asynchronously check the status of job J. Starting from the right-hand side of the figure, node-manager NM sends an RPC `heartBeat` to AM; the corresponding RPC handler in AM then creates an event `AS` to asynchronously update J's status to `ASSIGNED`. Since these two sets of operations are concurrent with each other, `UP` may non-deterministically read the job status *before* `AS` updates the status to `ASSIGNED`, causing a `JobNotExisting` exception and a job abort.

A naïve way to fix this bug is to use a single-machine synchronization primitive like condition-variable `wait` to force the status read in `UP` to wait for the status update in `AS`. However, this could cause safety and liveness problems. The blocking wait may cause an RPC *timeout* in the RPC client, potentially leading to failures. Furthermore, it may cause deadlocks by blocking not only the status read but also the status update, as these two accesses may share the same event-handling or RPC-handling thread — the number of event-queue or RPC-server handling threads is usually configurable and is 1 by default in many of our benchmarks.

Fixing this bug in a distributed manner is still challenging. First, reasoning about the timing relationship among distributed operations is error prone. For example, naïvely forcing RM to send `killAttempt` after NM sends `heartBeat` can*not* guarantee event `UP` to execute after event `AS` if the system configures multiple event or RPC handling threads. Second, it is not trivial to enforce a distributed execution order. It often involves adding either a

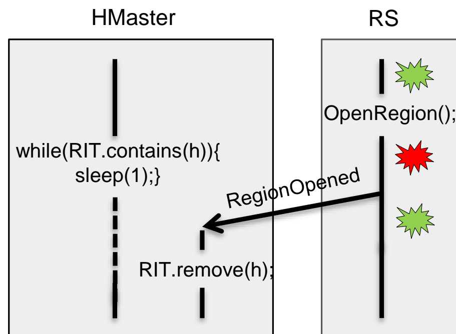


Figure 1.2: A fault-timing bug in HBase [7] (red explosion: bug-triggering crash; green explosion: tolerable crash).

distributed synchronization service like ZooKeeper [9] or a few new RPCs/messages among various nodes for coordination. Both may produce patches that are too complicated to adopt.

Figure 1.2 illustrates a *fault-timing bug* in HBase, triggered by an unexpected region-server (RS) node crash in the middle of opening a region. In HBase, a region server *RS* opens a region by executing an `OpenRegion` function, illustrated in the figure. This function first remotely causes a record to be inserted into HMaster's region-in-transition `RIT` map, indicating that a region is to be opened, and later remotely removes this record from `RIT` through a `RegionOpened` message, indicating that the region has been opened. If *RS* happens to crash in between, denoted by the red-explosion symbol in figure 1.2, HMaster would be stuck in a loop `while (RIT.contains (h))`, waiting for the record to be removed forever and making the whole HMaster service unavailable.

Clearly, fixing this bug cannot rely on traditional wait-based synchronization primitives like locks, condition variables, and semaphores — it is useless to wait as one cannot predict when a node would crash. Traditional file-system crash consistency does not help here either, as the impact of the crash goes beyond one node and beyond file systems.

1.1.3 Challenges & Goals

Those challenges discussed above widely apply:

- Fixing distributed timing bugs often cannot rely on traditional synchronization primitives like locks and condition variables, as blocking waits cannot help fix fault-timing bugs, and can introduce new bugs if put in event/RPC handlers where most message-timing bugs are located [100, 107].

- Fixing distributed timing bugs often requires global code changes and reasoning about operation timing and side effects across threads, processes, and nodes.

In addition to these unique challenges, there are also generic challenges like keeping patches not only correct, but also reasonably simple and well performing [154].

These challenges are reflected in how developers fix distributed timing bugs in practice.

Fixing distributed-timing bugs lacks dominant or systematic strategy in practice. According to the previous study [100], a variety of schemes, including ad-hoc ones, are used by developers: about half are fixed by a set of schemes like message retries, message ignoring, message faking (e.g., the bug in Figure 1.2), and others that allow software to better tolerate buggy timing; <30% are fixed by proactively disabling buggy timing, including <10% using distributed waits; about 20% are fixed through significant data-structure and semantic changes (e.g., the bug in Figure 1.1), partly due to the difficulty of finding semantic-preserving patches.

Fixing distributed-timing bugs is time consuming and error prone. Similar to single-machine timing bugs [65], distributed timing bugs take days to months to fix, with bug understanding and patch design all taking time. Even distributed timing bugs tagged with the highest priority [6, 10] suffer from incorrect patch releases after many days' patch design and review.

Although techniques have been proposed to help detect [68, 92, 98, 105, 107, 109, 152] and diagnose [40, 60, 88, 111, 118, 169, 173] distributed timing bugs, no techniques have been proposed to help automatically fix them. Auto-fixing has been proposed and heavily researched for local timing bugs where some use locks, condition variables, or well designed waits to fix

multi-threaded concurrency bugs after these bugs are detected [77, 80, 82, 84, 110, 116, 163], and some [16] use domain-specific event policies to fix event races in web applications. Unfortunately, these techniques cannot handle distributed timing bug challenges discussed above.

Overall, it is desirable to have techniques that automatically apply a unified strategy to fix many distributed-timing bugs through best-effort patches similar to what a human would create. Such a technique can save manual effort in fix-strategy design, global reasoning, global code changes, and code correctness review. The resulting patches could serve as either temporary patches, which improve system availability while helping developers to figure out final patches (as long as the automatically generated patches are in source code, which DFix patches are), or directly as final patches.

1.1.4 Contributions

This Chapter proposes DFix, a tool that takes in distributed timing bug reports, and automatically generates best-effort patches for many of them, through static program analysis.

At the high level, DFix does not use traditional blocking-wait synchronization primitives. Instead, it systematically generates patches that handle observed buggy timing through rollbacks [84, 130, 134, 151, 163] or fast-forwards.

Specifically, to fix a message-timing bug, DFix patches a selected code region r to observe if r's thread or node is executing undesirably fast, and if so, slow down by repeated roll-back and re-execution.

To fix a fault-timing bug, DFix patches a selected region r to observe whether another node has crashed at an undesirable moment and if so, rollback or fast-forward selected operations on behalf of the crashed node, pretending the crash occurred earlier or later.

At the low level, DFix uses static analysis to automatically decide where and how to observe buggy timing, and where and how to conduct rollback or fast-forward, so that the

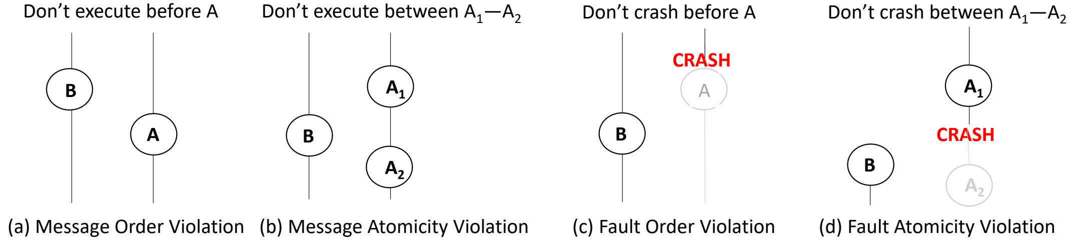


Figure 1.3: Distributed message-timing bugs ((a) and (b)) and distributed fault-timing bugs ((c) and (d)).

resulting patches satisfy several properties:

1. They are designed to only constrain the timing and not to change any computation logic of the original software execution. Specifically, the static analysis of DFix carefully avoids liveness violations like inserting blocking-waits inside RPC/event handlers, and safety violations like re-executing non-idempotent operations, etc.

2. They cover a wide variety of real-world fixing schemes through a unified high-level strategy. Depending on the rollback/fast-forward region location and length, some DFix patches are essentially equivalent with developer patches that proactively disable buggy timing, and some are equivalent with developer patches that reactively tolerate buggy timing (e.g., in the case of Figure 1.2); some involve message retries, and some essentially ignore or fake messages.

3. They often involve multiple threads and nodes. Through automated analysis and transformation, DFix relieves developers from distributed reasoning and code changes.

4. They do not introduce severe degradation to performance or code complexity. DFix intentionally gives up its bug fixing if the patch would be too complicated.

We evaluate DFix on *all* the 22 real-world distributed timing bugs reported by recently proposed message-timing bug detector DCatch [107] (10 bugs) and fault-timing bug detector FCatch [109] (12 bugs), which come from Cassandra, HBase, MapReduce, and ZooKeeper.

DFix automatically fixes 17 of them with similar performance and simplicity as manual patches. The source code of DFix and details about all the bug benchmarks, including patches generated by DFix and patches provided by developers, are all available at our website https://github.com/SpectrumLi/TimingBugFixing.

In summary, DFix is not a panacea. There are distributed timing bugs that DFix cannot fix. There are also bugs that DFix can fix but cannot fix in the most desirable way, particularly when the ideal patch requires semantics changes, which DFix has no conceivable way to automatically generate with correctness guarantees. However, we believe DFix provides a solid starting point towards solving this critical problem of patching real-world distributed timing bugs, which is very challenging for developers to manually reason about even *after* bug detection as we will see in DFix design, and improving availability of distributed systems.

1.2 Salable Thread-safety-violation Detection

1.2.1 Introduction

This Chapter specifically deals with a class of concurrency errors we call *thread-safety violations* or TSVs. Libraries and classes specify an informal *thread-safety* contract that determines when a client can and cannot concurrently call into the library/class. A TSV occurs when a client violates this contract. Figure 1.4 shows an example. The implementation of the `Dictionary` class allows multiple threads to concurrently call read operations, such as `ContainsKey`, but requires write operations, such as `Add`, to only be called in exclusive contexts. By violating this contract, Figure 1.4 contains a TSV.

This Chapter is motivated by the prevalence of such TSVs in production code. In fact, our evaluation discovered that the pattern in Figure 1.4 is fairly common as developers erroneously assume that concurrent accesses on different keys are “thread safe.” TSVs, while common, can have drastic consequences, including unexpected crashes or worse, silent data

```
// Dictionary dict
// Thread 1:
dict.Add(key1, value)
// Thread 2:
dict.ContainsKey(key2)
```

Figure 1.4: A thread-safety violation (TSV) bug

corruption that are difficult to diagnose and fix.

This Chapter describes TSVD, a dynamic-analysis tool for detecting TSVs. The tool is specifically designed to be used in an integrated build and test environment, such as Bazel [25] or CloudBuild [52]. Here, hundreds to thousands of machines build and run tests on software modules from thousands of engineering groups. We have designed the system end to end to operate at this scale. For instance, it is important to design tools that produce *little to no false bug reports*. A small loss of productivity per-user in chasing false bugs can quickly multiply to huge productivity losses across the organization. Similarly, it is important to design tools that incur *small resource overhead*. For instance, if a tool incurs a 10× run-time overhead or requires rerunning the same test 10× times, then the test environment needs to allocate 10× more machines to provide the same turnaround time to its users.

Since TSVs generalize the notion of data races to coarser-grain objects, techniques for detecting TSVs are naturally related to data-race detection (Section 3.1 has more discussion). But existing approaches fall short for our purposes.

Considering the goal about having little to no false bug reports, the approach of *active* delay-injection [50, 129, 147] is promising. This approach directs program threads towards making conflicting accesses by delaying threads at strategic locations at run time. Consequently, this approach only reports bugs that are validated at runtime, and thus are not false bugs.

Considering the goal about having small overhead however, existing active delay-injection techniques do *not* work. To better understand their limitations, we can broadly classify them based on the inherent tradeoff between the cost of injecting too many delays versus the

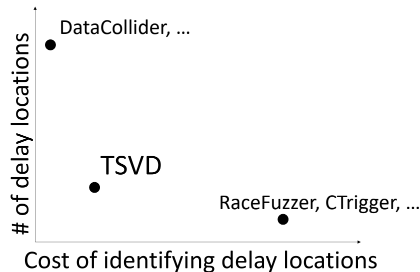


Figure 1.5: The design space of active testing

sophistication of analysis required to select where to inject delays:

At one extreme, as illustrated in the lower-right corner of Figure 1.5, tools like RaceFuzzer [147] and CTrigger [129] conduct static or dynamic analysis of memory accesses and synchronization operations in a program to identify potential buggy locations, and selectively inject delays only at these locations. While this reduces the number of delays injected, these techniques bear the cost of performing analysis, either through run-time overhead for dynamic analysis or the inscalability of precise static analysis required for identifying potentially conflicting accesses.

At the other extreme, as illustrated in the upper-left corner of Figure 1.5, tools like DataCollider [50] conduct little to no analysis, but instead probabilistically inject delays at many program locations. Such tools require many repeated executions of the same test to find a bug, which we deem unacceptable in our context as described above.

The goal of TSVD is to explore the middle ground in the design space of delay-injection tools, as illustrated in Figure 1.5. TSVD uses lightweight instrumentation and dynamic analysis to identify potential conflicting accesses without paying the overhead while being effective in finding bugs.

First, TSVD performs a lightweight *near-miss* tracking to identify potential thread-safety violations. It dynamically tracks threads that invoke conflicting methods to the same object close to each other in *real time*, without monitoring or analyzing synchronization operations.

Second, TSVD uses a *happens-before (HB) inferencing* technique that leverages the

feedback from delay injections to prune call pairs that are likely ordered by the happens-before relation [95] and thus cannot execute concurrently. The key observation is that if there is a HB relationship between two events a and b, then a delay in a will *cause* a delay in b. By inferring this causality, TSVD avoids injecting delays at these pruned pairs subsequently. Note, that unlike HB analysis performed by dynamic data-race detectors, HB inference does not require modeling, identifying, and analyzing synchronization operations in a program, which is expensive and complicated given the wide variety of synchronization mechanisms in production software.

An interesting benefit of the near-miss tracking and the HB inferencing is that it only requires *local* instrumentation. That is, only the libraries/classes whose thread-safety contract is checked need to be instrumented. All other parts of the code, including synchronizations such as forks, joins, locks, `volatile` accesses, and ad-hoc synchronization, do not need to be instrumented. This modularity dramatically reduces the runtime overhead. Furthermore, it enables incremental "pay-as-you-go" thread-safety checking of different classes and libraries, greatly simplifying the deployment of TSVD to thousands of engineering groups, who can configure the tool based on their needs.

Another key benefit of TSVD is that it is *independent* of the underlying concurrency programming model and *oblivious* to synchronization semantics, and thus can work across different programming models, such as threads, task-based programming, asynchronous programming, etc., and be applied to software using a variety of synchronization primitives.

Finally, TSVD compacts its delay injections, delay-location identification and adjustment all into the *same* run. In contrast, RaceFuzzer and CTrigger, require at least two runs, first to do the analysis and the second to inject delays. In many cases, as our evaluation shows, TSVD finds a large fraction of bugs in the first test run, making the best use of testing resources. This also allows TSVD to be effective where running many repetitions of the same test is not feasible due to large test setup and shutdown overheads.

We evaluate TSVD on tests from roughly 43,000 software modules developed by various product teams at Microsoft. By design, TSVD produces no false error reports, with each report containing stack traces of the two conflicting operations that violate the thread-safety contract. Overall, TSVD has found over 1,000 previously-unknown bugs, involving over 1,000 unique static program locations and over $20K$ unique call-stacks. To validate our results, we drilled into reports from four product teams in Microsoft. These groups confirmed that all the bugs reported are real bugs, 48% have been classified as high-priority bugs that would have otherwise resulted in service outages, and 70% have been fixed so far. Our evaluation also shows that TSVD introduces an acceptable overhead to the testing process — about 33% overhead for multi-threaded test cases while traditional techniques incur several times slowdowns.

TSVD is incorporated into the integrated build and test environment at Microsoft. Over last one year, 1,500+ software projects under active development at Microsoft have been tested with TSVD. An open-source version of TSVD is available at `https://github.com/microsoft/TSVD`.

1.3 Unsupervised Synchronization Inference

1.3.1 Introduction

Synchronization operations are fundamental to the correctness and performance of concurrent software. They determine which operations can and cannot execute concurrently. Many tools for analyzing concurrent programs, such as bug finding [31, 56, 74, 75, 89, 114, 119, 131, 145, 165, 172], bug fixing [81, 83, 85, 101, 117], performance profiling [2, 22, 40, 43, 103], and record-and-replay [140, 162] tools rely on understanding the semantics of program synchronizations. Typically, these tools rely on manually specification of this semantics. Incorrect and/or incomplete synchronization specifications can severely limit the effectiveness

of concurrency tools.

Unfortunately, correctly identifying *all* synchronizations in modern software systems is a challenging task. While common threading and locking primitives, such as those provided by `pthread` APIs, are easy to identify, programs can use various mechanisms to synchronize including data-parallel processing primitives, event-based asynchronous operations, shared-memory flag variables, language enforced semantics (e.g., finalizers only execute after an object is unreachable), system calls, and even server-side synchronization. Moreover, each form of coordination often has many variations: for example, C# standard `threading` library offers 5 lock classes and 9 signal-wait classes, with each containing many synchronization APIs and many sub-classes [5]. To complicate things further, programs and frameworks can roll their own or provide their own wrappers for underlying synchronization primitives. This complexity eliminates the hope of a once-for-all tedious manual annotations, which are likely to be error-prone anyway.

In this Chapter, we cast the synchronization inference as a *dynamic unsupervised* probabilistic inference problem. The basic idea is to dynamically observe various signals of synchronization behaviors during representative executions (say, during testing). While each signal could be noisy, the goal is to cumulatively combine these signals over multiple executions to predict synchronizations with high confidence. Being unsupervised has the crucial advantage that one needs no user-provided annotations, which we believe is essential for the general applicability of this technique.

This Chapter is inspired by prior work on probabilistic inference for security specifications [38, 112], which identifies source, sink, and sanitizers for vulnerability detection. In contrast to our work, these works use a semi-supervised approach that requires manual annotations to bootstrap their analysis. Also, they analyze the programs statically, while a key hypothesis of our work is that dynamic program behavior provides us a variety of signals to identify synchronization whose precision cannot be matched by those available statically.

This Chapter proposes SherLock, a tool for identifying program synchronizations. At high level, SherLock is based on the idea that all synchronizations are used to order events that would otherwise result in bugs. For instance, a data race occurs when two threads concurrently access the same variable with at least one of them being a write, which we refer to as concurrent conflicting accesses. Consider two conflicting accesses `a` and `b` in a dynamic execution as shown in Figure 4.1.a with `a` occurring before `b`. To prevent them from becoming a data race, programmers need to enforce a *happens-before* [96] relation between them by using a pair of synchronizations: a *release* synchronization after `a` and an *acquire* synchronization before `b`. The acquire synchronization blocks the execution of `b` until the release (and thus `a`) is complete or changes the control flow to prevent the execution of `b`. In this Chapter, we define *synchronization* as any operation or instruction that participates in forming a happens-before relation across threads.

Specifically, SherLock identifies synchronizations by relying on the following three insights.

Insight 1. We hypothesize that most (if not all) such conflicting accesses in mature programs are properly synchronized. Thus, if one considers an execution in Figure 4.1.a, it is highly likely that one of the operations in the *release window* that follows `a` is a release synchronization and one operation in the *acquire* window that precedes `b` is an acquire synchronization.

Insight 2. While we may not be able to precisely identify from a single execution which of the operations in the release (or acquire) window offers a release (or acquire, respectively) synchronization, we can do so by observing multiple executions and considering other characteristics of synchronization. Specifically, we design a set of properties and hypotheses that reflect fundamental assumptions of synchronizations and their behavior. They work together to enable us to pinpoint synchronizations. We discuss the details in Section 4.1.

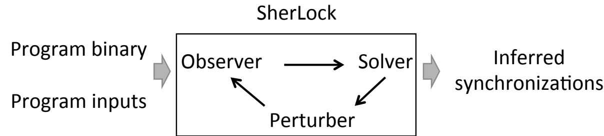


Figure 1.6: SherLock workflow

Insight 3. Effective inference depends on conflicting operations, such as the ones in Figure 4.1.a, being temporally close. Otherwise, large acquire/release windows will produce too many candidate operations. Rather than relying on the underlying scheduler to get lucky, we can actively perturb the execution at strategic locations to improve inference. We discuss the details in Section 4.2.

Guided by these insights, we have designed SherLock. As shown in Figure 1.6, given a program binary and its inputs, without any annotation, SherLock runs the program for all inputs and identifies a set of methods and variables whose entrances and/or exits, reads and/or writes are synchronizations, through three components working together:

1. Observer. This component instruments software binary and produces execution traces under the provided inputs. The goal of Observer is to make relevant observation of software behavior that can be compared against those properties and hypotheses of synchronization operations by the Solver. Its detailed implementation will be introduced in Section 4.3.1.

2. Solver. It processes all the observations collected so far. It identifies a set of acquire synchronizations and a set of release synchronizations, that all strictly satisfy those synchronization properties and altogether violate those hypotheses about synchronizations the least. This procedure is conducted by first encoding all the observations into linear constraints, representing must-be-satisfied properties, and objective functions, representing better-be-satisfied hypotheses, and then using a linear solver [1] to output every operation's probability of being an acquire or a release synchronization. The details are presented Section 4.3.2.

3. Perturber. To help the observer to make useful observations, without fully relying on

the random factors in concurrent program execution, the perturber injects delays at strategic points of the execution based on the Solver's earlier results. These delays will help observers collect key observations, which will then help the solver to draw more confident conclusions. The details will be presented in Section 4.2.

We applied SherLock on 8 C# open-source applications. In total, SherLock inferred 122 unique true synchronizations with few false positives. These include 1) standard synchronization primitives, such as monitors (`Monitor.Enter/Exit`), fork-join (`Task.Start/Wait`), and asynchronous tasks (`DataflowBlock.Post/Receive`); (2) variable-based synchronizations such as spin loops and flag variables; and, (3) application-specific methods that enforce happens-before relations by relying on underlying frameworks and language semantics (e.g. order between last-reference-removing instruction and the object dispose). A version of FastTrack [56, 57] that we built for C# applications detects 7× more true data races and 8× fewer false data races by using these inferred synchronization information than the default.

This Chapter makes the following contributions:

- Identifying a number of properties and hypotheses reflecting fundamental assumptions about synchronizations and how they are typically used in software, that work together to support effective synchronization inference.

- A feedback-based delay injection that actively exposes, instead of passively observes, run-time software behaviors that are particularly helpful for synchronization inference.

- An artifact SherLock that uses unsupervised inference to identify synchronizations with high coverage and accuracy.

CHAPTER 2

AUTOMATE DISTRIBUTED CONCURRENCY BUG FIXING

2.1 Background

Distributed Timing Bugs Previous study [100] categorizes real-world distributed timing bugs into two types, with each type containing two sub-categories, as illustrated in Figure 1.3. Before we present how DFix fixes these two major types in Section 2.2 and 2.3, we explain all the sub-categories below.

Message-timing bugs can be categorized into order violations and atomicity violations. A *message order violation* manifests when an operation B unexpectedly accesses a shared resource before an operation A, like reading data before initialization (figure 1.3a). A or B or both are initiated through messages. The bug shown in figure 1.1 belongs to this category.

A *message atomicity violation* manifests when an operation B unexpectedly executes in between and hence violates the atomicity of a code region A_1–A_2 (figure 1.3b). A_1, A_2, and B access the same resource, with at least one initiated by messages.

Fault-timing bugs can be similarly categorized.

In a *fault order violation*, operation B cannot proceed until A initiated by another node has executed. Unfortunately, when that other node crashes before A or drops a message whose handler conducts A, B waits forever and causes the system to partially or completely hang (figure 1.3c). The bug shown in figure 1.2 belongs to this category. In a *fault atomicity violation*, when a node unexpectedly crashes in a specific time window, denoted by A_1–A_2 in figure 1.3d, it leaves a system state that cannot be correctly handled by its recovery routine. Once that system state is read by the operation B in figure 1.3d, the recovery attempt or the whole system fails.

DFix Front End As an auto-fixing, *not* bug-detection, tool, DFix relies on its front end to provide accurate bug reports. Ideally, a bug report should contain (1) the bug type

```
Node AM: Thread 1
void handle (ASEvent e) {
  ...
  state = transition(state,ASSIGNED); //A
① +  DF_SET(...);
}
```

```
Node AM: Thread 2
void update (int id) {
  TA ta = TAmap.get(id);
  S state = ta.state;
② +  if (DF_CHECK(...))
  +      DF_ROLLBACK
  state = transition(state, UPDATE); //B
}
```

```
Node RM: Thread 1
③ + DF_ReEx_Start
  RPC (update, id);
  + DF_ReEx_End
```

Figure 2.1: DFix patch for a bug simplified from figure 1.1 (green part illustrates the patch; identities of A and B, including their bug-triggering causality stacks, are the inputs to DFix).

— which one out of the four types listed in figure 1.3; (2) racing instructions and buggy window boundaries — all the operations illustrated in figure 1.3, each identified by its static instruction ID and dynamic context (call stack and causality stack), under which the bug is triggered. DFix is *not* responsible for fixing unreported bugs and unreported contexts.

Here, *causality stack* refers to a sequence of causal operations that lead to the execution of I: when I is inside an event handler, the top of its causal stack is the corresponding event enqueue operation; when I is inside a message or RPC handler, the top of its causality stack is the message/RPC sending operation from another node; when I is inside a regular thread, the top is the thread creation from its parent. Causality stack is crucial to identify dynamic context related to bug manifestation [107, 109].

We expect DFix to work with most dynamic distributed-timing-bug detectors, as information like bug type and racing instructions are fundamental bug components and should be reported by most detectors. The dynamic context needed by DFix may not be reported by default, but should be easy to extract from any dynamic detector. The current DFix prototype uses DCatch [107] and FCatch [109] as front end for message-timing and fault-timing bugs, respectively. The bug report inputs used in DFix experiments can be found at https://github.com/SpectrumLi/TimingBugFixing.

2.2 Fixing Message-Timing Bugs

2.2.1 Overview

This section uses a simplified example from the real-world bug in figure 1.1 to explain how a DFix patch works and what are the key challenges in patch generation.

Running Example Figure 2.1 contains data races to a shared heap object `state` from the event handler `handle` and the RPC function `update`, denoted as A and B respectively. A job fails if B reads `state` before A updates `state`. Consequently, the patch should make sure A executes before B. Note that, B exists in an event handler instead of an RPC function in the real bug illustrated in figure 1.1. The simplification in figure 2.1 does not make the bug easy to fix: forcing RM to invoke RPC after NM (not shown in figure 2.1) still cannot guarantee B to execute after A due to the asynchronous nature of A; blocking-wait before B still incurs deadlocks and undesirable RPC timeouts. We will explain how to fix the original bug later in this section.

Roadmap All DFix patches for message-timing bugs, including both order violations and atomicity violations, contain similar components: somewhere before B like ② in figure 2.1, the patch calls `DF_CHECK` to see whether B's thread is executing too fast based on a flag set by `DF_SET` somewhere else like ① in figure 2.1, and, if so, the patch calls `DF_ROLLBACK` to roll back the execution to a place marked by `DF_ReEx_Start` like ③ in figure 2.1 and uses repeated re-execution from `DF_ReEx_Start` to slow down (the meaning of B is explained in Section 3.1).

To generate these patches, DFix takes several steps as sketched out in Algorithm 1:

First (line 2), DFix decides the re-execution region location by analyzing functions on the causality stack of B. The starting location of re-execution is particularly important, which is denoted as $\texttt{Loc}_{\text{Start}}$ in Algorithm 1 and exemplified by location ③ in figure 2.1. The details

of this step is explained in Section 2.2.2.

Second (line 5), DFix decides where to check the buggy-timing and potentially initiate rollback by analyzing functions on the causality stack of B and the $\texttt{Loc}_{\text{Start}}$ identified above. This location is denoted as $\texttt{Loc}_{\text{CHECK}}$ in Algorithm 1. It is sometimes right before the race instruction like ② in figure 2.1, but not always. The details are in Section 2.2.3.

Third (line 8), DFix generates code snippets that conduct the rollback by analyzing the locations of `DF_ReEx_Start` and `DF_CHECK` — whether they are inside the same function, whether they are inside the same node, and others. These code snippets are underneath `DF_ReEx_Start`, `DF_ReEx_End`, and `DF_ROLLBACK` located at ② of figure 2.1. The details are explained in Section 2.2.4.

Finally, DFix generates code snippets to allow its patch finding out the buggy timing at run time. Specifically, DFix decides where to set and unset a buggy-timing flag (line 11–15 in Algorithm 1), exemplified by the `DF_SET` located at ① in figure 2.1; DFix also generates the parameters for `DF_SET`, `DF_UNSET`, and `DF_CHECK` calls (line 18–21, 27–31). The details are presented in Section 2.2.5.

2.2.2 Where does the Re-execution Start?

The starting point of re-execution (i.e., the rollback destination) cannot be arbitrary. Particularly, it cannot be inside resource-holding or time-sensitive regions, like lock critical sections or event/RPC handlers, as repeated re-executions can cause deadlocks and/or timeouts. In fact, there may be no suitable rollback destination throughout the thread of B, because if B is in an RPC or event handler, everywhere in B thread is prone to deadlocks and/or timeouts.

Solutions Starting from the location right before B, DFixname searches backward for a suitable rollback destination along B's causality stack (i.e., *not* call stack like previous local concurrency-bug fixing tools [80, 82]). Given a location L, DFixname makes the following

check: (1) if L is inside a lock critical section, the search continues at where the corresponding lock was acquired; (2) if L is inside an event handler, the search continues at where the corresponding event was enqueued in another thread; (3) if L is inside an RPC or message handler, the search continues at where the RPC request was invoked at another node. If L does not fall into any of the three cases above, it is chosen as the starting point of re-execution (i.e., rollback destination), denoted as $\texttt{Loc}_{\text{Start}}$.

For the bug in figure 2.1, the search starts from inside the `update` RPC handler, right before B in the middle column of the figure. Since the rollback destination cannot be inside an RPC handler, the search moves to the RPC caller side on node RM, and ends at right before the RPC request on RM (i.e., ③ in figure 2.1).

2.2.3 Where to Check Timing and Initiate Rollback?

Somewhere before B and after the staring point of re-execution $\texttt{Loc}_{\text{Start}}$, the patch should check for the buggy timing and potentially initiate the rollback (i.e., invoking `DF_ROLLBACK` based on the `DF_CHECK` result in figure 2.1). There is a trade-off here about the `DF_CHECK` and hence `DF_ROLLBACK` location: on one hand, we prefer the checking to be closer to $\texttt{Loc}_{\texttt{Start}}$, as a shorter rollback/re-execution region is easier to guarantee correctness; on the other hand, we prefer the checking to be closer to B, as it is easier to predict whether B will execute shortly before B.

Solutions To balance this tradeoff, DFix starts from $\texttt{Loc}_{\texttt{Start}}$ identified earlier, and searches forward along B's causality chain towards B for a suitable location $\texttt{Loc}_{\text{CHECK}}$, so that $\texttt{Loc}_{\text{CHECK}}$ dominates B and the resulting region is the longest one that contains *no side-effect* instructions. This way, we can easily guarantee that repeated rollback and re-execution will not change program semantics without checkpoints. DFix gives up its

0. Inputs to this algorithm are provided by the front-end bug detector, and are explained in Section 3.1 and Figure 1.3. We use `PATCH X @ Y` to indicate that a DFix patch inserts code snippet `X` at location `Y`.

Algorithm 1: Patching a message-timing bug.

```
   Input  : {A, B} for an order violation,
            {A1, A2, B} for an atomicity violation
1  /*locate the re-execution region so that re-execution is not inside lock critical sections or RPC
    handlers */
2  {Loc_Start, Loc_End} = FindReexecutionRegion (B);
3
4  /*locate the check location so that re-execution is idempotent*/
5  Loc_CHECK = FindCheckLocation (Loc_Start, B);
6
7  /*generate rollback routine based on where re-execution region is*/
8  {DF_ROLLBACK, DF_ReEx_Start, DF_ReEx_End}= CausalityRollBack
    (Loc_Start, Loc_CHECK);
9
10 /*locate flag (un)set location to observe buggy timing*/
11 if (the bug is an Order-Violation) then
12     Loc_SET = After (A);
13 else
14     Loc_SET = Before (A1); Loc_UNS = After (A2);
15 end
16
17 /*pre-compute the race-location for DF_CHECK*/
18 RaceID_pre = Slicing (B.raceobj, Loc_CHECK);
19 if ( !Idempotent (RaceID_pre) ) then
20     RaceID_pre = Constant;
21 end
22
23 /* Generate the Patch */
24 PATCH (
25   DF_ReEx_Start                @ Loc_Start,
26   DF_ReEx_End                  @ Loc_End,
27   DF_SET (ID(A.raceobj))       @ Loc_SET, //order vio.
28   DF_SET (ID(A1.raceobj))      @ Loc_SET, //atom. vio.
29   DF_UNSET (ID(A2.raceobj))    @ Loc_UNS, //atom. vio.
30   "IF(DF_CHECK(RaceID_pre)) {DF_ROLLBACK}"
31                                @ Loc_CHECK);
```

patching attempt if $\texttt{Loc}_{\text{CHECK}}$ ends up in a different node from B.

Naturally, all read operations and writes to stack variables whose stack frames are rolled back (e.g., the write to `ta` and `state` in function `update` of figure 2.1) are free of side effects. An event enqueue, an RPC call, and a message sending are free of side effects if the corresponding handler only updates return values or stack variables during re-execution. This applies for RM's `RPC` request in figure 2.1, because the `update` function only updates stack variables before B.

2.2.4 How to Rollback

Since every rollback region in DFix patches is free of side effect, rollback and re-execution do not require any checkpointing and can be repeated for many times without introducing new bugs. We just need to pick different re-execution mechanisms for intra-node situations and inter-node situations, respectively.

To roll back a code region inside one thread or one handler, the patch puts the code into a loop and rolls back by a loop `continue` — such a loop and loop `continue` are put underneath `DF_ReEx_Start` and `DF_ReEx_End` surrounding the `RPC` call on RM in figure 2.1. Inter-procedural rollback is done similarly and uses exception throw to rollback from the callee function to the caller.

To roll back code regions across threads, handlers, or nodes, the rollback procedure is essentially multiple intra-thread/handler rollbacks chained together. Starting from the thread/handler where rollback is initiated (e.g., `DF_ROLLBACK` in figure 2.1), every thread-/handler simply terminates itself and then informs its causality parent (i.e., the parent thread, the message sender, the event creator, etc.) to also rollback until the rollback-destination thread/handler is reached, where a loop continues to launch re-execution (e.g., the loop underneath `DF_ReEx_Start` and `DF_ReEx_End` in figure 2.1).

How to inform the causality parent to rollback varies for different causality operations:

For RPC calls or other synchronous messages, this is done by throwing a remote exception/error. For example, in figure 2.1, `DF_ROLLBACK` throws an exception with the exception handler executing loop `continue` on RM as part of `DF_ReEx_End`.

For asynchronous causal operations like event enqueues, a flag variable is used to coordinate between the event handler and the event creator function C. After C enqueues the corresponding event, it repeatedly checks the flag until: (1) the flag is set by the event handler indicating that no buggy timing is observed and hence no rollback is needed, in which case C continues its execution; (2) the flag is set by the event handler indicating that buggy timing is observed and rollback is needed, in which case C rolls back itself and notifies its causality parent; or (3) the flag is not updated for a threshold amount of time and C is a time-sensitive code region like in an RPC handler. In this case, C rolls back and throws an exception to trigger an RPC resending. The patch makes sure that the re-sended RPC does not cause redundant event enqueues. This is how the original bug in figure 1.1 is fixed in a semantics-preserving way.

2.2.5 How to Observe the Buggy Timing

At the checking site identified earlier, the DFix patch checks a flag to decide whether the current thread is executing too fast. In figure 2.1, this checking is done by `DF_CHECK` function and the function parameter helps decide which flag to check.

The challenge here is to maintain a precise flag, particularly (1) when to set and unset the flag, and (2) how to distinguish flags that belong to different dynamic instances of one static bug. The latter was ignored by local concurrency-bug fixing tools [80, 82], but is particularly important in cloud systems, where the same instruction could execute for many times (e.g., once per task attempt or job) and ordering is expected only among certain dynamic instances (e.g., the status read of job J needs to wait for initialization of J only, not other jobs).

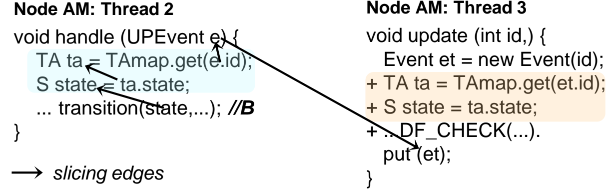


Figure 2.2: Pre-computation of race-object hash-code.

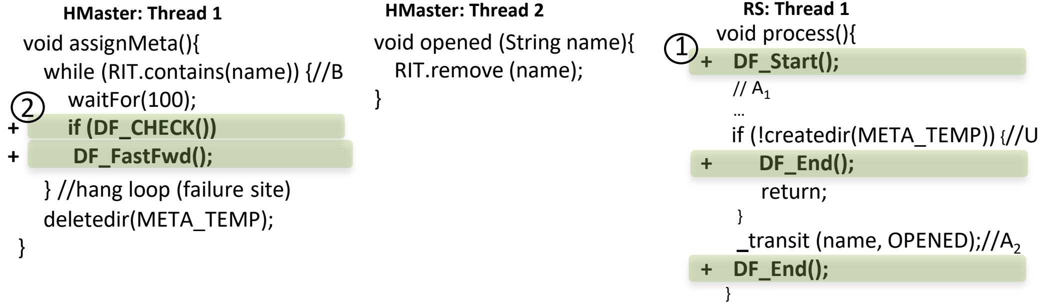


Figure 2.3: DFix fast-forward patch for the bug in figure 1.2 (this patch, in green, is essentially the same with developers' patch; identities of A_1, A_2, and B are the inputs to DFix).

Solutions For an order violation $\{A, B\}$, a flag is set right after A, and B can execute once the flag is set (line 12 in Algorithm 1). For an atomicity violation $\{A_1\text{–}A_2, B\}$, a flag is set right before A_1 and reset right after A_2, and B can execute if the flag is not set (line 14 in Algorithm 1).

Since different dynamic instances of A or B could access different objects, DFix creates a flag map for each bug, indexed by the hash-code of the race object m accessed by racing instructions. `DF_CHECK` and `DF_SET` in figure 2.1 use such hash code as parameters, which help locate the right flag.

The remaining question is to figure out which object will be accessed by the buggy instruction B at the checking site. This is trivial if `DF_CHECK` is right before B like that in figure 2.1, but challenging otherwise like that in figure 2.2.

DFix first statically identifies all instructions that may affect the race-object identity along any path between `DF_CHECK` and B. To do so, DFix extends traditional static slicing by concatenating slices in multiple threads or handlers together along the causality chain (line 18 in Algorithm 1). DFix then puts these instructions together, with variable names

adjusted (e.g., replace `this.f1` by `o.f1`) or replaced by corresponding getter method (e.g., replace `this.f1` by `o.getf1()`), to get the race-object reference for `DF_CHECK`. Finally, DFix statically checks whether the above pre-computation is idempotent. If not, DFix simply uses a constant number as the flag-map key, making all dynamic instances of a bug share the same flag (line 19–21 in Algorithm 1).

For example, in figure 2.2, DFix wants to pre-compute the race-object identity, which is `state` used by B in Thread 2, right after the event-enqueue in Thread 3. To accomplish this, static slicing starts from B and identifies two operations that determine the identity of `state` in the `handle` function, as indicated by the arrows in figure 2.2. Then, following the causality chain, DFix knows that the `handle`'s parameter `e` is actually the local variable `et` inside the `update` function in Thread 3. Then, these related instructions are cloned to function `update` with some name adjustment (i.e., replacing `e` by `et`) to pre-compute the race-object identity.

The above pre-computation is accurate when other threads cannot affect which object B would access between `DF_CHECK` and B (i.e., no time-of-check time-of-use races [26, 28]).

DFix proves this assumption when (1) B accesses a global/static object; or (2) the pre-computation only reads variables whose content cannot be changed by other threads between the checking and B, like local variables, final variables, heap objects whose references haven't been put to another shared objects yet. etc. When DFix cannot prove so, it still generates the patch. In the very rare cases when a TOCTOU race happens right before a buggy timing occurs, the patch would fail to fix the bug but would not introduce new bugs.

2.2.6 Patch Correctness Analysis

DFix carefully prevents its patches from introducing new bugs through several efforts. First, DFix makes sure that the rollback/re-execution region in its patch does not contain any side-effect operations (Section 2.2.3), and the extra pre-computation conducted to observe the buggy timing is idempotent (Section 2.2.5). This way, DFix patches guarantee to only

change the execution timing, but not introducing new program semantics. Second, a DFix patch does not introduce any blocking wait inside RPC handlers, given its overall design and its safety-net timeout (Section 2.2.4), and hence does not introduce any unexpected RPC timeout exceptions. Third, DFix sets a threshold for the number of rollbacks in its patch (20 by default). This way, a DFix patch is guaranteed not to cause deadlocks, although in theory it may fail to prevent a bug manifestation that could have been prevented if there were more rollbacks.

We further elaborate on the last point. Naïvely, one may want to delay B until A is executed. However, doing so may cause deadlocks if A never executes, a realistic issue in distributed systems where a node crash easily makes an instruction that was supposed to execute disappear. This is why DFix needs to set a threshold for the number of rollbacks.

Furthermore, although unnecessary for patch-correctness guarantees, DFix still makes its best effort in analyzing whe-

ther the waited-for instruction is guaranteed to execute, if there is no node crashes or message drops, to provide more information to developers. To do so, DFix leverages the traditional dominating and post-dominating relationship analysis [19] and also extends it with causality information to reason beyond one thread or one node — if a causality child like an RPC handler has executed, we know for sure that the causality parent like the RPC caller has executed. This way, DFix can ensure that the waited-for instruction is guaranteed to execute for most message-timing bugs that we encountered in fault-free situation (Section 2.5.3).

2.3 Fixing Fault-Timing Bugs

2.3.1 Overview

We use the bug discussed in figure 1.2 to demonstrate how DFix fixes a fault-timing bug through fast-forward.

Running example As shown in figure 2.3, HMaster's thread 1 executes a loop that exits only when a specific entry `name` is removed from the `RIT` map. This entry is inserted through a request by RS right before the `process` function, and is removed through an `OPENED` handler in HMaster-Thread 2 also remotely requested by RS at A_2. When RS crashes inside this `process` function before the `OPENED` request is sent out at A_2, HMaster gets stuck in the while loop, making the whole HBase system unavailable.

Roadmap All DFix patches for fault-timing bugs contain similar components: right before where the failure gets exposed according to the bug report like ② in figure 2.3, the patch checks whether another node N_{fault} has crashed in the buggy time window. If so, this checking node attempts to either rollback (in rollback patch) or fast-forward (in fast-forward patch like figure 2.3) some of the crash-persistent operations on behalf of N_{fault} (i.e., inside `DF_FastFwd` in figure 2.3); logging surrounding the buggy time window (i.e., inside `DF_Start` and `DF_End` in figure 2.3) is used to support the above checking and rollback or fast-forward.

To generate these patches, DFix takes several steps as sketched out in Algorithm 2:

First (line 2–4 in Algorithm 2), DFix decides where to insert logging code to allow its patch checking whether and where node N_{fault} has crashed inside the bug-timing window, by analyzing every path connecting A_1 and A_2. In the example shown in figure 2.3, the locations of `DF_Start` and `DF_End` are decided at this step. The details are presented in Section 2.3.2.

Next, DFix checks whether a fast-forward patch is suitable for the bug (line 7 in Algorithm 2). If so, DFix identifies all the crash-persistent operations that potentially need fast-forward,

Algorithm 2: Patching a fault-timing bug.

```
   Input  : A_1, A_2, B
1    /*locate the start and end of buggy window */
2    Loc_Start = Before (A_1);
3    Loc_End = CFGExitNodes (A_1, A_2);
4    OP = CrashPersistentOperations (A_1, A_2);
5
6    /*Identify fastforward operations and their parameters*/
7    if (SafeFastFwd(A_1, A_2)) then
8        OP_FF = RequireFastFwd (OP);
9        Params_OP_FF = Slicing (OP_FF, A_1);
10
11       /* Generate Fast-Forward Patch */
12       PATCH (
13         Log(START, Params_OP_FF)  @ Loc_Start,
14         Log(END)                  @ Loc_End,
15         Log(OpID)                 @ After(OP_FF),
16         "IF(DF_CHECK()) {DF_FastFwd();}"
17                                   @ Before(B);
18   end
19
20   /*Identify rollback operations */
21   if (SafeRollback(A_1, A_2)) then
22       OP_RB = RequireRollBack (OP);
23
24       /* Generate Roll-Back Patch */
25       PATCH (
26         Log(START)               @ Loc_Start,
27         Log(END)                 @ Loc_End,
28         Log(OpID, CheckPoint) @ Before(OP_RB),
29         "IF(DF_CHECK()) {DF_RollBack()};"
30                                  @ Before(B);
31   end
```

denoted as $\mathbb{OP}_{\text{FF}}$ in Algorithm 2, decides how to pre-compute all the parameters needed by them before entering the buggy window (line 9), and generates the patch (line 12–17). The details of this step is presented in Section 2.3.3.

Finally, DFix checks whether a rollback patch is suitable for the bug (line 21). If so, it identifies all the crash-persistent operations that potentially need rollback, denoted as $\mathbb{OP}_{\text{RB}}$ in Algorithm 2, inserts a content-checkpoint function right before each of these operations (line 28), and generates the remainder of the patch similar to that of a fast-forward patch (line 26–30). The details are presented in Section 2.3.4.

Note that, unlike generic transactional rollback or fast-forward design, DFix puts much attention to patch simplicity — if too complicated, the patch will probably not be accepted in practice anyway. As a result, although DFix attempts to generate a rollback patch and a fast-forward patch for every fault-timing bug, either or both attempts may fail as simple rollback/fast-forward patches do not exist for every bug.

2.3.2 How to Observe the Buggy Timing

To allow a DFix patch checking whether and where node N_{fault} has crashed inside the bug window A_1 – A_2, logging code is inserted to mark the beginning and the end of the bug window, and all operations inside the window that might require rollback or fast-forward.

Specifically, DFix statically analyzes every path connecting A_1 to A_2 on the control flow graph (CFG), and conducts the following checking for every operation I along the path.

First, if I has a successor I' that cannot reach A_2, DFix logs "END" right before I' to indicate the end of the bug-triggering fault window (line 14 and 27 in Algorithm 2) — that is how the two `DF_End` locations are decided in Figure 2.3.

Second, if I has side-effect beyond its local node N_{fault}, including global file-system updates and message/RPC sending, DFix adds logging at I unless proved to be unnecessary later in Section 2.3.3 and 2.3.4. When I is a file operation, DFix places the logging right

before and after the operation for rollback (line 28 in Algorithm 2) and fast-forward (line 15 in Algorithm 2) patches, respectively. This way, even if N_{fault} crashes between I and the logging, we can still guarantee correct patch semantics. Every log entry is appended with node and process ID, source file and line number. In figure 2.3, the two operations tagged with U and A_2 respectively are identified as having global side effects. Neither of them needs logging for different reasons: DFix decides U needs no fast-forward, which will be explained in Section 2.3.3, and hence does not insert logging for it; the logging for A_2 is combined with the end-of-buggy-window logging inside `DF_End`.

DFix currently only supports A_1 and A_2 coming from the same thread. We discuss how to obtain them for both sub-types of fault-timing bugs in Section 3.3.

2.3.3 Fast-Forward Design

At high level, a DFix fast-forward patch makes system updates that N_{fault} planned to but did not make due to its crash, which we refer to as *operation fast-forward*, so that the system can proceed as if N_{fault} crashed after the bug-triggering window.

Now that the DFix patch can decide which crash-persistent operations have yet executed (Section 2.3.2), we discuss below (1) how to identify operations that do not require fast-forward — the details behind the `RequireFastFwd(`$\mathbb{OP}$`)` on line 8 of Algorithm 2; (2) how to get parameters of a to-be-fast-forward operation with its execution context wiped out by the crash — the details behind the $\texttt{Params}_{\mathbb{OP}_{\text{FF}}}$ on line 9 of Algorithm 2; (3) how to execute the operation *on behalf of* another node (i.e., N_{fault}) — the details behind the `DF_FastFwd` function in Algorithm 2 and Figure 2.3.

Note that, DFix identifies and gives up its fast-forward patching in either of these two cases, which is represented by the `SafeFastFwd` function in line 7 of Algorithm 2. First, there is a thread-waiting (e.g., a condition variable wait) between A_1 and A_2. In this case, fast-forwarding any operation after the wait may require fast-forwarding the waited-for thread

too, which greatly complicates the patch. Second, whether a may-fast-forward operation will be executed and what are its parameter values cannot be decided *before* entering the buggy time window at runtime. We will elaborate on this case later.

Identify Operations That Need Fast-forward The crash-persistent operations that N_{fault} did not execute did not all require fast-forward. Specifically, DFix skips two types of operations I. First, I is post-dominated by another operation I' inside the buggy time window and they write to the same file, without any file read, synchronization, or communication operation in between. Second, a similar I' like the first case exists in code regions that post-dominate the failure site. For example, in figure 2.3, DFix static analysis identifies a `deletedir` operation that post-dominates the failure site and updates the same resource as `createdir`, which makes `createdir` unnecessary to fast forward[1]. Consequently, the `DF_FastFwd` function in figure 2.3 executes the other crash-persistent operation inside the buggy window — the `_transit` function tagged as A_2.

Fast-forward Parameter Preparation A DFix patch needs to pre-evaluate and record all the parameter values of all operations that may need to be fast-forwarded by another node before entering the buggy time window. To do so, DFix uses static slicing to identify all the instructions used to compute a parameter value, and clone these instructions to right before the buggy time window, just like how DFix pre-computes race-object identity in Section 2.2.5. Also like that in Section 2.2.5, DFix gives up its fast-forward patching if the pre-computation is not idempotent or relies on shared variables that might be updated by other threads during the buggy time window. In practice, file-update and message-sending parameters, like file paths, IP addresses, port numbers, and file/message content, are often constant or determined by thread-local variables, and hence can often pass the checking of DFix. At runtime, N_{fault} may crash right before or in between the pre-computation. since the pre-computation occurs

1. DFix has built-in knowledge about Java.io.File library.

right outside the buggy time window, such a node crash will not trigger any bugs.

Take the bug in figure 2.3 as an example. `_transit` has two parameters: `name` is a local variable and `OPENED` is a constant. In fact, `name` is updated to be `this.region.toString()` inside the `process` function. DFix identifies this assignment, and makes its patch pre-evaluates and records the content of `this.region.toString()` right before the buggy window, inside `DF_Start`. Here, `region` is a `final` field and hence its value is guaranteed not to change.

These pre-computed parameter values, as well as their type information, are logged to a file. DFix uses protocol buffer [12] for non-primitive parameters. When another node attempts fast-forward, it retrieves parameter information from the file and then conducts fast-forward. In figure 2.3, this is inside `DF_FastFwd` in the HMaster node.

Operation Fast-forward Finally, executing an operation on behalf of a different node is straight-forward for file updates, but requires API-specific handling for inter-node communication. The _transit function in figure 2.3 is a ZooKeeper API that any node can execute and achieve the same effect — certain znode is updated in ZooKeeper server with a particular value, which then causes ZooKeeper to send out corresponding messages. For socket message and RPC request sending, DFix pre-evaluates and records the destination IP address, port number, and message content in advance. We skip the detailed implementation for space constraints[2].

2.3.4 Rollback Design

At the high level, DFix's rollback patch follows the traditional transaction undo-log approach [66, 71]. When N_{fault} executes inside the buggy window, right before every crash-persistent write, the DFix patch records the original content and the update location. Later on, if N_{fault} crashes inside the buggy window, the DFix patch rolls back N_{fault}, pretending that

2. Like previous tracing [118] and bug detectors [107, 109] for distributed systems, DFix requires built-in knowledge about common library interfaces.

N_{fault} crashed before the buggy window.

The main challenge is to balance patch simplicity and capability. At one extreme, to roll back arbitrary code, the patch could be very complicated in supporting coordinated rollback across many threads or nodes [34]. At the other extreme, existing single-node rollback techniques are simple, but too limited for many fault-timing bugs. For example, the rollback used in Section 2.2.4 to fix message-timing bugs and existing transactional memory techniques [71] do not support file or message rollbacks, and hence cannot help roll back crash-persistent operations. Existing file-system transaction techniques [11] do not handle message rollbacks; it is also impractical to change file-system in one bug patch.

Solutions DFix chooses a design that balances simplicity and capability: (1) it supports multi-thread and multi-node rollback, but is limited to the thread of A_1–A_2 on N_{fault} and the thread of the failure site where the crash is observed and rollback is launched, which we denote as B on N_B like HMaster-Thread 1 in figure 2.3; (2) it supports rollback of some common, but not all, heap accesses, file accesses, and messages.

Given a fault-timing bug, DFix first decides whether its rollback can fix this bug by statically analyzing every path between A_1 and A_2. DFix gives up its rollback patching once it identifies (1) a thread-signal, a thread-creation, or an event enqueue; or (2) an inter-node communication whose recipient handler cannot be statically identified; or (3) a recipient handler with side effects beyond the thread of B, A_1, and A_2. This is how the `SafeRollback` function on line 21 of Algorithm 2 works.

During the above static checking, DFix also identifies all operations that may require rollbacks — file updates and N_B's heap updates. For simplicity and performance concerns, DFix further prunes out operations that do not require rollback: (1) any write w dominated by write w' that updates the same object or file in the buggy window; (2) any file update dominated by the creation of its parent directory in the buggy window. This is how $\mathbb{OP}_{\mathrm{RB}}$ is computed on line 22 of Algorithm 2.

After a bug passes the above checking and has all the may-rollback operations identified, DFix inserts content recording before every such operation (line 28 in Algorithm 2). For a heap update, an object clone to a shadow object is inserted. For a file update, the current prototype of DFix simply copies the whole to-be-updated file into a shadow file, as well as records the original file name. This simple scheme could lead to performance problems for large files, and our future work will improve this part by recording only the updated file content or leveraging copy-on-write mechanisms like `reflink` provided by some file systems.

One issue of this implementation is that a file update, if it is towards a global file, may already be read by another node before the rollback. This is usually not a problem, because if another node could read the inconsistent file image, another fault-timing bug would have been reported. Our future implementation can also try buffering global-file updates.

2.3.5 Patch Correctness Analysis

DFix relies on several assumptions to work correctly. First, the input provided by front-end bug detectors is correct. If this assumption does not hold, fast-forward or rollback may be conducted while node N_{fault} is still alive, which could lead to software misbehaviors. This bug-detection problem goes beyond the scope of DFix bug fixing. Second, the target software uses standard APIs for thread creation/join, signal/wait, and other synchronization operations. As discussed in Section 2.3.3 and 2.3.4, DFix may give up its patching when certain types of synchronization operations exist in the buggy time window. Automatically identifying arbitrary custom synchronization goes beyond the scope of DFix.

Under these assumptions, the DFix patch can correctly judge whether a fault-timing bug has occurred and make a best-effort fault handling like what developers' patches usually do. The logging design in Section 2.3.2 makes sure that the patch can correctly judge which operations require fast-forward or rollback, and the fast-forward and rollback design in Section 2.3.3 and 2.3.4 make sure that the patch does not introduce any new semantics and only

Table 2.1: DFix benchmarks. Those 4/5-digit numbers are bug-IDs in bug databases.

ID	App-BugID	LoC	Manual Fixing Time
Root Cause: Message Atomicity Violations			
AM1	CA-1011	61K	114 days
AM2	HB-4539	188K	5 days
AM3	HB-4729	213K	27 days
Root Cause: Message Order Violations			
OM1	CA-extra1	61K	182 days
OM2	MR-3274	1.2M	4 days
OM3	MR-4637	1.3M	48 days
OM4	ZK-1144	102K	8 days
OM5	ZK-1270	110K	7 days
OM6	ZK-1194	110K	45 days
Root Cause: Fault Atomicity Violations			
AF1	HB-2611	137K	998 days
AF1	HB-3596	137K	12 days
AF2	HB-12241	137K	918 days
AF3	ZK-1653	67K	272 days
Root Cause: Fault Order Violations			
OF1	CA-5393	159K	22 days
OF2	CA-6415	159K	5 days
OF3	CA-extra2	159K	91 days
OF4	HB-10090	536K	5 days

makes the system behave as if the crash occurred later or earlier. There is also no risk of deadlocks, as the DFix patch does not conduct any blocking waits. DFix static analysis is conservative, and would give up chances for any optimization, like combining two updates to the same resource during fast-forward or rollback, if correctness cannot be guaranteed.

2.4 Implementation Details & Limitations

DFix is implemented using WALA Java byte code analysis framework v1.3.5 [78] and JavaParser v3.2.4 for a total of around 5000 lines of code.

Bug Report Pre-processing Our current prototype uses bug reports automatically generated by a message-timing bug detector DCatch [107] and a fault-timing bug detector FCatch [109]. They are both dynamic bug detectors, and hence easily provide all the needed call stack and causality chain information. They also provide bug-triggering support, so that

users can confirm whether a bug can truly cause severe failures before bug fixing.

The main information that is missing and has to be filled up by DFix is the buggy time window for fault order violations (figure 1.3c). FCatch reports A and B for this type of bug. A is clearly the end of the buggy fault time window. However, in practice, there is often a start point of the window — the bug is often not triggered if the node crashed too early. For example, the bug in figure 1.2 is a fault order violation, but the bug does not manifest if RS crashes earlier than `OpenRegion`. The current prototype of DFix uses the fault injection support provided by FCatch to figure out what is the start of the time window before B in the thread of B.

Library Specifications DFix identifies RPCs, message sending operations, event related functions, and synchronization following the related library interfaces, like `VersionProto` [14], `ProtoBase` [15], Hadoop `GenericEventHandler` [13], Zoo-Keeper [9], similar to previous work that dynamically analyzes and detects bugs in distributed systems [107, 109, 118]. DFix also contains built-in knowledge about Java basic file-operation APIs. A system that uses a different RPC, message, or event library would require customizing DFix with a new interface specification, which is easy to provide as also suggested by previous work [107, 109, 118].

Generate Source-code Patches DFix static analysis is conducted at WALA intermediate representation level, which WALA transfers from source code. At this level, all the object IDs are replaced by virtual register IDs. To generate source-code level patches for program developers to review, we translate the intermediate code back to source code leveraging a virtual-register-to-object map maintained by WALA.

Causality Clone Like previous tools that patch single-machine concurrency bugs [80, 82], DFix clones bug-related functions and applies its patch only in cloned functions, so that the patch only takes effect under the causality and calling context indicated by the bug report.

Imagine the bug report indicates that a racing instruction is inside an RPC function `foo` remotely invoked by function `bar`. DFix would clone `foo` into a new function `foo_DCFix`, and add patching code in `foo_DCFix` but not `foo`. Similarly, DFix also clones `bar` into `bar_DCFix`, and changes `bar_DCFix` to invoke `foo_DCFix`. Since this clone technique is very similar to that in previous work [80], we skip the details due to space constraints.

Limitations of DFix DFix does not aim to fix all bugs. In fact, semantics-preserving simple patches do not exist for many bugs. As discussed earlier, DFix gives up on a message-timing bug, if the location that DFix chooses to initiate rollback, $\texttt{Loc}_{\text{CHECK}}$, ends up on a different node from B; DFix gives up on fixing a fault-timing bug, if (1) the bug-triggering fault window boundaries cannot be identified or cannot be located in one thread and (2) the rollback/fast-forward may involve more than one thread on the crashed node. DFix also gives up on using fast-forward to fix a fault-timing bug, if it cannot correctly pre-compute the context of the to-be-fast-forward operations; DFix gives up on using rollback to fix a fault-timing bug, if the rollback goes beyond the faulty node and the system-failure node. Furthermore, DFix fixes bugs that are triggered under specific call stack and causality stack reported by the front-end detector. DFix cannot fix bugs that are not reported.

2.5 Evaluation

2.5.1 Methodology

Benchmarks To evaluate DFix, we try *all* the 22 harmful bugs reported by front-end bug detectors in their papers [107, 109][3]. These include 10 message-timing bugs by DCatch [107], with 3 atomicity violations and 7 order violations, and 12 fault-timing bugs by FCatch [109], with 7 atomicity violations and 5 order violations. These are real-world bugs in 4 widely used

3. The original papers showed more, but some bugs are fixed once some other bugs are fixed and hence we did not double count.

systems: Cassandra key-value stores (CA), HBase key-value stores (HB), Hadoop Mapreduce (MR), and ZooKeeper metadata management service (ZK).

DFix successfully fixes 17 out of these 22 benchmark bugs. The remaining 5 include 4 fault-timing bugs whose bug-triggering time window A_1–A_2 goes beyond one thread and hence cannot be handled by DFix; and one message-timing bug that cannot be reliably triggered and hence is dropped from our benchmark suite.

Table 2.1 provides details for these 17 bugs. Clearly, developers took a long time to fix them[4], with only 4 of them fixed in < 1 week. Note that, these bugs took long time to fix **not** because they are considered not critical. Among them, 7 are tagged as top 10% important bugs by developers (i.e., 4 “blocker”s and 3 “critical”s), 7 are “major”, 1 is “minor”. The remaining 2 did not appear in bug-reporting systems.

Setup We use two machines, with Intel i7-3770 CPU, 8GB RAM, Ubuntu 14.04, and JVM v1.7. We evaluate whether DFix can fix a bug, as well as the performance and simplicity of the resulting patches. We also compare DFix with manual patches and alternative naïve patching strategies.

2.5.2 Overall Result

Functionality As shown in table 2.2, DFix successfully fixes all 17 bugs listed in table 2.1; naïve patch strategies like adding locks around racing operations and restarting the crashed node after local-file cleanups can only fix 2 benchmarks, OM1 and AF4 (they also cannot fix any of the 5 not in table 2.1 and 2.2 that DFix cannot fix); there is one benchmark, AF2, that even the final manual patch did not completely fix — the manual patch only added sleep to lower the failure probability.

We make our best effort in evaluating patch correctness through stress testing, code

4. The time listed in Table 2.1 includes both bug-understanding time and patch-design time. We can only see that both are very time consuming, but cannot tell exactly how much time went to each from bug reports.

Table 2.2: Overall results. (*: manual patching takes so long that the software has changed too much for performance comparison; $T_{correct}$: no sleep inserted, baseline is original software; T_{buggy}: sleep inserted to trigger the bug, baseline is manual patch; #: goes down with shorter sleep.)

	Is the bug fixed?			Overhead (%)		
				$T_{correct}$		T_{buggy}
ID	Manual	DFix	Naive	Manual	DFix	DFix
AM1	✓	✓	$\times_{hang}$	*	0.5	*
AM2	✓	✓	×	0.7	0.2	-0.1
AM3	✓	✓	×	2.3	2.4	-1.6
OM1	✓	✓	✓	*	0.4	*
OM2	✓	✓	$\times_{hang}$	0.2	0.4	$21.2^{\#}$
OM3	✓	✓	$\times_{hang}$	0.2	0.7	$23.3^{\#}$
OM4	✓	✓	$\times_{hang}$	-1.2	0	0.3
OM5	✓	✓	$\times_{hang}$	-2.1	0	0.2
OM6	✓	✓	$\times_{hang}$	-0.5	0.8	-0.1
AF1	✓	✓	×	*	-4.2	*
AF2	✓✗	✓	×	13.0	0.1	-0.4
AF3	✓	✓	×	*	1.6	*
AF4	✓	✓	✓	-3.4	-4.0	-0.1
OF1	✓	✓	×	0.2	1.6	-1.1
OF2	✓	✓	×	-2.6	0.6	2.1
OF3	✓	✓	×	*	2.4	*
OF4	✓	✓	×	3.3	-4.3	3.4

inspection, and comparison with manual patches. Our stress testing inserts random-length sleeps around racing operations for message-timing bugs, and injects node faults randomly in the bug-triggering time window. Under this setting, the original software fails about 50 – 100% of times in 20 runs; the software with DFix patches never fail in our experiments.

Naïve patches for message-timing bugs apply traditional locks and condition-variable signals/waits around racing operations. It fails to fix 8 out of 9 message-timing bugs. It does not apply to AM2 and AM3, as racing objects are znodes on third-party ZooKeeper servers and we cannot change the ZooKeeper library. It causes deadlocks for 6 benchmarks, due to circular waits on event handling thread (OM3), message handling threads (AM1, OM2), and locks (OM4, OM5, OM6).

Naïve patches for fault-timing bugs always restart the crashed node after clean up local temporary files (e.g., we clean up the hadoop.tmp.dir directory). This strategy can fix 1 out of 8 fault-timing bugs (i.e., AF4). For the 4 order violations, simply restarting the crashed node does not help sending a message that another node is waiting for; for 3 atomicity violations (AF1, AF2, AF3), the node crash left inconsistency among global files and local cleanup does not work. Furthermore, restart without local cleanup cannot fix any bugs.

Performance We have measured performance of DFix pat- ches under both correct timing and bug-triggering timing, indicated by $T_{correct}$ and T_{buggy} in table 2.2.

DFix patches introduce almost no overhead under correct timing, comparing with the original software with no code changes ($T_{correct}$ in table). **No** sleep/fault was injected and the baseline run-time varies from 1 second to half a minute in different benchmarks. We take the average of 40 runs for every performance number in the table. As we can see, the overhead of both manual and DFix patches are negligible. The only exception is that, the manual patch inserts a random sleep for AF2, which can lead to huge overhead non-deterministically and leads to a 13% overhead on average in our experiments. Both manual patches and DFix patches occasionally incur negative overheads. The amount is negligible and is caused by

fluctuation of the running environment.

DFix patches are as fast as manual patches under the bug-triggering timing for all but 2 benchmarks, OM2 and OM3 (T_{buggy} in table). We use the manually patched software as baseline, because the original software fails under bug-triggering timing. For OM2, OM3, manual patches actually changed the original software data structure, related processing algorithms and semantics, so that the B operation no longer needs to wait for A operation. The 20% overhead of DFix in actually *inevitable* for any semantic-preserving patches — to trigger the bug, 1–20 seconds of random sleeps are inserted, which not only delays A but also B to guarantee B executes after A. In manual patch, B does not wait for A and hence is not delayed for that 1–20 seconds. In fact, this overhead is much smaller when we trigger bugs with short sleeps.

There are 3 benchmarks, AF1, AF3, and AF4, that DFix can generate two patches, a fast-forward and a rollback patch. Table 2.1 only reports the performance of their fast-forward patches. The performance of rollback patches is similar.

2.5.3 Patch details

Message-timing patches Among the 9 message-timing bugs, 5 of them (AM1, AM2, AM3, OM2, and OM3) require inter-node rollbacks, while the other 4 are fixed through intra-node rollbacks. Among these 4, three of them have their race operations inside synchronized blocks. Consequently, DFix identifies rollback destination outside the synchronized blocks to avoid deadlocks. Object-specific flag IDs (Section 2.2.5) are crucial for 5 of them (AM1, AM2, AM3, OM2, OM3).

As discussed in Section 2.2.6, DFix proves 7 of 9 bug patches to be deadlock free – the waited-for instruction is guaranteed to execute. For the remaining two benchmarks (AM1 and OM2), their waited-for instructions are inside a branch body of the heartbeat handler. Static analysis cannot guarantee those instructions will execute and also cannot identify a

Table 2.3: DFix fault-timing patches.

ID	Fast-forward Fixed?	Rollback Fixed?	#crash-persistent Ops raw	#crash-persistent Ops optimized
AF1	✓	✓	1	1
AF2	×	✓	4	2
AF3	✓	✓	1	1
AF4	✓	✓	2	2
OF1	✓	×	1	1
OF2	✓	×	1	1
OF3	×	✓	1	1
OF4	✓	×	18	2

location, after which those instructions definitely will not execute due to the periodic nature of heartbeat protocol. DFix uses timeouts in all its patches to prevent infinite rollbacks.

Fault-timing Patches As shown in table 2.3, fast-forward and rollback each can fix most but not all of these 8 benchmarks. Fast-forward can fix all but two bugs. For AF2 and OF3, the parameter pre-evaluation depends on shared variables whose content might be changed later during N_{fault}'s execution in the buggy window. Consequently, DFix gives up. Rollback can fix all but three bugs, OF1, OF2, and OF4. The buggy windows of these three bugs contain inter-node communication that DFix cannot handle. These two strategies nicely complement each other and work together to help DFix fixes all these 8 bugs.

The last two columns of table 2.3 present the number of crash-persistent operations inside each buggy time window. One bug's window only involves file updates; 3 only involve message sending; the remaining 4 involve both types of operations. For AF2 and OF4, some of their crash-persistent operations are considered as unnecessary for rollback/fast-forward through DFix static analysis, as discussed in Section 2.3.3 and 2.3.4.

Simplicity DFix patches for all the 17 benchmarks range between 5 to a little over 20 lines of code, excluding utility functions like logging and wait-with-timeouts in DFix library. The path-size differences are mainly affected by the complexity of pre-computing the race-object identity, and the number of crash-persistent operations.

Patch Generation The static analysis used by DFix to generate patches is scalable, as the analysis is applied only to bug related call stacks and causality stacks. All the patches are generated within 200 seconds in our experiments.

Comparison with Manual Patches 10 out of 17 bugs are fixed by developers using *exactly the same* strategy as DFix. One bug's manual patch (AF2) does not completely work. For the other 6, developers redesigned algorithms/data-structures, so that the race operations either disappear (3 bugs) or the originally unexpected timing no longer causes failures (3 bugs). The former three patches are much more complicated, and the latter three are simpler than DFix patches.

Here are two examples where patches manually developed by programmers use exactly the same strategy as patches automatically generated by DFix. Bug ZK-1270 (OM5) is a message-timing bug: O_1, a local operation, races with O_2, an operation inside a message-handler. Manual patch introduces a variable to indicate if O_1 has executed (like DFix in Section-3.3) and rolls back O_2's message handler when the checking fails (like DFix in Section-3.4). Manual-patch's re-execution region is the same as DFix-patch. Bug ZK-1653 (AF4) is a fault-timing bug: a node writes new epoch-ID to file A and then B. If it crashes in between, restart would fail due to inconsistent epoch-IDs in A-and-B. The manual patch creates a flag-file to indicate whether A and/or B was updated, just like DFix. Using that file, restart checks if the crash was in between and, if so, copies new epoch-ID from A to B, just like how DFix fast-forwards B's missing update.

2.5.4 More

After I playing with many difficult synchronization in these cloud system, one question raised to my head: can we have a general model to understanding the synchronization for concurrency bug detection and fixing? This question guide me to work on the following two

chapters.

CHAPTER 3

EFFICIENT AND SCALABLE THREAD-SAFETY VIOLATION DETECTION

3.1 Motivation and Background

3.1.1 Why Concurrency Testing in the Large?

The broader goal of this Chapter is to find concurrency errors during testing, ideally "in the small."

Our specific goal here is to build a concurrency testing tool as part of integrated build and test environments that are becoming popular in software companies today [25, 52]. Such an environment provides a centralized code repository for all the different engineering groups. The code is incrementally built on every commit and on specific successful builds; a collection of servers automatically run unit and functional tests. It is not uncommon to have hundreds of servers dedicated to run tests from tens of thousands of modules written by many different groups.

Testing at this scale introduces key challenges: to minimize productivity loss, tools should have little/no false positives; to reduce overheads, the tools should have minimal runtime overhead and not require rerunning the test too many times; for wide adoption across a variety of product groups, the tool should be "push button" requiring little or no configuration and support a variety of programming models. TSVD discussed in this Chapter is one such tool.

3.1.2 Why Thread-Safety Violations?

A data race occurs when two threads concurrently access the same variable and at least one of these accesses is a write. Thread-safety violations are a generalization of data races to objects and data structures. Each class or library specifies, sometimes implicitly, a *thread-safety*

contract that determines the set of functions that can be called concurrently by threads in the program. For instance, a dictionary implementation might allow two threads to concurrently perform lookups, while another implementation might disallow such calls if lookups can sometimes trigger a “rebalance” operation. In the extreme, a lock-free implementation might concurrently allow any pairs of calls.

A thread-safety violation occurs when the client fails to meet the thread-safety contract of a class or library, as discussed earlier in Figure 1.4. While the notion of thread-safety contract can be more general, in this Chapter we will assume that the methods of a data structure can be grouped into two sets — the read set and the write set, such that two concurrent methods violate the thread-safety contract if and only if at least one of them belongs to the write set. All the data structures we study in this Chapter have this property.

Thread-safety violations are easy to make but hard to find in code reviews. Sometimes programmers deliberately make such concurrent accesses due to their misunderstanding of the thread-safety contract. In fact, TSVD is particularly inspired by a thread-safety violation similar to Figure 1.4 that caused a significant customer-facing service outage for days and took weeks to debug and fix.

Focusing on thread-safety violations solves some of the issues with data race detection. By definition, every TSV is a concurrency error while a data race can either be a concurrency error or a *benign* one [125] (that is nevertheless disallowed by modern language standards [41, 120]). Since thread-safety violations are defined at a larger granularity of objects and libraries, TSVD needs not to monitor every shared-memory access as data-race detectors do, which is one, but not all, of the reasons for the good performance of TSVD.

Note that in common parlance the term “thread safety” is defined ambiguously. Our notion here is different from high-level data race [128]. Similarly, [133] use “thread-safety” as a synonym to linearizability. Instead, throughout the Chapter, we will use TSVs to refer the violation of the thread-safety contract as defined above.

3.1.3 Why Not Happens-Before (HB) Analysis?

Happens-before [95] analysis is widely used in data-race and general concurrency-bug detection. In theory, once we capture all the synchronization/causal operations at run time and figure out all the happens-before relationships, we can accurately tell which operations can execute in parallel. Unfortunately, in practice, this is extremely challenging to conduct accurately and efficiently. We discuss the challenges in the context of TSVD below, which motivates the design of HB *inference*, instead of HB analysis, in TSVD. We will also design a variant of TSVD that conducts HB analysis (Section 3.2.5), which allows us to compare TSVD with different detection techniques.

General challenges First, it is difficult to capture all synchronization actions, as programs developed by thousands of engineering teams use numerous concurrent libraries, system calls, asynchronous data structures, volatile variables, and others for communication and coordination. Missing an HB-edge associated with such an action can result in false bug reports, while a spurious edge can hide true bugs.

Second, the cost of tracking and analyzing all synchronization actions, including many library calls, volatile/atomic variable accesses, and others, is high. Particularly, the analysis of HB relationships in programs that allow asynchronous concurrency, like all C# programs TSVD targets, is extremely memory and time consuming [75, 119, 139, 144].

Challenges for arbitrary task parallelism Previous work lowers the cost of dynamic HB tracking and analysis based on certain assumptions, such as (1) a fixed or small number of threads, or (2) having many more data accesses than synchronization operations [56, 132, 149], or (3) having task forks and task joins in a structured task parallel program restricted to series-parallel graphs only [54, 136, 137].

These assumptions however do not hold for the programs TSVD targets: (1) these programs dynamically create many tasks, many more than the number of threads, and

dispatch them to execute on concurrent background threads; (2) the number of data accesses no longer dominates that of synchronization operations, which include frequent task creations and joins, because data accesses are traced and checked at the granularity of thread-unsafe method calls and hence have a much smaller quantity in TSV detection; (3) these programs pass around task handles as first-class values and join with *any* task via its handle — forks and joins no longer follow series-parallel graphs. Such paradigms include not only traditional threading libraries like POSIX threads in C, Java's `Thread` class, and `System.Threading` in .NET, but also task-parallel libraries built on top of such primitives, like `java.util.concurrent` and `std::future` in C++.

For example, in .NET Task-Parallel Library (TPL), asynchronous work units are represented by `Task` objects. Tasks can be forked explicitly through APIs like `Task.Run`, or implicitly through data-parallel APIs like `Parallel.ForEach`, or through C# async/await paradigm. Any task can join with any other task using various mechanisms, such as explicit `Task.Wait` or implicitly blocking on a task's result via `Task.Result`. Task-level parallelism is hence unstructured.

An example To illustrate the arbitrary task parallelism and TSVs, Figure 3.1 shows a simple C# code snippet that uses explicit task creation and the async/await paradigm. Its core is the function `getSqrt`, which gets the square-root of its first argument by either retrieving from a table (lines 3–4) or computing from scratch (line 7).

Figure 3.2 illustrates the execution order of this code snippet when neither `a` nor `b` is in the lookup table, with the *happens-before* relation highlighted in arrows. As we can see, a call to `getSqrt` from lines 13 quickly leads to the fork of a `Task` by line 6 to carry out the computation (7a) in a concurrent background thread, while the `getSqrt` itself returns the spawned task immediately and moves on to the next call of `getSqrt` on line 14, denoted by 8a→ 14 in Figure 3.2. Later on, when the computation of the forked task completes, the *continuation* after the `await` statement is resumed from line 9 (i.e., 9a and 10a in Figure

```
async Task<double> getSqrt(double x,
            Dict<double, double> dict) {
  if (dict.ContainsKey(x)) {
    return dict[x];      // fetch from cache
  } else {
    Task<double> t = Task.Run(() =>
        Math.sqrt(x));   // background work
    double s = await t; // resume when done
    dict.Add(x, s);      // save to cache
    return s; }}

/* Assumes `a`, `b`, and `dict` exist */
Task<double> sqrtA = getSqrt(a, dict);
Task<double> sqrtB = getSqrt(b, dict);
Console.WriteLine(sqrtA.Result    // blocks
               + sqrtB.Result); // blocks
```

Figure 3.1: Example of task parallelism in C#. TSVs can occur due to concurrent accesses of `dict`.

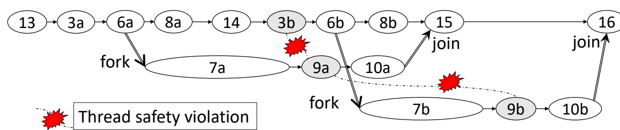


Figure 3.2: Happens-before graph for Figure 3.1, assuming neither `a` nor `b` is in the dict. The nodes show the executed line#; subscripts distinguish multiple executions of the same line.

3.2). Meanwhile, any attempt to retrieve the task's result like that on line 15 blocks until the corresponding continuation returns from `getSqrt` after line 10, illustrated by the Join arrow in Figure 3.2.

As shown in Figure 3.2, two instances of `dict.Add` (9a and 9b) are concurrent with each other and form a write-write TSV. Similarly, `dict.Add` (9a) and `dict.ContainsKey` (3b) form a read-write TSV. Detecting them via precise happens-before analysis in a large program is difficult due to challenges discussed earlier.

3.2 Algorithm

This section describes the design of TSVD along with its variants that represent alternative designs and will be used to compare with TSVD in our evaluation.

3.2.1 Overview

TSVD and its variants share the same *trap framework* shown in Figure 3.3, similar with that in DataCollider [50].

We assume a static analysis that identifies all the call sites to relevant data structures' methods whose thread-safety needs to be checked in the code (Section 3.3 will provide more details of such a list of methods used in TSVD). This analysis is straightforward and it reports call sites without checking their calling contexts (e.g., even if they are used within locks). We will refer to these call sites as *TSVD points*. We also assume an instrumentation framework that can instrument the program with calls to TSVD right at these TSVD points. The only interface to TSVD is the `OnCall` with arguments that show the thread making the call `thread_id`, the object being accessed `obj_id`, and the operation being performed `op_id`.

The trap mechanism works as follows. Consider a thread calling the `OnCall` method. On some calls chosen by the `should_delay` function at line 3, the thread sets a trap by registering the current method call in some global table at line 4. A trap is identified by the triple of

```
OnCall(thread_id, obj_id, op_id){
  check_for_trap(thread_id, obj_id, op_id)
  if(should_delay(op_id)){
     set_trap(thread_id, obj_id, op_id)
     delay()
     clear_trap(thread_id, obj_id, op_id) }}
```

Figure 3.3: The trap mechanism used by TSVD and its variants

the identifiers of the thread, the object, and the operation. Then, the thread calls the delay method to sleep for some period, during which the trap is "set."

Every other thread on entering OnCall checks if it conflicts with the traps currently registered at line 2. Formally, if a trap (tid_1, obj_1, op_1) is set and another thread enters the method with the triple (tid_2, obj_2, op_2), this pair results in a TSV iff $tid_1 \neq tid_2$, $obj_1 = obj_2$, and at least one of the two operations op_1 and op_2 is a write.

When such a conflict occurs, we have caught both threads "red handed" as they are at their respective program counters making the conflicting method calls to a common object. We can report the TSV with enough information gleaned from the current state for root causing this bug, such as the stack trace of the two threads. In theory, we could report other relevant information such as object content, but we have not found the need to do so in our experience with TSVD so far.

When the first thread wakes up from its delay, irrespective of whether a conflict was found or not, it clears the trap at line 6 and returns from the OnCall method.

TSVD and its variants differ in their answers to two key design questions about should_delay:

1. Where to inject delays? Which program locations are eligible for should_delay to suggest delay injections.

2. When to inject delays? In which program runs and at which dynamic instances of an

eligible delay location, should should_delay suggest a delay injection.

When making the design decisions above, we have to keep the following goals and constraints in mind:

Runtime overheads — since the should_delay function is called inside *every* OnCall function, it is important to limit its complexity to reduce the runtime overhead. At the same time, if this function is not selective enough, we will inject too many delays, which in turn increases the runtime overhead.

Number of testing runs — if the should_delay function is too selective, then it can either miss bugs or require running the test too many times, increasing the testing-resource requirements.

Accuracy goals — no/little false positives are demanded; few false negatives without consuming too much resource is desired.

Complexity constraints — we would like the tool to work across code from many engineering teams. Specifically, the tool should handle a variety of synchronization mechanisms that developers can use to prevent TSVs. Moreover, sophisticated static analysis for arbitrary C# is challenging. For instance, a ForEach.Parallel block can sometimes be translated to a sequential loop when only one physical thread is available, and to a parallel loop otherwise. Determining this statically is a hard problem. For this Chapter, we restrict ourselves to dynamic techniques. As any dynamic analysis tool, TSVD cannot guarantee to find all TSVs in the program. However, by guaranteeing to not report any false errors, it guarantees "soundness of bugs" [64].

Next, we present the variants that take different design points in the design space in Figure 1.5. We start with two simple variants that occupy the top-left corner in Figure 1.5 and serves as baselines for TSVD.

3.2.2 TSVDbase

The simplest algorithm takes every TSVD point as an eligible delay location (*where*), and injects delays at random moment (*when*). In other words, the call to should_delay return true with some (small) probability. Once deciding to delay a thread, the thread sleeps for a random amount of time.

3.2.3 DataCollider

Previous work that uses delay injections to expose data races, such as DataCollider [50], observed that dynamic sampling is ineffective, as it tends to insert a lot more delays along hot paths and ignore cold paths where even more bugs may be hidden. To avoid redundantly introducing delays in hot paths and to focus on cold paths, DataCollider samples memory accesses "statically" — that is, static program locations are sampled uniformly irrespective of the number of times a particular location is sampled. We emulated this algorithm in our DataCollider variant, where static call sites of relevant data structures' methods are sampled uniformly.

Comparing with TSVDbase, *where* to inject delays does not change, still all the TSVD points, but *when* to inject delays differentiates code paths from hot paths.

3.2.4 TSVD

Our motivation for TSVD came after our initial experience implementing TSVDbase and DataCollider in our setting. While these variants were finding bugs, many of the bugs required running the same test tens of times, which was unacceptable in our setting. On investigating further, we found that these variants were injecting a large fraction of its delays at the "wrong" places — either when the program is executing in a single-threaded context, say when all other threads are waiting for external events, or when the program was accessing the data structure in a "safe" manner - say by holding the right lock. In both cases, a delay directly

translates to an end-to-end slowdown, forcing us to drop the probability of delays to reduce the runtime overhead. This, in turn, resulted in missed bugs or a proportionally more number of runs to find true bugs.

TSVD is specifically designed to avoid these problems. As illustrated in Figure 1.5, TSVD explores a new design point. It uses *local* instrumentation and no synchronization modeling or happens-before analysis to reduce the runtime overhead, while identifying potential candidates for delay injection. By focusing on more likely candidates of thread-safety violations, it can achieve much better bug-exposing capability with no more, much less in fact, resource consumptions.

Where to inject delays?

At high level, TSVD uses lightweight analysis to dynamically maintain a set, called *trap set*, of *dangerous*-pairs of program locations that can likely contribute to thread-safety violations and injects delays at only these locations.

During a testing run, the size of the trap set grows as TSVD discovers new dangerous pairs, or shrinks as TSVD prunes existing dangerous pairs.

TSVD identifies a pair of program locations as a dangerous pair if (1) they correspond to a near-miss TSV (§3.2.4) — intuitively, TSVD hopes to turn previous near misses into true conflicts, and (2) at least one of these two locations runs in a *concurrent phase* with multiple active threads of the program (§3.2.4).

A dangerous pair is pruned from the trap set if (1) TSVD infers a likely happen-before relationship between the two locations (§3.2.4) and hence the pair is unlikely to violate thread-safety, or (2) a violation is already found at the pair.

Finally, the delay injection is probabilistic—each program location *loc* in the trap set is associated with a probability P_{loc} at which delay is injected at location *loc*. TSVD sets $P_{loc} = 1$ when a dangerous pair containing *loc* is added to the trap set, and the probability

is decayed with time (§3.2.4). This probabilistic design can help avoid excessive overhead caused by too many delays for dangerous pairs that are actually not bugs.

We now describe the lightweight mechanisms TSVD uses to infer the key components required to make the above decisions. Note that, our evaluation section will show thorough parameter-sensitivity study of *all* the thresholds/parameters mentioned below.

Identifying near misses

Intuitively, if a pair of program locations never even get close to creating a thread-safety violation under the intensive testing environment, they might just never be able to create one due to underlying program semantics and synchronization operations. On the other hand, say an access[1] happens at program location loc_1 with the access triple (tid_1, obj_1, op_1) at time t_1 and another access happens at program location loc_2 with the access triple (tid_2, obj_2, op_2) at time t_2. TSVD considers the pair (loc_1, loc_2) as a dangerous pair if $tid_1 \neq tid_2$, $obj_1 = obj_2$, at least one of op_1 and op_2 is a write, and $|t_1 - t_2| \leq \delta$ for some time threshold δ.

To make this judgment, TSVD maintains a list of accesses per object since δ time ago. For implementation simplicity, rather than storing this state in the object metadata, TSVD maintains a global hash table indexed by the object's hash-code containing this list of previous accesses. On each access, TSVD consults this list to identify dangerous pairs.

Inferring Concurrent Phases

Synchronization operations like `forks`, `joins`, `barriers`, and `locks` could lead to sequential execution phases during the execution of a concurrent program, like initialization phase, clean-up phase, join-after-fork phase, and so on, and a TSVD point inside such a sequential phase can never execute in parallel with another TSVD point.

1. For the ease of discussion, we do not differentiate an access to an object and a thread-unsafe method call of an object in this section.

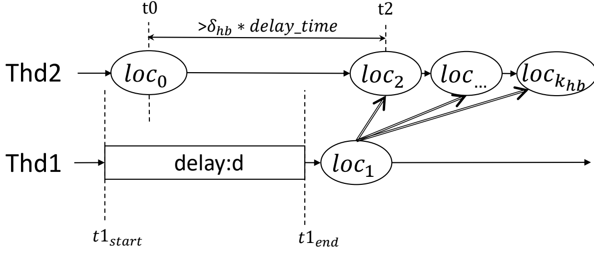


Figure 3.4: Happens-before inference in TSVD (the thick arrows indicate inferred happens-before relationship)

TSVD infers concurrent execution phases by checking the execution history of TSVD points. Specifically, it maintains a global history buffer with a fixed number of most recently executed TSVD points. If the TSVD points in this buffer come from more than one thread, TSVD considers the execution to be inside a concurrent phase.

Inferring likely HB relationship

TSVD's identification of dangerous pairs based on near-misses and concurrent phases can potentially be incorrect. For instance, two accesses that are consistently protected by the same lock can happen close to each other. TSVD will fail to convert these near misses into true conflicts due to the dynamic happens-before (HB) relation between the two accesses. Rather than track synchronizations to calculate this HB relation, TSVD uses the lightweight technique below to infer likely HB relationships.

TSVD's dynamic HB inference works based on the following crucial observation. If loc_1 happens-before loc_2 in an execution, then a delay right before loc_1 will *cause* a proportional delay of loc_2. As shown in Figure 3.4, consider the case when loc_1 and loc_2 are consistently protected by a lock and loc_1 occurs first. When one injects a delay right before loc_1, this delay occurs when the thread executing loc_1 is still holding the lock. Thus, when the thread executing loc_2 tries to acquire this lock (which it needs to before accessing loc_2), it will block. By dynamically tracking this causality between program locations, TSVD infers *likely* HB

relationships between them.

Tracking this causality works as follows. Apart from per-object history described above, TSVD additionally maintains a history of the most recent access made by every thread. Consider a delay d_1 injected right before access loc_1 that starts at $t1_{\text{start}}$ and ends at $t1_{\text{end}}$. At a subsequent access at loc_2 in a different thread Thd_2 happening at time $t2$, we will check when the previous access from Thd_2 occurred, denoted as $t0$, and infer a dynamic HB relationship between loc_1 and loc_2 if (1) there is a long gap between loc_2 and that previous access, $t2 - t0 \geq \delta_{hb} * delay_time$, a causal delay threshold and (2) the long gap overlaps with the injected delay, $t0 \leq t1_{\text{end}}$. If multiple delays $\{d$ at $loc\}$ satisfy the inferring condition for Thd_2 and loc_2, we contribute the long gap in Thd_2 to the most recently finished delay d and infer the HB relationship accordingly. Considering the transitivity of happens-before relationship, the next k_{hb} accesses in thread Thd_2 are also considered as likely happens-after loc_1. Again, k_{hb} is a tunable parameter and we will evaluate its sensitivity in the evaluation section.

Delay Decaying

Of course, all the inferences above, including the near-miss tracking, the concurrent phase inference, and the likely HB-relationship inference, still do not guarantee to prune out all non-TSV pairs. Consequently, for every dangerous pair $\{p_1, p_2\}$ in the dangerous list, every delay injection at either p_1 or p_2 that fails to make p_1 and p_2 execute in parallel with the same object accessed will cause TSVD to decay the probability of future delay injection at p_1 and p_2. When the probability of a location loc drops to 0, all its related pairs are removed from the trap set.

When to inject delays?

Different from previous work that separates delay planning and delay injection into different runs [129, 147] and/or using many runs to gradually inject delays to all planned locations [50, 129, 147], TSVD conducts its delay planning *and* injections in the *same* run, making the best use of testing resources.

Planning & injection in the same run Once TSVD identifies a dangerous pair of program locations (that nearly missed each other), TSVD does not wait until the next run to expose a potential thread-safety violation between them. The rationale is that most instructions execute more than once and hence most bugs have multiple chances to manifest. This is confirmed by our experimental results in Section 4.4 showing that TSVD hits the same bug multiple times. Consequently, TSVD can still try to expose a bug after an initial near-miss. Specifically, whenever a program location *loc* is going to be executed, TSVD checks if *loc* is part of the current trap set. If it is, TSVD will insert a delay at *loc* with probability P_{loc}.

Multiple testing runs The above delay injection scheme would miss a bug if the near-miss situation observed by TSVD was actually the only chance to expose the bug; i.e., if the dangerous pair of locations never run together after the pair is identified. When the testing resource allows, TSVD runs the program for another time, carrying the knowledge obtained from the first run, to help discover these bugs.

During the first run, TSVD records its trap set in a persistent *trap file*. At the end of the run, the trap file contains the final set of dangerous pairs. At the beginning of the second run, TSVD initializes its trap set from the file, allowing it to inject delays at pairs even at their first occurrences.

Parallel delay injection There are clear trade-off among injecting only one delay or multiple delays throughout a run, and allowing only one thread or multiple threads to be delayed at a time: when multiple delays are injected, they could cancel each other's effect; when few delays are injected, we will need too many runs to test all possible delay situations.

TSVD decides to conduct delay injection in an aggressive way — delays are inserted strictly following the dangerous pair list and the decay-probability scheme, regardless of whether another thread is already blocked. Note that, although this aggressive strategy can cause some injected delays to overlap with each other, the overlaps will only be partial and are unlikely to cancel each other out. This is because different threads will be unblocked at different time and our decay-probability scheme also helps avoid too many threads blocked at the same time. On the other hand, an alternative design that strictly avoids delay overlaps would lead to too few delay injections and hence hurts our chance of exposing bugs within the tight testing budget, as we will demonstrate in the evaluation section.

3.2.5 TSVD with happens-before analysis

To compare TSVD with existing techniques that rely on HB analysis, we designed a variant of TSVD, referred to as TSVDhb, that follows the approach of RaceFuzzer [147]. Note that, TSVDhb uses a few optimizations to speed up traditional HB analysis following the type of bugs it targets.

Where to inject delays? Just like that in TSVD, TSVDhb dynamically maintains a trap set, with every program location appearing in the trap set a delay-injection candidate. At run time, TSVDhb adds a pair of program locations $\{loc_1, loc_2\}$ into the trap set, if the corresponding accesses (tid_1, obj_1, op_1) and (tid_1, obj_1, op_1) satisfy that $tid_1 \neq tid_2$, $obj_1 = obj_2$, op_1 and op_2 conflict, and most importantly they do not have happens-before relationship with each other.

TSVDhb monitors synchronization operations, such as locks, forks, and joins, and uses vector clocks [55] to compute the happens-before relation between accesses.

TSVDhb optimizes traditional race detection to speed up its HB analysis following the observation that, different from traditional race detection, where execution traces contain few synchronization operations but many memory accesses, TSVDhb faces many synchronization operations (the large number of `async`/`await` in C# cloud service modules) yet relatively few accesses (i.e., TSVD points — invocations of thread-unsafe methods of some data structures).

The first optimization is to increase local timestamps at accesses (TSVD points), which are relatively few at run time, but not at synchronization operations, which are relatively frequent, opposite from traditional race detection [56].

The second optimization is that TSVDhb uses immutable data structures—AVL tree-maps—to represent vector clocks, unlike traditional implementations which use mutable arrays or hashtables. For vector clocks with n components, a message-send (or other similar types of synchronization) event requires an $O(n)$-time/memory copy with traditional mutable tables, whereas immutable clocks can be passed by reference in $O(1)$-time. On the flip side, TSVDhb requires $O(\log n)$ time/memory to perform increment operations, whereas mutable clocks can be updated in-place in $O(1)$ time. Fortunately, TSVDhb restricts increment operations to infrequent TSVD points only. Finally, message-receive (or other similar type of synchronization) events require $O(n)$ time with both data structures, since an element-wise max must be computed. Fortunately, in the relatively common case where tasks fork and join without going through any TSVD points, the vector clocks associated with the join-message and with the receiving thread are exactly the same object; such an operation can be optimized to $O(1)$ by simply checking for reference equality.

When to inject delays Unlike RaceFuzzer, which uses many runs that are each targeted at a specific pair of potential data races, TSVDhb aims to expose bugs in a small number of testing runs just like TSVD. Consequently, just like that TSVD, TSVDhb injects delays in

the same run as the happens-before analysis and trap-set fill-up; it can delay multiple threads simultaneously. When one testing run is finished, those remaining pairs in the trap set are recorded and fed to the testing run, if there is one. Like TSVD, TSVDhb uses a probability decay to prune possibly spurious dangerous pairs.

3.3 Implementation

We have implemented TSVD for .NET applications (e.g., C# and F#). It has two key components: (1) TSVD runtime library that implements the core algorithm, and (2) TSVD instrumenter that, given a .NET binary, automatically incorporates TSVD runtime library into it by static binary instrumentation. We chose static instrumentation of application binary as this enables us to use TSVD instrumented binaries with unmodified, existing test pipeline (unlike framework instrumentation [168], which requires updating test pipeline with instrumented Common Language Runtime (CLR) or dynamic instrumentation [23], which requires modification of test environment). While our implementation is specific to .NET, we believe the techniques underlying TSVD can be easily implemented for other languages and runtimes.

TSVD Instrumenter This takes a list of application binaries and a list of target thread-unsafe APIs. In the current prototype of TSVD, we focus on 14 C# thread-unsafe classes (e.g., List, Dictionary) and manually identified 59 of their APIs to be write-APIs and 64 as read-APIs. This process is straightforward for these data-structure classes and has a small cost that is easily amortized by reusing this list for all C# program testing. It then instruments the binaries by replacing each thread-unsafe API call with an automatically generated proxy call, as shown in Figure 3.5. The proxy wraps the original call; but, before making the original call, it calls the `OnCall` method (Figure 3.3) implemented in TSVD runtime library. TSVD uses Mono.Cecil [79] to do the instrumentation.

```
List<int> listObject = new List<int>();
listObject.Add(15);
```

(a) Original code

```
List<int> listObject = new List<int>();
int op_id = GetOpId();
Proxy_123(listObject, 15, op_id);
```

(b) Instrumented code

```
void Proxy_123(Object obj,int x,int op_id) {
  var thread_id = GetCurrentThreadId();
  var obj_id = obj.GetHashCode();
  OnCall(thread_id, obj_id, op_id);
  obj.Add(x); }
```

(c) Proxy method

Figure 3.5: TSVD instrumentation

The TSVD instrumenter can be used as a command line tool or with Visual Studio (as a post-build step). To allow a developer to use TSVD without additional configuration, TSVD comes with an extensible list of thread-unsafe APIs in .NET standard libraries. The list also includes information about whether the APIs are read- or write-APIs.

TSVD Runtime This implements the `OnCall` method, which executes the core TSVD algorithm (Figure 3.3). In addition, (1) it logs runtime contexts when a bug is detected. The contexts include bug locations (e.g., method name and source code line number) and stack traces of racing threads. (2) It tracks total delay injected per thread and per request so that one can limit the maximum delay per thread or request. This helps in avoiding test timeouts. (3) It tracks test coverage of instrumented APIs. (4) Finally, it updates signatures of instrumented signed binaries.

While using TSVD in Microsoft, we observed that many asynchronous programs using async/await run synchronously in test settings, and therefore, TSVD cannot find thread-safety bugs that could manifest in production runs when the programs do run asynchronously. We

found that the underlying reason is a .NET runtime optimization that executes fast async functions synchronously. This affects many tests that replace async I/O calls with fast mock implementations. To address this, TSVD instruments async/await code to force all async functions, slow or fast, to run asynchronously. Note that, in our experiments, this strategy will be applied to not only TSVD but also all the techniques that will be compared with TSVD.

An open-source version of TSVD is available at `https://github.com/microsoft/TSVD`.

3.4 Evaluation

3.4.1 Methodology

Benchmarks For our evaluation, we collected approximately $43K$ software modules from 1,657 projects under active development at Microsoft. We call this the *Large* benchmark. These modules are regularly tested in an integrated build and test environment. Each module contains a collection of unit tests as well as all binaries required to execute the tests. Together, they contain a total of around $122K$ multi-threaded unit tests and $89K$ program binaries. To test a module with TSVD, we instrument its binaries and run its existing unit tests. 2% of the modules also included long-running end-to-end tests, that we also run. Of all the software modules that were available in the integrated environment we selected those modules which a) were written in a .NET language such as C# or F# as TSVD is currently implemented for .NET programs, b) passed all their tests over the past 2 builds, ensuring that these modules are stable and well-tested, and c) used multithreading primitives or C# `async/await` mechanisms at least once. We will use this to evaluate the bug-exposing capability and performance of TSVD.

To evaluate TSVD against alternate designs and parameter settings, we sampled 1000 modules randomly from the Large benchmark. We call this the *Small* benchmark. Together,

this suite consists of 3350 multi-threaded unit tests.

Experiment Setting We run the small benchmark suite on a small server (S1), with Intel(R) Xeon(R) E5-1620 CPU, 16G Memory, and 1T SSD. Since all the software modules under test are independent with each other, we run 10 modules at a time on S1. We run the big suite on a large server (S2), with Intel(R) Xeon(R) E5-2650 CPU, 128G memory, and 6T SSD. Both S1 and S2 use Windows10 operating system.

3.4.2 Overall Results

Bugs Found Over last few months, we have been using variants of TSVD on the Large benchmark. Table 3.1 summarizes the results. Overall we found 1,134 bugs, uniquely identified by the pair of static program locations participating in the TSV, involving 1,180 unique static program locations.[2] 1,009 of these bugs were found using the core TSVD we present in this Chapter; while the remaining bugs came from our previous attempts of using variants of TSVD, namely TSVDbase and TSVDhb.

For each bug, TSVD produces a pair of stack traces that shows two threads actively participating in a thread-safety violation. Note that the same bug can manifest at multiple stack trace pairs. On the other hand, multiple bugs in a high-level data structure might show up as a conflict in a low-level data structure with the same program-location pair. Overall, we found the above bugs in 21,013 different stack trace pairs (i.e., 18.5 stack trace pairs per bug). Due to our inability to identify whether two different stack-trace pairs correspond to the same "bug" or not, we will use the conservative program-location pair for unique bug counts.

On average, we found at least one bug in 9.8% of the projects and 1.9% of the modules. As TSVD find these bugs while running existing tests provided by developers, TSVD does

2. Note that unsynchronized accesses to n locations can theoretically result in n^2 location pairs. However, our reported bug count (i.e., 1,134 location-pairs) is not bloated by this phenomenon (for 1,180 locations).

Test Targets	
# of projects	1,657
# of test modules	$\approx 43K$
# of binaries	$\approx 89K$
# of tests	$\approx 122K$
Bugs found	
# of unique bugs (location pairs)	1,134
# of unique bug locations	1,180
# of unique stack trace pairs	21,013
% of projects with bugs	9.8%
% of modules with bugs	1.9%
Bugs properties	
% of read-write bugs	48%
% of same location bugs	34%
% of bugs in `async` code	70%
Avg. (Median) occurrence of a bug location	36(4)
Avg. (Median) # stack trace pairs/bug	18.5(3)
Avg. Stack depth	9.1
% of `Dictionary` bugs	55%
% of `List` bugs	37%

Table 3.1: Summary of bugs found by TSVD tools.

not report false error reports.

Bug Validation We contacted four product teams in Microsoft to validate *all* bugs TSVD found in their software (total 80). *All* of the bugs were confirmed as *true* bugs. Of these, 77 were *new* bugs previously not known to the developers and the remaining 3 were concurrently found by other means. We are still in the process of contacting other teams.

Bug Quality 38 of these bugs were confirmed as "high priority." This internal classification of bugs signifies that these bugs, if manifested in production, would result in a customer-facing outage and thus needs to be fixed immediately. These bugs were immediately fixed by the developers. 22 bugs were of "moderate priority" of which 18 have been fixed so far. The remaining 20 bugs were in non-critical code, such as the test driver or mock code written for unit testing. While these bugs are in lower priority, they nevertheless need to be fixed for preventing nondeterministic test failures.

Actionable Reports Overall, the developers found the error reports to be sufficiently actionable. The stack traces of the two conflicting threads provide enough detail to root cause the problem. In many cases, the fix involves introducing additional synchronization or replacing the data-structure with a thread-safe version of the data structure.

Apart from the error reports, TSVD also reports statistics on the instrumentation points that were hit during the test in any context and in a concurrent context. One team found these "coverage" statistics to be very useful and identified a few blind spots in their testing, such as critical parts only called in sequential contexts during unit testing etc.

Bug Nontriviality Note that all these software teams employ their usual production test pipelines to uncover various types of bugs. The bugs TSVD found had been in their systems for many months but remained undetected by the test pipelines. To further confirm that the bugs TSVD finds are not easy-to-find, we worked closely with one of the teams to track and

compare the bugs found by TSVD and those found by their usual test pipeline. During a 3-months period, TSVD found 15 thread-safety bugs (all confirmed as "priority one" and immediately fixed) in their code, none of which was found by the test pipeline during the same period.

Bug Characteristics 49% of the 1,134 bugs we found were due to concurrent read-write conflicts on thread-unsafe objects. An interesting root cause of such bugs is locking only writes, but not reads, to a thread unsafe object. 70% of the bugs were triggered by code using async/await or Task Parallel Library. Many of these bugs were found benefiting from TSVD's instrumentation design that forces all async functions, slow or fast, to run asynchronously (§3.3). 34% of the bugs manifest due to two threads executing the same operation. Average stack depth of the bug location from test code is 9.1, indicating that many of the bugs appear deep inside production code. Note that the same bug location can appear in multiple bugs (counted as location-pairs) and the same bug can appear multiple times during the same test run. In our experiments, a bug location appeared 35.6 times on average (median 4 times). Despite repeated appearances, the buggy locations remained undetected with existing test techniques in Microsoft. More than half of the bugs involve the `Dictionary` data structure. Note that .NET's standard library includes thread-safe `ConcurrentDictionary`. Yet, our results show that developers often use thread-unsafe `Dictionary` in multi-threaded applications, without necessary synchronization primitives. Talking to developers, we found one interesting cause behind such incorrect usage: erroneously assuming that two writes to a dictionary would be thread-safe as long as the keys are different.

3.4.3 Comparison with other detection techniques

We compare TSVD with DataCollider and other detection techniques discussed in Section 3.2 using the 1000-module Small benchmark suite. We compare them in terms of the number of

	# bug				
	Total	Run1	Run2	overhead	# delay
DataCollider	25	22	3	378%	77402
TSVDbase	13	6	7	178%	31456
TSVDhb	41	25	16	310%	3328
TSVD	53	42	11	33%	22632

Table 3.2: Comparing TSVD with other detection techniques.

bugs found in two rounds of run and the overhead, which is computed based on the additional amount of time imposed by a tool upon uninstrumented baseline testing runs. Results are summarized in Table 3.2 (TSVDbase uses 0.05 probability in its delay injection).

Number of bugs found TSVD found the most number of bugs: 42 bugs in the first round, and 11 additional bugs in the second run. Each of these 11 bugs contains a TSVD point that only executes once during the unit testing, and hence cannot be exposed after the *near-miss* at the first run. TSVDbase and DataCollider detected significantly smaller number of bugs as they have a small probability (0.05 for TSVDbase) injecting delay at a TSVD point. They can potentially find more bugs if run additional rounds, but it would take many rounds for them to catch the bug-reporting capability of TSVD. TSVD outperforms TSVDhb because TSVDhb can both miss Happens-Before edges and add spurious ones, just like other dynamic Happens-Before-based techniques [129]. The huge analysis overhead of TSVDhb also interferes more with bug exposing at run time.

We observed a similar trend in a larger benchmark of 8000 modules. In two rounds of run, TSVD found 256 bugs (227 in the first round), while TSVDhb found 192 bugs and TSVDbase found 79 bugs.

One interesting observation from Table 3.2 is that TSVD found more bugs in its first round alone than other techniques found in two rounds. Moreover, TSVD's first round found about 80% of all bugs found by all tools. Both these observations hold for our full benchmark suite as well. This justifies our design of not separating delay planning and injection into

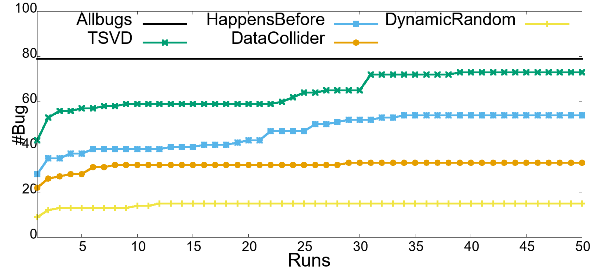


Figure 3.6: Number of bugs found after more runs

different runs. It also provides a more resource-conserving option for using TSVD: while testing under severe resource constraints, one can choose to run TSVD for only one round and still capture vast majority of the bugs.

Note that even though TSVDhb found more *new bugs* than TSVD in the second round in Table 3.2, they both found comparable numbers of *total bugs*: 41 and 53 respectively.

Performance comparison As we can see in Table 3.2, TSVD not only finds the most bugs, but also with the least overhead. The performance advantage of TSVD over all other variants comes from two aspects. Comparing with TSVDbase, TSVD injects most delays when program running with multiple threads. TSVDbase injects many delays in sequential phase which never detects bugs but also introduces huge overhead. Comparing with TSVDhb, TSVD and TSVDhb both inject delay in the concurrent phase, yet TSVDhb spends much time in analyzing the happens-before relationship at run time, which introduces 270% run-time overhead on average in our experiments.

Number of bugs after more runs Given the non-deterministic nature of TSVs and the probabilistic nature of some of these techniques, we ran each of these techniques 50 times with the Small benchmark suite to see how many bugs can be found at the end. As we can see in Figure 3.6, even after many more runs, the bug detection advantage of TSVD still holds over the other techniques. All the four techniques altogether discovered 79 bugs at the

end, with 73 of them discovered by TSVD, 54 of them discovered by TSVDhb, and much fewer by the other two. Furthermore, most of these bugs (close to 70%) can indeed get caught with just two runs of TSVD! We will use these results to help understand the false negatives of TSVD next.

False positives of TSVD TSVD does not report any false positives. Every bug reported by TSVD is a true bug — a thread-safety violation can and is already triggered by TSVD.

False negatives of TSVD Given the non-deterministic nature of TSVs and the tight testing budget TSVD faces, TSVD inevitably has false negatives. Since it is impractical to know the ground truth about how many TSVs exist in our benchmarks[3], we use the 79 bugs discovered by 4 tools at the end of accumulated *200* runs in Figure 3.6 as our best-effort ground truth. We analyzed the 26 bugs that were missed by TSVD in its merely *2* testing runs and put them into three categories:

(1) Near-miss false negatives. For 19 bugs, the two racing operations execute close to each other only under rare schedules (e.g., a resource usage and a resource de-allocation). In most runs, there is a long time gap between them, like more than a few seconds. Consequently, the near-miss tracking in TSVD did not identify them as a dangerous pair and hence injected no delays to expose the bug in two runs. After running TSVD for 50 runs, 15 out of these 19 bugs were caught.

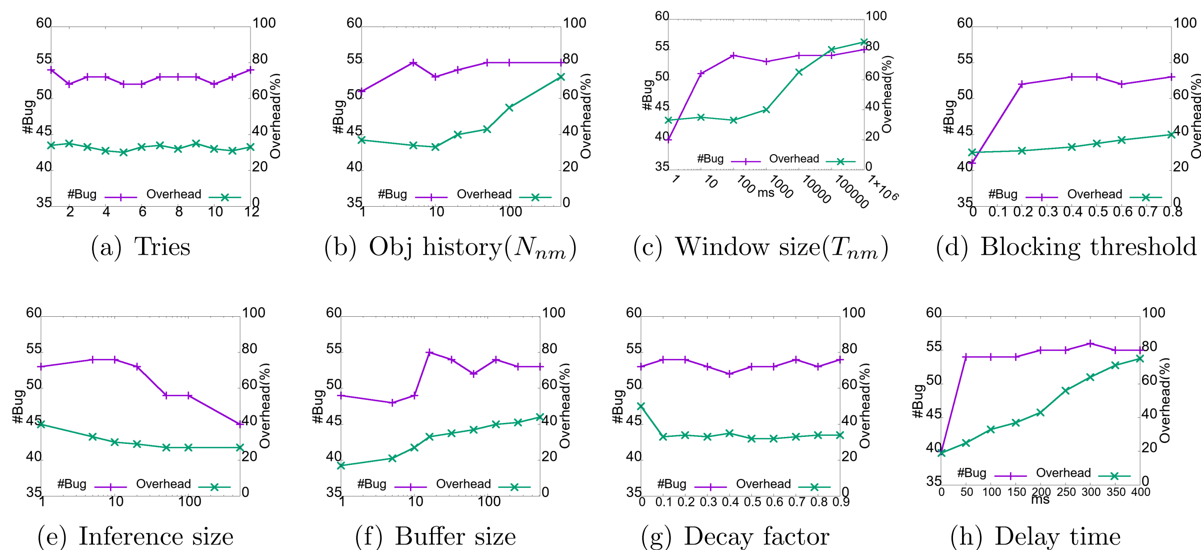
(2) Happens-before inference false negatives. For 2 bugs, TSVD's HB inference mistakenly treats two concurrent operations as happens-before ordered, and hence lost the opportunity of triggering the bug. Even after 50 runs, these 2 bugs still cannot be detected by TSVD.

(3) Delay-injection false negatives. For 5 bugs, the timing during the two testing runs happens to be that the injected delay was not long enough to trigger the bug. After running

3. Note that, before the use of TSVD, few TSVs were known in the code base tested by TSVD, which is exactly the motivation behind TSVD.

a couple of more runs, these bugs were all discovered by TSVD.

Finally, given the design goal of TSVD, it will naturally miss timing bugs that are not TSVs. For example, a recent study of real-world incidents in Azure software [108], a similar set of software projects that TSVD experiments target, showed that many more incidents are caused by timing bugs than that in traditional single-machine systems and about half of these timing bugs are related to concurrent accesses to persistent data. The current prototype of TSVD only focuses on in-memory data structures and hence cannot detect these persistent-data bugs. Future research can extend the idea of TSVD to detect other types of timing bugs.



(a) Tries (b) Obj history(N_{nm}) (c) Window size(T_{nm}) (d) Blocking threshold

(e) Inference size (f) Buffer size (g) Decay factor (h) Delay time

Figure 3.7: Sensitivity analysis of various parameters of TSVD.

3.4.4 Evaluating TSVD parameters

We now use the 1000-module small benchmark suite for parameter-sensitivity experiments.

Probabilistic nature of TSVD Since TSVD injects delays probabilistically, one may wonder how much the results of TSVD may vary across different tries. Figure 3.7(a) shows

the number of detected bugs and the overhead of TSVD across 12 tries. In every try, TSVD uses the same default parameter setting and is applied to two consecutive runs, just like that in Table 3.2. As we can see in the figure, the results indeed vary across tries, but only slightly: the number of detected bugs varies from 52 to 54 and the overhead varies between 30% and 35%, with a median of 53 bugs and 33% overhead.

Near-miss tracking parameters TSVD uses two parameters for tracking near-misses: the number N_{nm} of recent accesses that TSVD keeps for each object and the physical time window T_{nm} that TSVD considers two accesses as a near miss. As shown in Figure 3.7(b) and (c) (note log-scale of the x-axis), roughly speaking, both bug count and overhead increase with both parameters. Using too small a value (e.g., $N_{nm} = 1$ or $T_{nm} = 1ms$) misses many bugs because of not identifying dangerous pairs. TSVD's default values of $N_{nm} = 5$ and $T_{nm} = 100ms$ finds almost all the bugs, with small overhead. Larger values do not significantly increase bug count, but increases overhead (especially for N_{nm}).

HB inference parameters TSVD uses two parameters for HB inference: a causal-delay blocking threshold δ_{hb} and an inference window of k_{hb} accesses (Section 3.2.4). Figure 3.7(d) shows the effect of varying δ_{hb} from 0 to 0.8. A value too small like 0 infers many non-existing HB relationships, and hence misses many bugs; a larger value has stricter constraints in inferring HB relationship. As shown, the overheads and bug counts do not change much beyond TSVD's default value of 0.5. Figure 3.7(e) shows the effect of varying k_{hb}. A larger value translates to more HB relationship, reducing the number of dangerous pairs, and eventually the number of bugs and overhead. Too large a value drastically reduces the bug count. TSVD's default value of $k_{hb} = 5$ gives a sweet spot between bug count and overhead.

Buffer size of concurrent phase detection Figure 3.7(f) indicates that the overhead and the number of detected bugs both grow with the size of the global history buffer — with

	# bug			
	Total	Run1	Run2	overhead
TSVD	53	42	11	33%
No HB-inference	45	36	9	84%
No windowing in near-miss	46	35	11	143%
No concurrent phase detection	54	42	12	61%

Table 3.3: Removing one technique at a time from TSVD

a large buffer, TSVD may mistakenly treat sequential operations as concurrent and generate dangerous pairs that do not lead to any bugs; yet, with a small buffer, real concurrent operations can be mistakenly treated as sequential. TSVD uses a default size of 16 that gives a good trade-off between overhead and bug count.

Delay injection Figure 3.7(g) shows the impact of decaying delay injection probability at each TSVD point. A particularly bad configuration is when the factor is 0, meaning that TSVD injects delay in all occurrences of these TSVD points without any decay. This configuration would introduce too much overhead: for 3% modules, the overhead of zero decay was more than 100% (maximum overhead we observed was 6600% for one module!). These modules use TSVD points repeatedly and frequently (e.g., they are inside loops). Figure 3.7(h) shows the impact of the amount of delay TSVD injects at one trap point. As expected, longer delay increases runtime overhead, but also creates more opportunities for conflicting operations to overlap in time. TSVD uses 100ms as the default value.

Effectiveness of various TSVD techniques Table 3.3 shows how TSVD performs when one of its core components is disabled at a time. (In "No windowing", TSVD treats conflicting accesses by different threads *in the entire history* as near-misses.) As the results show, HB-inferencing and windowing are the most crucial techniques for finding bugs—without them bug counts drop from 53 to 45 and 46. Windowing is the most important factor in reducing overhead—without it, overhead increases from 33% to 143%. Overall, all the techniques are

```
Async Task<T> ClientStatusUpdate(int clientID,
        Status s){
        ...
        GlobalStatus[clientID] = s;
}
```

(a) Device Manager

```
Parallel.ForEach(hostlist,
     delegate(string host){
    ConfigLevel cl = GetConfigLevel(host);
    configureCache[host] = cl;
    ...
} );
```

(b) Network Validation

Figure 3.8: Examples

needed for TSVD's effectiveness.

3.4.5 TSVD CPU/Memory Consumption

To understand the detailed resource consumption of TSVD, we ran every unit test in the Small benchmark suite with and without TSVD, while recording the largest memory usage and average CPU usage in each run. Across all the unit tests, the median increase on maximum memory is 17% and the median increase on average CPU utility is 82%. The extra memory is mainly used for keeping the near-miss pairs and the access history of every thread-unsafe object. The extra CPU utility is mainly due to our instrumentation that forces all async functions to run asynchronously, as explained in Section 3.3. In comparison, without TSVD, the .NET optimization makes many async functions run synchronously, using much fewer cores.

3.4.6 Examples of bugs found by TSVD

Device Manager As shown in Figure 3.8(a), a device manager uses a Dictionary `GlobalStatus` to maintain the status of every client, where client ID is the key and the client status is the value. The device manager has a thread responsible for listening from multiple clients. Whenever the manager receives a message from a client, this listening thread will create an asynchronous task shown in Figure 3.8(a) to update the status of the corresponding client, and continue its listening. A concurrent write violation on the Dictionary class could happen when two clients send messages at similar time, which could then cause two concurrent execution of Line 4 in the figure and hence two concurrent Dictionary-set operations. As a result, the `GlobalStatus` Dictionary could get silently corrupted.

Network Validation Figure 3.8(b) shows another example of concurrent- write violation on a Dictionary class, which has a very different code pattern from the first example. When a network service starts up, a validator needs to verify the configuration information of every host, which involves reading a host's configuration information (Line 3) and storing it to a `configureCache` Dictionary for further verification. To speed up this process, the validator uses a `Parallel.ForEach` primitive (Line 1) to parallelize the validation for different hosts. The `Parallel.ForEach` automatically generates multiple concurrent threads and when some of these threads concurrently execute Line 4, a concurrent-write violation occurs to corrupt `configureCache`.

Production-Incident Example This is a bug about two threads trying to sort a list at the same time in a production run. The sorting result of an unprotected list is undetermined when two threads are doing that concurrently. This undetermined behavior propagated and finally caused the service to go down for several hours. TSVD can reproduce this bug without any prior knowledge and help the developers reduce the debugging effect.

Project	LoC	# tests	# run	# TSV	overhead
ApplicationInsights [24]	67.5K	934	2	1	15.31%
DataTimeExtention [42]	3.2K	169	1	3	18.51%
FluentAssertion [58]	78.3K	3076	1	2	8.89%
K8s-client [94]	332.3K	76	2	1	11.79%
Radical [135]	96.9K	965	1	3	1552.13%
Sequolocity [148]	6.6K	209	1	3	2.97%
Stastd [155]	2.5K	34	2	1	9.72%
System.Linq.Dynamic [157]	1.2K	7	1	1	41.39%
Thunderstruck [160]	1.1K	52	1	2	3.33%

Table 3.4: TSVD results on open source projects.

3.4.7 TSVD on Open Source Projects

To evaluate whether TSVD can be used beyond Microsoft, we applied TSVD to 9 open source C# projects (Table 3.4). Without any existing C# bug benchmark suite, we searched Github using "race condition" keyword and identified these 9 that contain (1) confirmed bug reports about TSVs on standard library APIs and (2) developer-written test cases.

Using exactly the same parameters as before, TSVD successfully detects and triggers all the TSVs under test in at most 2 runs. For all but 3 projects, TSVD detects these bugs using the *original* test cases released with the buggy software — if TSVD was used, these bugs would have been caught before code release. For Thunderstruck, TSVD detects one TSV that was not part of the original bug report.

We also evaluated the performance of TSVD on *all* the test cases. The overhead is mostly <20%, consistent with earlier performance results. Two projects incur large overhead, as they have many short-running tests (¡1 ms). The average slowdown for their tests is actually less than 400ms.

3.4.8 More

After working on TSVD, I realized a small portion of synchronization can be automatically understanded by the delay injection. But we also need the complete specification of synchronization in order to accurately perform analysis on concurrent software. For that, I had the

third chapter.

CHAPTER 4
UNSUPERVISED SYNCHRONIZATION-OPERATION INFERENCE

4.1 What are synchronization behaviors?

To effectively pinpoint synchronizations, we will use a set of properties and hypotheses that capture common behaviors of synchronizations. The properties represent fundamental assumptions that every synchronization should satisfy; the hypotheses reflect how *most* synchronizations are *typically* designed and used. They work together to support our inference.

The goal of this Chapter is to identify synchronizations in the application *without* understanding the semantics of underlying framework, library, or operating system that implements them. We define synchronization as any instruction or operation in the application that enforces a happens-before relation across threads. In this Chapter, we consider every synchronization, acquire or release, to take one of these forms: a read of a heap variable; a write to a heap variable; an invocation or entry point of an API or method; and an exit of an API or method. For example, the invocation of a thread-creation API and the entry point of the specified delegate method of the child thread form a pair of release and acquire that we aim to identify. Note that the actual implementation of the threading library or framework that enforces this happens-before relation is irrelevant to SherLock.

Property: Read-Acquire & Write-Release. Not every operation has the capability to release or acquire. Among memory-access operations, a heap read does not change system states and hence cannot be a release synchronization; on the other hand, a heap write cannot perceive what is going on in the system, and hence cannot be as an acquire synchronization. Similarly for method-related operations, the invocation of an application method can block

the caller till some condition is satisfied, while the exit of an application method may satisfy such conditions. Consequently, we associate a method's exit with a release and a method's invocation as an acquire, but never the other way around. We also enforce that a release synchronization cannot be an acquire and vice versa.

In addition to these properties, we add a set of hypotheses which are *soft* constraints — they are satisfied by most synchronizations most of the time, but not always.

Hypothesis: Mostly Protected. Most, if not all, conflicting accesses in a mature software should be synchronized. Consequently, given the observation in Figure 4.1.a, we could hypothesize that there probably exists at least one acquire among a_1, a_2, and a_3, which form the *release window*; and that there probably exists at least one release among b_1, b_2, and b_3, which form the *acquire window*.

Hypothesis: Synchronizations are Rare. In most software, synchronizations should constitute a small portion of all operations—most methods' invocations and exits, and most heap accesses will not cause threads to block or wake up. Further more, within any acquire/release window as illustrated in Figure 4.1, it is unlikely for the same synchronization to occur for many times — a non-synchronization like reading a popular variable or the invocation of a popular API could occur for many times, but a synchronization typically does not.

This hypothesis well complements the mostly-protected hypothesis, as the latter can be easily, yet incorrectly, satisfied by identifying all operations that ever appear in an acquire/release window as synchronizations.

Hypothesis: Acquisition-Time Mostly Varies. Intuitively, how long a thread needs to wait during an acquire varies a lot at run time. For example, when a thread acquires a lock, it could get the lock immediately if no one else holds the lock or take a long time if

many threads are competing for the lock. Specifically, SherLock applies this hypothesis to every method: if every execution of a method m takes roughly the same amount of time, the invocation of m is unlikely to be an acquire. This hypothesis helps identify acquires.

Hypothesis: Mostly Paired. Given the strong semantic connection between a release and its corresponding acquire, they are often defined in a paired or clustered way in well-maintained software. Specifically, if the read of a variable `C::v` is used for acquire synchronization, the corresponding release synchronization is very likely the write of the same variable `C::v`. As for methods, if a method of a class C is involved for release synchronization, its corresponding acquire is often a method that belongs to the same class C. For example, in C#, invoking `Monitor.enter` is an acquire, and its corresponding release synchronization is the exit of method `Monitor.exit` from the same system class `Monitor`.

This hypothesis helps identify a release, once its corresponding acquire is identified with high confidence, vice versa.

4.2 How to facilitate interesting behaviors?

Challenges Most of the hypotheses discussed in Section 4.1 are about dynamic software behavior. Given the non-determinism of concurrent software's dynamic behavior, we may not draw inference conclusions based on just one run.

For example, for any pair of conflicting accesses `a` and `b`, the physical time gap between them likely varies across runs and hence which operations appear in their acquire window and release window vary from run to run. When `a` and `b` happen to execute far away from each other, the acquire window and the release window will contain too many operations to be useful for synchronization inference. On the other hand, if `a` and `b` happen to execute close to each other, with few operations in between, such a behavior would be extremely useful and will be referred to as interesting behavior below.

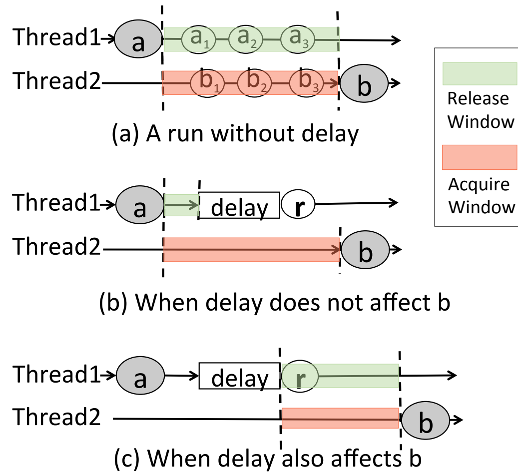


Figure 4.1: Acquire/release windows for synch. inference

However, a straw-man approach that simply re-executes the program many times is ineffective. Considering the huge interleaving space of a program, such a passive approach may never observe interesting behavior even after many runs.

To better handle this challenge, SherLock actively injects delays at strategic locations based on its inference from earlier runs to facilitate more interesting software behavior to occur.

Delay injection At high level, in every run, SherLock injects delays around those top synchronization candidates based on previous runs, so that the software's reaction towards these delays can either strongly support or strongly dispute existing inference results, allowing true synchronizations to surf up after a small number of runs (1–3 runs in our evaluation).

The exact strategy is illustrated in Figure 4.1. Imagine that SherLock has inferred `r` to be the most likely release synchronization between conflicting accesses `a` and `b` based on the observations collected so far, as shown in Figure 4.1 (a). In the next run, SherLock would inject a delay right before `r`.

This delay injection will produce valuable feedback no matter how the execution reacts. If, as shown in Figure 4.1 (b), this delay in the thread of `a` fails to cause a cascading delay in

the thread of `b`, we can conclude that `r` is actually not the release coordinating `a` and `b`, and that the real release, if exists, should be between `a` and `r` — a much smaller release window than the initial window between in `a` and `b` in Figure 4.1 (a).

On the other hand, if, as shown in Figure 4.1 (c), the delay manages to propagate to the other thread, we can trust the current inference results more. Furthermore, we can confidently refine the acquire window to be between `r` and `b`, also a smaller window than the initial `a`—`b` window in Figure 4.1 (a).

Note that our delay injection and delay-propagation observation is similar to TSVD [102], which uses delay injection to identify thread-safety violations and uses delay-propagation to infer happens-before relation between conflicting thread-unsafe API calls. In contrast, the goal of delay injection in SherLock is to refine the acquire/release windows and delegates the actual inference of synchronizations and implied happens-before relations to the Sovler.

This is because delay injection can only help refine the inference process but cannot replace it. If there are too many release candidates, there will be too many delays injected, which not only takes long time but also makes it difficult to judge whether a delay has propagated. All the properties and hypotheses discussed in Section 4.1 help the Solver to identify a small number of highly likely release candidates. Without them, delay injection alone can hardly discover real release and acquire synchronizations. We will explain the exact implementation of SherLock Perturber in Section 4.3.3.

4.3 SherLock

We now describe the design and implementation of various components of SherLock: the Observer, the Solver, and the Perturber. We also discuss how they work together.

4.3.1 Observer

Operations to trace SherLock instruments a given application binary to trace two types of operations during run time. First, it traces read/write operations that may form conflicting-access pairs useful for Mostly-Paired hypothesis. The operations include (1) read from or write to heap variables (e.g., public fields of a class), (2) getter and setter methods of public properties of a class, and (3) invocations of read/write APIs of thread-unsafe libraries (e.g., `List.Add()`).[1] Second, SherLock traces potential synchronizations including accesses to heap variables and entry and exit of methods. For application methods, SherLock instruments entry and exit points of their implementations. For library or system API calls, SherLock instruments immediately before and after the call sites.

Log-entry content At run time, SherLock records the following information for an operation: (1) timestamp, (2) thread ID, (3) operation type: read, write, method entry, or method exit; (4) field name and its memory address for each read/write operation, and (5) method name and parent object id for each method entry/exit operation.

Forming acquire/release windows In theory, we could identify every pair of conflicting accesses from the execution log and report an acquire/release window like the one in Figure 4.1.a. However, that would form overwhelmingly large number of windows. Particularly, when two conflicting accesses are far away from each other, there are too many operations between them to serve as useful hints.

Consequently, SherLock uses a physical time window `Near` to filter out less useful pairs (1 second by default). For every pair of conflicting accesses a and b that are nearby (i.e., $(T_b - T_a) \leq$ `Near`, assuming a is before b without losing generality), SherLock extracts all the operations in the log that executes between T_a and T_b as release candidates (if they are from

1. In the current prototype, SherLock instruments well documented thread-unsafe C# library APIs in System.Collections.Generic namespace.

the thread of a) or acquire candidates (if from the thread of b).

Note that a static code location may execute multiple times and hence form many acquire/release windows, particularly when it is inside a loop. Not to be overwhelmed with very similar windows from the same pair of static locations, SherLock sets a upper bound (15) for the number of windows that one location pair can form. After reaching the upper bound of a location pair, SherLock ignores subsequent windows for it.

4.3.2 Solver

The goal of the solver is to infer likely synchronizations from the observations collected from many runs. The key idea is to treat this as a probabilistic inference problem [38, 112]. Intuitively, if operations often appear to separate a pair of conflicting operations at runtime, then the probability of these operations being synchronization is higher.

Next, we explain the details of our encoding, with every constraint and objective function term representing one property or hypothesis discussed in Section 4.1.

Variables SherLock encodes every synchronization candidate as a random variable, whose assigned probability indicates the likelihood of this candidate being a synchronization.

For field reads, SherLock generates random variables $read(f)^{acq}$ and $read(f)^{rel}$ to respectively represent the probability the operation is an acquire or a release. Field writes are treated similarly. SherLock identifies the variables with the fully-qualified type of the field (i.e., `ClassName ::FieldName`), and assumes that all dynamic instances behave the same. That is, if a field is used as a synchronization once, it is always used thus. This reflects how synchronization variables are typically used in practice. More importantly, doing so enables SherLock to easily generate multiple observations for the same variable, in one run or across multiple runs, greatly improving the efficacy of inference.

For every method invocation, SherLock defines two variables, $begin(m)^{rel}$ and $begin(m)^{acq}$.

Similar representations are used for method exits, $end(m)^{rel}$ and $end(m)^{acq}$. As in field accesses, SherLock associates all dynamic instances of a method to a single variable identified by its fully qualified type (i.e., `ClassName::MethodName`). SherLock additionally assumes that the synchronization behavior remains the same independent of the polymorphic types or parameter values, assigning various forms to the same underlying variables.

Constraints SherLock encodes the *Read-Acquire & Write-Release* property discussed in Section 4.1 as linear constraints that should never be violated. This encoding is straightforward, as we simply need to assign corresponding variables to be 0 (i.e., there is no chance for them to fulfill a specific type of synchronization).

$$\begin{aligned} \text{for any field, } read(f)^{rel} = 0,\ write(f)^{acq} = 0 \\ \text{for any method, } begin(m)^{rel} = 0,\ end(m)^{acq} = 0 \end{aligned} \tag{4.1}$$

In addition, SherLock encodes the *Single Role* assumption that any library API l is only used to serve one type of synchronization, either acquire or release, encoded as $begin(l)^{rel} + end(l)^{acq} \leq 1$.

Objective function The properties above are encoded as *hard* constraints that can never be violated. SherLock also uses many hypotheses, which are essentially *soft* constraints, in that they can possibly be violated, but we want such violations to be rare. To represent these soft constraints, the basic idea, as in previous work [38], is to use a relaxed constraint $C \leq \epsilon$ and instruct the solver to minimize an objective function that incorporates ϵ, instead of strictly requiring $C \leq 0$. Using this idea, SherLock encodes the hypotheses in Section 4.1.

Mostly protected. Given $a_1, a_2, \ldots, a_n$ in a release window w^{rel} and $b_1, b_2, \ldots, b_m$ in a corresponding acquire window w^{acq} (as in Figure 4.1), SherLock formulates the hypothesis that "there probably exists an release synchronization among a_1, a_2, $\ldots a_n$" and "there probably exists an acquire synchronization among b_1, b_2, $\ldots b_m$" as minimizing the following

two terms:

$$
\begin{aligned}
rel(w) = & max(0, 1 - \sum_i a_i^{rel}), a_i^{rel} \in w^{rel} \\
acq(w) = & max(0, 1 - \sum_i b_i^{acq}), b_i^{acq} \in w^{acq}
\end{aligned}
\tag{4.2}
$$

When at least one of a_1, a_2, ... and a_n is assigned a high probability of being an release, the top term would become 0; otherwise, it remains a positive number. The similar trend applies to the acquire probability assignment. Consequently, by minimizing the sum of all the *rel* and *acq* scores from all observed release and acquire windows, we can support the hypothesis that most conflicting accesses are protected. Note that, an operation o may have multiple dynamic instances in an acquire or a release window, but we always only subtract its corresponding probability variable once. Otherwise, the *rel* or *acq* term can be easily minimized without any variable having a close-to-1 synchronization-probability.

Synchronizations are rare. To encode the hypothesis that there are few synchronization operations in the program, SherLock simply adds a regularization term, which is the sum of all the variables, to the objective function. Minimizing it will minimize the number of synchronization operations.

$$
reg(v_i) = v_i \tag{4.3}
$$

To encode the hypothesis that a synchronization is typically not invoked frequently in any acquire/release window, SherLock adds the following penalty for a variable based on its average number of occurrence in every window that it appears in.

$$
rare(v_i) = 0.1 * (average\ occurrence\ time\ of\ v_i) * v_i \tag{4.4}
$$

We choose the coefficient to be 0.1 so that most $rare(v_i)$ is between 0 and 1, similar as the range of the regularization term above and the variation term below.

Acquisition time mostly varies. SherLock calculates the standard deviation and mean of every method m's duration. SherLock then adds the following term to the objective penalty function, which helps prioritize those methods that have high duration variation when identifying acquire synchronization.

$$var(m) = (1 - percentile(CV(duration(m)))) * begin(m)^{acq} \tag{4.5}$$

Here, coefficient of variation (i.e., standard deviation divided by mean) represents how much variation a method m's duration has. This term $var(m)$ compares m with all other methods. The higher the variation is, the less penalty, ranging from 0 to 1, we get when inferring m to be an acquire.

Mostly paired. To capture the hypothesis that release method often comes from the same class as its corresponding acquire method, SherLock introduces the following penalty for every class c that contains candidate operations, which is minimized to 0 when the number of acquire synchronizations in c equals the number of release synchronizations in c:

$$pair_c(c) = \left| \sum_{op \in c} op^{acq} - \sum_{op \in c} op^{rel} \right| \tag{4.6}$$

To capture the hypothesis that if the read of a field f is used to acquire, the write of f is often used to release, vice versa, SherLock introduces the following penalty score:

$$pair_f(f) = |read(f)^{acq} - write(f)^{rel}| \tag{4.7}$$

Overall objective function. Putting the above terms together, the overall penalty objective

function is the following:

$$
\begin{aligned}
\sum_{w}(rel(w) + acq(w)) + \lambda[\sum_{c} pair_c(c) + \sum_{f} pair_f(f) \\
\sum_{v} reg(v) + \sum_{v} rare(v) + \sum_{m} var(m)]
\end{aligned}
\tag{4.8}
$$

Here, λ is a trade-off knob. It determines the relative weight between Mostly-Protected hypothesis and all other hypotheses in our inference. By default, SherLock sets λ to be 0.2. We will evaluate different settings of λ in our evaluation section.

Solving & Result interpretation SherLock uses a state-of-the-art linear solver [1] to find an assignment to all the variables that collectively satisfies all the constraints and minimize the penalty computed by the objective function.

Given all the variable assignment, we then check which acquire-probability variables and release-probability variables are assigned 1, and identify corresponding operations as acquire and release synchronization accordingly.

An important point to note here that these system of equations do not have a *trivial* solution — say one that makes every operation a non-synchronization, or one that makes every write a release and every read an acquire. This is because we require not only that at least one variable in the acquire (release) window to be an acquire (release), but also that the number of synchronization should be minimized. This together prevents trivial solutions. This is an important property that allows our system of equations to have meaningful solutions without requiring bootstrapping with user annotations.

4.3.3 Perturber and Feedback across Runs

SherLock executes the target application multiple times (3 times per input in our evaluation). Across runs, observations are accumulated; new inferences are made; delays are injected

accordingly, which then facilitate new observations.

To accumulate the observation across runs, SherLock does not throw away any constraints or objective function terms obtained from previous runs. Instead, it keeps (1) adding new variables and corresponding constraints, if new synchronization candidates show up, (2) adding new objective function terms for every newly observed release window and acquire window, and (3) updating existing objective function terms, like the average occurrence of a candidate operation, the co-efficient of variance of a method's duration, etc. Since the variables in our linear constraint system correspond to static names of methods and fields instead of their dynamic instances, the number of variables is guaranteed to be bounded and correlating information across runs is straightforward.

A special type of observation gets accumulated is that, sometimes, SherLock could observe a *data race*. This occurs when SherLock observes a concurrent execution of two conflicting accesses with every operation in the acquire (release) window guaranteed to not be an acquire (release) synchronization. This can happen when either the acquire (release) window is empty or every operation in the window is a write (read) operation. When SherLock encounters such a data race between accesses a and b, SherLock remembers it and removes all the Mostly Protected penalty term associated with the acquire and release windows between a and b in all runs.

After every run, given the solver's updated inference results, SherLock Perturber injects a 100 milli-seconds delay right before every[2] dynamic instance of every operation that is currently considered as a release synchronization by the solver (i.e., no delay is injected for the first run). SherLock then checks whether the injected delay is propagated. Depending on that, the Perturber notifies the Observer to adjust the observed acquire window and release window accordingly, as illustrated in Figure 4.1 (b) and (c).

2. We also tried injecting the delay probabilistically, but did not see much difference in inference results.

ID	Name	LoC	#Stars	#Tests
App-1	ApplicationInsights	67.5K	306	1193
App-2	DataTimeExtention	3.1K	335	219
App-3	FluentAssertion	78.1K	1886	3729
App-4	K8s-client	332.4K	395	139
App-5	Radical	95.9K	33	798
App-6	RestSharp	19.8K	7363	92
App-7	Stastd	2.3K	125	34
App-8	System.Linq.Dynamic	1.1K	399	7

Table 4.1: Applications in benchmarks

4.4 Evaluation

4.4.1 Methodology

We implemented SherLock using Mono.Cecil [4] binary instrumentation framework, and evaluated SherLock on the latest versions (as of April 2020) of 8 open-source projects from Github. We run available unit tests of these projects for our dynamic monitoring. 7 of these projects (all but App-6 in Table 4.1) are from the benchmark suite set up by a recent C# concurrency-bug detection paper [102]. Only 2 projects from that suite were not evaluated here, because one closed its source code recently and one does not contain any multi-threaded unit tests. We also randomly picked one C# application, App-6 in Table 4.1, from the search results for "race condition" in Github. Table 4.1 shows the details of all these applications: they are all well maintained and reasonably popular based on the number of stars on Github; their sizes range from around one thousand lines of code to more than three hundred thousand.

We run the benchmark suite on a Windows 10 laptop, with Intel(R) i7-8750 CPU, 16G Memory. Our evaluation answers the following key questions: (1) What synchronization operations can SherLock identify? (2) How helpful are these inferred synchronization operations in data-race detection? (3) What false positives and false negatives did SherLock incur? and (4) How do different components and parameter settings of SherLock affect the inference results?

ID	Syncs	Data Racy	Instr. Errors	Not Sync
App-1	46	10	2	7
App-2	6	0	0	0
App-3	8	0	2	0
App-4	20	0	1	0
App-5	14	2	0	2
App-6	14	0	0	2
App-7	19	4	0	0
App-8	6	0	0	1
Sum	133 (122)	16	5	12

Table 4.2: SherLock inferred results after 3 rounds. The sum in the parentheses is the unique synchronizations across applications.

4.4.2 Overall results

Table 4.2 shows the results of running SherLock on our benchmarks. The table reports the results after 3 runs. Later sections will evaluate how the results vary with the number of runs. As the table shows, SherLock successfully identifies many real synchronizations. Of the 133 synchronizations identified, 122 are unique across the applications. Surprisingly, many of them are non-traditional synchronizations such as relying on finalizers to be called after the instruction that makes an object unreachable. We discuss more details and how the inferred synchronizations help data-race detection in Sec. 4.4.3 and 4.4.4.

As with other probabilistic inference techniques, SherLock makes 33 misclassifications shown in Table 4.2. 16 out of these 33 participate in true *data races* (forming 8 data race pairs). These include accesses that should be marked `volatile` to prevent memory consistency issues [17] as well as bugs due to missing synchronizations. For 5 cases, SherLock honed in on the right synchronization neighborhood but failed due to limitations of our current C# instrumentation infrastructure. The remaining 12 are instances where SherLock erroneously identified nonsynchronization operations as synchronizations. We discuss false positives and negatives in Section 4.4.5.

4.4.3 What synchronizations are inferred?

SherLock identified 122 unique synchronizations for the applications in our benchmarks.[3] Of these, 51 are release synchronizations and 71 are acquire synchronizations. We classify these synchronizations into 19 system-API-based synchronizations, 12 variable-based synchronizations, and 91 application-method-based synchronizations.

System-API-Based Synchronization

This class includes methods in libraries that provide synchronization primitives for applications. This includes classic methods such as `Monitor::Enter` and `Monitor::Exit` and specialized methods related to to asynchronous processing like `DataflowBlock::Post` and `DatalowBlock::Receive`. An example of the latter is shown in Figure 4.2.A. Here `::Post` is a release synchronization that happens before the entrance of an event handler and `::Receive` is an acquire synchronization that happens after the exit of the event handler.

If one would manually annotate synchronizations, these APIs are the simplest to do and this annotation effort can be amortized across multiple applications. Unfortunately, only a small percentage of synchronizations (19 out of 122) fall in this class. Moreover, 13 of these 19 API methods are used in only one application, validating a long tail of API-based synchronizations even in the small set of applications we study.

Variable-based synchronization

Four applications in our benchmarks use variable-based synchronizations, contributing to 12 out of the 122 synchronizations. These include 4 while-loop synchronization and 8 if-checking based synchronization. For example, Figure 4.2.B shows a variable-based synchronization inferred from application App-4. Here, `endOfFile` is a flag indicating if the file writing has

3. The Appendix lists the exact synchronizations SherLock inferred.

```
 //Example A @App-7
_block = CreateMessageParserBlock()
T1: _block.Post(Event e)
    _block.Receive()
T2: Messgaehandler(Event e){...}

//Example B @App-4
volatile bool endOfFile = false;
T1: this.endOfFile = true;
T2: while (!this.endOfFile){/*wait*/}

//Example C @App-2
ConcurrentDictionary<T,T> dayCache;
T1: dayCache.GetOrAdd(year,delegate d1);
T2: dayCache.GetOrAdd(year,delegate d2);

//Example D @App-7
task = new Task(Action a1)
task.ContinueWith(Action a2)
T1: Action a1(){...}
T2: Action a2(){...}

//Example E @App-1
T1: void TestInitialize(){...}
T2: void BasicStartOperationWithActivity(){...}
```

Figure 4.2: Inferred synchronization examples

finished. Thread $T1$ sets it to be `true` after flushing the buffer to file; Thread $T2$ uses a while-loop to wait for the flushing to finish.

Application-Method-based synchronization

This is by far the largest class of synchronizations, contributing to 91 out of the 122 inferred ones. In these cases, the application relies on happens-before relation on a method return or a method entrance for synchronization.

Applications can use a variety of mechanisms to enforce happens-before ordering of these methods. First, such ordering can be guaranteed by the language semantics. For example, C# ensures that all static fields are properly initialized. This enforces a happens-before relationship between the return of the static constructor to any use of the object. Similarly, C# ensures that finalizers on objects only run when the object is not reachable. Thus, the instruction that removes the last reference of an object happens before the beginning of the object's finalizer. SherLock inferred these relationships with no prior knowledge of the language semantics.

Method ordering can also be enforced by registering them as callbacks to system APIs. In Figure 4.2.C, two threads invoke `GetOrAdd` method on a concurrent dictionary. The delegate provided as a parameter to `GetOrAdd` executes when the specified key is not in the dictionary and is guaranteed to be atomic with respect to other delegates from concurrent calls to `GetOrAdd` on the same dictionary. This semantics guarantees a happens-before relation between the return of `d1` and the start of `d2` (or vice versa). Without understanding the involved semantics of the `GetOrAdd` method, SherLock identified that the starts and returns of the two delegates as synchronization.

Figure 4.2.D shows another such example. The program registers an action `a2` to continue with a predefined task `a1` using the `ContinueWith` API. This API guarantees `a2` to execute after `a1` independent of the threads executing these methods. Again, without understanding

ID	# True Data Races		# False Data Races	
	Manual_{dr}	SherLock_{dr}	Manual_{dr}	SherLock_{dr}
App-1	0	4	263	14
App-2	1	1	0	0
App-3	1	18	31	2
App-4	0	0	32	15
App-5	2	1	0	6
App-6	0	3	31	12
App-7	0	2	33	1
App-8	0	0	1	1
Sum	4	29	391	51

Table 4.3: SherLock vs. manual annotation in race detection

the semantics of `ContinueWith` mechanism in C#, SherLock infers the return of `a1` as a release synchronization and the start of `a2` as the corresponding acquire synchronization.

Some application frameworks enforce happens-before relations. As shown in Figure 4.2.E, `Microsoft.VisualStudio. TestTools.UnitTesting`, a popular testing framework, provides a `TestInitialize` method to set up test environments. This method is guaranteed to execute before any unit test, like the `BasicStartOperationWithActivity` test function in App-1. Here, SherLock correctly infers the return of `TestInitialize` as a release synchronization, and the start of all executed test methods as acquire without knowing its semantics or analyzing any code inside the testing framework.

4.4.4 How helpful are inferred synchronizations?

Synchronizations are crucial in reasoning about concurrent programs. We quantitatively evaluate the benefit of synchronizations inferred by SherLock by using them in a dynamic data-race detector that mimics FastTrack [56, 57].

Since the original FastTrack algorithm was implemented for Java applications, we re-implemented FastTrack for C# and created two variants. The first, referred to as Manual_{dr}, is equipped with a list of manually identified synchronizations. For every Java synchronization

tracked by FastTrack, we annotated corresponding C# synchronization API. We took care to support `volatile` variables, wait-notify synchronization, barriers, and happens-before relations from static initialization as reported in the paper [57] and from manual code inspection. The second, referred to as $\text{SherLock}_{\text{dr}}$, *only* uses the synchronizations inferred by SherLock. We compared the two versions when running all the unit tests of our benchmarks and manually inspecting error reports for true and false data races.

Table 4.3 shows that $\text{SherLock}_{\text{dr}}$ reports more true data races (29, compared to 4) and fewer false data races (51, compared to 391) than $\text{Manual}_{\text{dr}}$. Our manual inspection showed that all the false data races reported by $\text{Manual}_{\text{dr}}$ are due to missed synchronizations. For example, 323 out of the 391 false data races reported by $\text{Manual}_{\text{dr}}$ are related to not handling the numerous ways of creating and executing tasks in C#, like those from `TaskFactory`, `ThreadPool`, etc. $\text{SherLock}_{\text{dr}}$ eliminates most of these false positives by its inference. Improving tools like FastTrack this way with automatically inferred synchronizations is a key motivation of this work.

$\text{SherLock}_{\text{dr}}$ reporting fewer false data races than $\text{Manual}_{\text{dr}}$ is natural as the former uses more happens-before relations than the latter. What is surprising is that $\text{SherLock}_{\text{dr}}$ reports more true data races than $\text{Manual}_{\text{dr}}$. On investigating further, we believe this is because FastTrack guarantees only hold till the first data race report. It continues to report subsequent data races, but only in a best-efforts manner. If the first data race is a false one due to missing synchronization, the reduced quality of subsequent reports can prevent it from detecting true data races.

$\text{SherLock}_{\text{dr}}$ suffers less from this problem and thus reports more true data races. $\text{SherLock}_{\text{dr}}$ still reports 51 false data races. This is due to SherLock's failure to infer 12 synchronizations (shown under #Missed Sync column in Table 4.4), which we discuss below.

	SherLock	$\text{SherLock}_{\text{dr}}$	
	#False Sync.	#Missed Sync.	#False Races
Instr. Errors	5	3	17
Double Roles	2	1	15
Dispose	5	4	11
Static Ctr.	4	2	3
Others	2	2	5
Total	17	12	51

Table 4.4: Breakdown of false positives/negatives.

4.4.5 What caused false positives and false negatives?

As shown in Table 4.2, SherLock inferred 33 incorrect synchronizations. Of these 16 arise from 8 data race pairs. On manual inspection, we found that 2 of these data races resulted in test assertion failures, implying that these data races are harmful.

Table 4.4 summarizes the remaining 17 false positives. Every false synchronization inferred corresponds to a true synchronization that SherLock missed. These misclassifications result in $\text{SherLock}_{\text{dr}}$ reporting false data races due to missed synchronizations, which are both shown in the table. We only report the false negatives identified during our manual inspection of data-race reports from $\text{Manual}_{\text{dr}}$ and $\text{SherLock}_{\text{dr}}$. There might be other synchronizations that we might have missed.

5 false positives resulted from errors in our instrumentation framework. Our instrumentation uses heuristics to identify and skip compiler-generated and library code. The heuristics mistakenly skipped some application methods and did not expose them to SherLock. These resulted in 3 missed synchronizations and 17 false data-race reports. We report this category separately in Table 4.2 as this is an error in our implementation but not in our algorithm. We plan to rectify these errors in future versions.

2 false positives arise due to SherLock's Single-Role assumption that acquire and release cannot occur inside the same system method. In C#, there are a few APIs that violate that

	# Inferred Sync. Ops.		
	#Correct	#Total	Precision
SherLock	122	155	79%
w/o Mostly are Protected	0	0	n/a
w/o Synchronizations are Rare	112	271	41%
w/o Acq-Time Varies	106	152	70%
w/o Mostly are Paired	101	158	64%
w/o Read-Acq & Write-Rel	100	152	66%
w/o Single Role	111	156	71%

Table 4.5: Inference with or without certain hypothesis

assumption. For example, `UpgradeToWriteLock` releases a reader lock and then acquires a writer lock all inside one API. This resulted in 1 missed synchronization and 15 false data-race reports. Future SherLock can try turning the Single-Role assumption into a soft constraint.

The remaining 11 false positives arose because of SherLock's inability to refine the acquire/release window effectively. These include failure to identify the acquire pair for object disposals (5), the release pair for static constructors (4), and other synchronization (2). For instance, dispose functions are often called during garbage collection which can execute at a much later time after the pairing release instruction that removes the last reference to the object. Since SherLock's delay injection does not control the garbage collection, it was not able to refine the windows.

4.4.6 More detailed results

How hypotheses helped. Table 4.5 shows how different hypotheses and synchronization properties have helped in reaching the SherLock inference results. All the numbers discussed in this sub-section and related tables and figures are about unique synchronizations inferred from 8 applications.

The Mostly-Protected hypothesis is the most crucial one. Without it, the Solver will simply decide that no synchronization exists in a program. Apart from it, the Synchronizations-

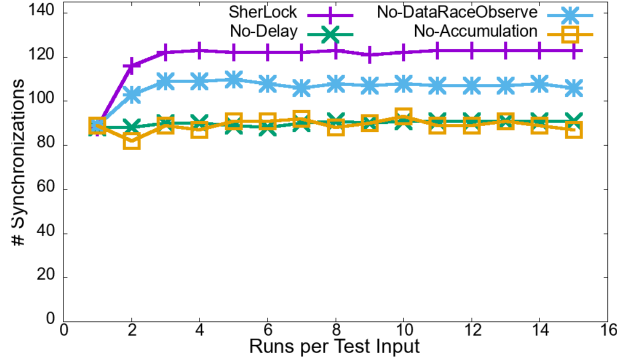


Figure 4.3: The numbers of correctly inferred unique synchronizations under different Perturber and feedback settings.

λ	0.1	0.2	0.4	0.6	0.8	1	5	10	50	100
#correct	118	**122**	115	111	111	110	76	67	29	19
#total	157	**155**	156	147	144	142	95	85	36	29

Table 4.6: Sensitivity of λ (numbers are the unique sums across 8 applications after running each test 3 times.)

are-Rare hypothesis is also crucial: without it, the precision of SherLock drops from 79% to merely 41%. It well complements the Most-Protected hypothesis to make sure that not too many operations are tagged as synchronization. The other hypotheses and properties are also helpful, as removing any one of them leads to fewer true synchronization identified.

How Perturber and multi-run feedback helped. Figure 4.3 evaluates several design decisions in our Perturber and feedback mechanism across runs. The worst performing schemes are when we do not accumulate constraints across runs at all (the yellow curve) and when we do not inject delays (the green curve). In both cases, the number of correctly inferred synchronizations drop from above 120 (SherLock) to around or lower than 90. The other decision that observes data races and removes corresponding Mostly-Protected terms from the objective function is also helpful (the blue line). We can also see that the number of correctly inferred synchronizations increases significantly for SherLock in the first three runs and becomes stable after that.

Parameter setting λ Our overall objective function (Equation 4.8) contains a parameter λ that balances the weight of Mostly-Protected hypothesis term and all other terms. Table 4.6 shows the results under different settings of λ. When the λ increases, the weight of Mostly-Protected hypothesis decreases and hence SherLock infers fewer synchronizations; when λ decreases, SherLock infers more to ensure that every conflicting pair is protected. SherLock uses 0.2 as the default setting.

Enhancing TSVD inference A recent thread-safety violation detector TSVD [102] infers happens-before relation between thread-unsafe API calls through delay injection and propagation check: when TSVD injects a delay at a thread-unsafe API call site a in thread 1 and observes this delay propagates to block thread 2, it infers that a happens before a thread-unsafe call site b in thread 2 right after the blocking period. This inferred happens-before knowledge is crucial for TSVD to save unnecessary effort in trying to expose thread-safety violation bugs among already synchronized calls. Note that, TSVD is not designed to pinpoint exact synchronizations and intentionally uses only simple heuristics to make quick inference, as low overhead is crucial for it. In this evaluation, we check if the synchronizations inferred by SherLock can help enhance TSVD happens-before inference.

We ran the open-source TSVD [3] for all applications in our benchmark suite.After 3 runs of every test input for every application, TSVD reports happens-before relation among 8 conflicting API-call pairs, with 7 of them truly synchronized (some applications do not call thread-unsafe APIs concurrently at all).By applying $\text{SherLock}_{\text{dr}}$ to the same set of benchmarks, 20 pairs of conflicting API-call pairs are identified as being truly synchronized—indeed, SherLock can be used to enhance TSVD in its happens-before inference and hence bug exposing.

Overhead The overall overhead of applying SherLock to one test run, including instrumentation, tracing, and solving, ranges from 24% to 800%. The average overhead across

all test cases is 278%, where tracing incurs 170% and solving incurs 94% overhead. In the default setting of running each test cases 3 times, the average overhead of using SherLock versus the baseline of executing the test cases 3 times without any instrumentation or delay, is 434%, where the delay injection introduces 156% overhead. Since we do not expect the current version of SherLock to be used in production, we consider the overhead of SherLock acceptable and have not worked on optimizing its various parts yet.

CHAPTER 5

RELATED WORK

5.1 Related Work

Automated Bug Fixing Motivated by the huge cost in bug fixing and its huge impact to software availability, much research has been conducted for automating patch generation recently [44, 86, 158, 166]. In addition to single-node concurrency bug fixing techniques discussed in Section 1.1.2, other auto-fixing techniques have also been proposed. Some of them focus on specific types of bugs that are unrelated to distributed timing bugs [61, 143]; some use program verification and synthesis techniques to find patches that fit a specific template (e.g., only change one variable or one operator in software) [33, 113]; and many techniques use heuristics to search software mutation space to find mutations that can pass all regression tests [97]. None of them suits the problem of fixing distributed timing bugs, where the program space is huge and the bug exists in timing, instead of computation logic.

Distributed Timing-Bug Detection Bug detection [30, 107, 109] and model checking tools [68, 92, 98, 105, 152] have been proposed to detect distributed timing bugs. They are all potential front ends to auto-fixing tools like DFix.

Note that, bug detection tools can help but can**not** replace bug fixing tools. As we can see in the design of DFix, even after the details of a bug are known, there are still many program analysis and reasoning required to produce a patch. It is desirable to relieve developers from such costly and error-prone effort.

Improving Distributed System Availability Program verification and auto-proving [72, 164] is a promising direction to build highly available distributed systems. Existing techniques cannot scale to large distributed systems with many protocols, and sacrifice performance greatly.

PAR [20] uses protocol-specific knowledge to design correct disk-failure recovery routines in cloud storage that is built upon replicated state machine (RSM) protocol. It does not help fixing our bugs, as none of them is from RSM protocol. Olive[150] describes an approach, lock with intent, that provides exactly-once semantics for a transaction of operations that work on certain type of global storage. It is very effective for fault tolerance of certain type of transactions, and can help fix some of our fault-timing bugs like AF1 – AF3, but cannot help other fault-timing bugs that go beyond storage problems and involve more complicated storage systems. These fault tolerance techniques do not help fix the message-timing bugs.

Data Race Detection Since the tolerance for false error reports is low in our context, we do not focus on static data-race-detection techniques [49, 53, 106, 170]. TSVD is closely related to dynamic active data-race detection techniques [129, 147], as discussed in Section 1.2.1 and Figure 1.5.

Dynamic passive detection techniques [56, 132, 146, 149, 168] perform happens-before analysis, lock-set analysis, or a combination of the two to predict whether a data race could have manifested. Recent work [76, 93, 153] infer more data races by generalizing beyond executions that are happens-before equivalent to the current execution. These techniques pay the runtime cost of monitoring synchronization operations and also suffer from false positives [90, 126, 147]. Several techniques have used sampling to reduce the runtime overhead [27, 91, 121], but like DataCollider [50], they need to repeatedly run the test many times to expose bugs.

Another type of passive detection tools like AVIO [114] and Bugaboo [115] catches concurrency bugs *when* they manifest at run time. Since they do not *predict* or *expose* bugs that have not manifested yet, which TSVD does, they do not need to analyze happens-before relation but are also fundamentally unsuitable for the build and test environment TSVD targets.

Race detection was studied for asynchronous event-driven programs such as Android apps

and Web applications [74, 75, 119, 139]. There, the key challenge is to model and analyze the causality between asynchronous requests and responses, which incurs huge run-time slowdowns. TSVD instead automatically infers happens-before relationships. While not the focus of this Chapter, techniques behind TSVD can work for applications targeted by these works.

Systematic Testing Systematic testing techniques discover concurrency errors by driving the program towards different possible interleavings [29, 32, 59, 63, 73, 99, 124]. These techniques are either guided by the need to cover all interleavings within some bound [59, 63, 99, 124], or a coverage notion [29, 73], or provide a probabilistic guarantee of finding bugs [32]. While these techniques can find general concurrency bugs, they are not designed to discover most bugs in a small number of runs. Instead, TSVD is specifically designed for finding TSVs within the first few runs and without paying the cost of controlling the thread scheduler.

Generating Unit Tests Tools were proposed to synthesize unit tests to help expose concurrency bugs inside library functions [39, 133, 141, 142]. Given multi-threaded libraries that are expected to allow safe concurrent invocations, these tools synthesize concurrent method-call sequences to help expose bugs inside the library implementation. They are orthogonal to TSVD: TSVD focuses on improving the efficiency of exposing thread-safety violation bugs through existing unit tests of software that uses thread-unsafe libraries.

Timing hypothesis Snorlax [87] reproduced and diagnosed in-production concurrency bugs leveraging a coarse interleaving hypothesis. Snorlax experiments showed that the time elapsed between events leading to concurrency bugs ranges between 154 and 3505 micro-seconds, based on which Snorlax could reproduce concurrency bugs without fine granularity monitoring and recording. This coarse-interleaving hypothesis and TSVD's near-miss tracking look at

different aspects of concurrency bug's timing window characteristics — Snorlax believes the timing window is not as small as people used to think, and TSVD believes conflicting accesses in small windows are more likely to lead to real bugs — and leverage the characteristics in different ways.

Causality inference Past work has used run-time trace analysis to infer happens-before relationship among message sending/receiving operations or distributed system tasks in the context of system-performance analysis [18, 40] and network-dependency analysis [37]. Due to the different usage scenarios, the exact inference algorithms differ between TSVD and these previous tools. Previous tools all require a large number of un-perturbed system traces, and statistically infer happens-before relationship based on whether two operations never flip execution order [40] or always execute with a nearly constant physical-time gap in between [37]. Different from previous tools, TSVD works in the unique testing environment where the system trace contains much perturbation introduced by TSVD; TSVD also faces the unique goal of finding bugs in a small number of runs, and hence cannot wait until a large number of traces become available. Consequently, TSVD's happens-before inference is uniquely designed based on observing delays in each testing run.

Happens-before identification and inference Previous research worked on automatically identifying custom synchronization that uses synchronization variables. They [36, 161, 167, 171] apply static program analysis to identify specific program structures, like spin loop [161], shared-variable predicated control dependency[36, 167], and queues [171]. The applied static analysis is complicated and focuses on specific types of synchronization. In contrast, SherLock can identify various types of synchronization without sophisticated static analysis.

Some recent work infers the existence of happens-before relationship based on dynamic observation [37, 40]. Mystery-Machine [40] infer task A happens-before task B if A always executes before B in millions of production runs. Orion [37] infers happens-before relation

between two network operations whose time gap is nearly constant in thousands of runs. Furthermore, by considering multiple hypotheses and properties of synchronizations and feedback-based delay injection, SherLock does not require many runs to reach the inferring results.

Role inference in program analysis Specification inference for program analysis is a well studied problem [38, 112, 138]. SUSI [138] trains a supervised support vector machine to identify the privacy roles in Android APIs. Merlin[112] and Seldon[38] use probabilistic inference for identifying source, sink, and sanitizer specification for identifying security vulnerabilities.

SherLock is inspired by these works but applies them to the new setting of synchronization inference. This obviously requires new set of hypotheses and properties related to synchronizations, and leads to many differences in respective constraint systems: previous work [38, 112] uses non-linear constraints or linear constraints collected from many applications, while SherLock only uses linear constraints collected from the target application; SherLock updates its constraint system after every run, instead of statically [38, 112]. SherLock focuses on unsupervised inference while prior work requires to be bootstrapped with manually provided annotations.

Others Decades of research has been conducted on analyzing concurrent programs, detecting concurrency bugs [31, 56, 74, 75, 89, 114, 119, 131, 145, 165, 172], and tuning performance of concurrent programs [2, 22, 40, 43, 103, 156]. SherLock is orthogonal to all these work and help them to achieve better analysis accuracy and capability with greatly decreased effort in annotating synchronization operations.

The hypothesis that mature software is mostly correct has also been used in statistical bug detection [48, 70], failure diagnosis [104], and inferring likely program invariants [51, 114]. SherLock shares similar philosophy with these work, but is solving fundamentally different

problems and using completely different designs from them.

CHAPTER 6

FUTURE WORK

After working on concurrency for five years, there are something that I feel may be interesting for future work.

Fixing There are basically two general approaches for concurrency bugs fixing. One is fantastic program change without correctness promise. The other is removing the concurrency from programs. For the first approach, under the context of fixing bug, programmers care more about if the bug is fixed without hurting other components. If the change is beautiful is not first priority. Furthermore, the beauty claimed by the fixing tool is far different with what programmers think is beautiful. For the second approach, removing the concurrency is force the program to "old fashion"/sequential. Somehow, it sounds like the best way of avoiding bugs in program is not writing program at all to me. People actually want concurrency but end with bug with careless implementation. Automatically filling this careless while maintaining the concurrency is really interesting to me.

Detection For concurrency bug detection, people have explore the optimization for Vector-Clock algorithm for decades. The current state-of-art solution is deep enough but difficult to apply. TSVD is one approach to infer/sense the synchronization in programs based on cascading delay. But the cascading delay is not the only observation to infer synchronization. Also cascading effect is not easy to judge. There is no easy solution to judge cascading effect with low false positive and false negative. MysteryMachine is a good start that we can observe relative order. But it still have challenges like running too many times. I feel there are definitely other observations we can use to sense the synchronizations. Even under the cascading effect observation, one question question is the relation between injected delay and observed correct effect. We observe nothing if no delay. If too many delay, we also cannot observe anything due to delay cancellation.

Role Inference After reading Seldon [38], I realized one interesting problem is we can boost the program technology with data-driven approach. Classical program analysis explores various implementation of the same purpose (e.g assignment for point analysis). Due to the complexity of modern language, the speed of handle language feature is much slower than language birth. Eventually, program analysis cannot catch up with the programming language. But the important insight is that various implementation is always for the one same purpose. And usually this purpose is easy to observe during runtime.

A Good Question None of the questions I mentioned before is easy to solve. And some of them are actually hard and probably no solution at all. But a good question is usually something other people cannot even imagine it is possible to solve. Good luck and have fun!

CHAPTER 7

CONCLUSION

Synchronization has many format and implementation in the current mature software system. The variants in synchronization bring unique challenges to the program analysis. New programming framework or environment introduce new synchronization primitives. These new primitives require human to provide details documents and explanation about the synchronization effect. Even for the traditional synchronizations, the language introduces different format for easy-using and performance. To handle these synchronization is really challenging.

Fixing them encounters unique synchronization and scope challenges. DFix explores using carefully designed rollback and fast-forward to handle bug-triggering timing and fixes distributed timing bugs without introducing new bugs.

TSVD is a new thread-safety violation detection technique . Being part of an integrated build and test environment, TSVD provides a push-button no-false-positive bug detection for .NET programs that use complex multi-threaded and asynchronous programming models with a wide variety of synchronization mechanisms. TSVD provides a starting point for exploring the wide design space of active testing and resource-conscious delay injection design.

ShherLock made the first step in using unsupervised inference to identify synchronizations. The result shows that SherLock is effective in identifying various types of synchronizations using its well designed set of hypotheses and synchronization properties, assisted by its perturbation and feedback accumulation across runs.

These tools are just the starting pint, Future work can further explore the design space for synchronization understanding.

REFERENCES

[1] Flipy: linear solver. https://pypi.org/project/flipy/. Accessed: 2020-8-9.

[2] Ibm thread and monitor dump analyze for java. https://www.ibm.com/support/pages/ibm-thread-and-monitor-dump-analyzer-java-tmda. Accessed: 2020-8-9.

[3] Microsoft tsvd. https://github.com/microsoft/TSVD. Accessed: 2020-8-9.

[4] Mono.cecil. https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/. Accessed: 2020-8-9.

[5] Overview of synchronization primitives. https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives. Accessed: 2020-8-9.

[6] Zookeeper 1653. https://issues.apache.org/jira/browse/ZOOKEEPER-1653, 2013. Accessed: 2013-11-26.

[7] Hbase 10090. https://issues.apache.org/jira/browse/HBASE-10090, 2017. Accessed: 2017-09-16.

[8] Mapreduce 4637. https://issues.apache.org/jira/browse/MAPREDUCE-4637, 2017. Accessed: 2017-09-16.

[9] Zookeeper. https://zookeeper.apache.org/, 2017. Accessed: 2017-09-16.

[10] Zookeeper 1270. https://issues.apache.org/jira/browse/ZOOKEEPER-1270, 2017. Accessed: 2017-09-16.

[11] Btrfs rollback. https://ramsdenj.com/2016/04/05/using-btrfs-for-easy-backup-and-rollback.html, 2018. Accessed: 2018-04-30.

[12] Google protocol buffer. https://developers.google.com/protocol-buffers/docs/reference/overview, 2018. Accessed: 2018-05-01.

[13] Hadoop asyncdispatcher. https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/yarn/event/AsyncDispatcher.html, 2018. Accessed: 2018-05-01.

[14] Hadoop versionproto. https://blog.woopi.org/wordpress/files/hadoop-2.6.0-javadoc/org/apache/hadoop/yarn/proto/YarnServerCommonProtos.VersionProto.html, 2018. Accessed: 2018-05-01.

[15] Hbase protobase. http://www.grepcode.com/file/repository.cloudera.com/content/repositories/releases/org.apache.hadoop/hadoop-yarn-common/2.3.0-cdh5.1.4/org/apache/hadoop/yarn/api/records/impl/pb/ProtoBase.java?av=h, 2018. Accessed: 2018-05-01.

[16] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. Repairing event race errors by controlling nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering*, pages 289–299. IEEE Press, 2017.

[17] Sarita V Adve and Mark D Hill. Weak ordering—a new definition. *ACM SIGARCH Computer Architecture News*, 18(2SI):2–14, 1990.

[18] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 74–89. ACM, 2003.

[19] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[20] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Protocol-aware recovery for consensus-based storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 15–32, Oakland, CA, 2018. USENIX Association.

[21] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 151–167, 2016.

[22] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *EuroSys*, 2017.

[23] Aleksandr Mikunov. Rewrite MSIL Code on the Fly with the .NET Framework Profiling API. `https://www.cnblogs.com/WCFGROUP/p/5136703.html`.

[24] ApplicationInsights. Broadcast processor is dropping telemetry due to race condition. `https://github.com/Microsoft/ApplicationInsights-dotnet/issues/994`.

[25] Bazel. Bazel: Build and test software of any size, quickly and reliably. `https://bazel.build/`.

[26] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 2(2):131–152, 1996.

[27] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada*, 2010.

[28] Nikita Borisov and Robert Johnson. Fixing races for fun and profit: How to abuse atime. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, 2005.

[29] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, Chicago, IL, USA*, 2005.

[30] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 458–472, 2017.

[31] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[32] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, 2010.

[33] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 121–130. IEEE, 2011.

[34] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[35] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[36] Feng Chen, Traian-Florin Serbanuta, and Grigore Rosu. jpredictor: a predictive runtime analysis tool for java. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, 2008.

[37] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, volume 8, pages 117–130, 2008.

[38] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 760–774, 2019.

[39] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017.

[40] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, 2014.

[41] Csharpmm. Standard ECMA-334 C# Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`.

[42] DateTimeExtensions. Resolve a random race condition. `https://github.com/joaomatossilva/DateTimeExtensions/pull/86`.

[43] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. *ACM SIGPLAN Notices*, 49(10):291–307, 2014.

[44] Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. A novel fitness function for automated program repair based on source code checkpoints. 2018.

[45] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[46] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[47] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, 48(6):321–332, 2013.

[48] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[49] Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles , SOSP , Bolton Landing, NY, USA*, 2003.

[50] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *OSDI*, volume 10, pages 1–16, 2010.

[51] Michael Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[52] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. Cloudbuild: Microsoft's distributed and caching build service. In *SEIP*. IEEE - Institute of Electrical and Electronics Engineers, June 2016.

[53] Facebook. A tool to detect bugs in Java and C/C++/Objective-C code before it ships. `https://fbinfer.com/`.

[54] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. ACM.

[55] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1987.

[56] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[57] Cormac Flanagan and Stephen N Freund. The fasttrack2 race detector. Technical report, Technical report, Williams College, 2017.

[58] FluentAssertion. Race condition in SelfReferenceEquivalencyAssertionOptions.GetEqualityStrategy. `https://github.com/fluentassertions/fluentassertions/issues/862`.

[59] Pedro Fonseca, Rodrigo Rodrigues, and Björn B Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, 2014.

[60] Dennis Michael Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. 2006.

[61] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 359–373, 2017.

[62] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[63] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[64] Patrice Godefroid. The soundness of bugs is what matters. In *PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools*, BUGS'2005, 2005.

[65] Patrice Godefroid and Nachiappan Nagappani. Concurrency at Microsoft – an exploratory survey. Technical report, MSR-TR-2008-75, Microsoft Research, May 2008.

[66] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[67] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC, 2014.

[68] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 265–278. ACM, 2011.

[69] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure recovery: When the cure is worse than the disease. In *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*, 2013.

[70] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[71] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[72] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[73] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[74] Chun-Hung Hsiao, Satish Narayanasamy, Essam Muhammad Idris Khan, Cristiano L. Pereira, and Gilles A. Pokam. Asyncclock: Scalable inference of asynchronous event causality. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, Xi'an, China*, 2017.

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

[76] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[77] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: recording local executions to reproduce concurrency failures. In *PLDI*, 2013.

[78] IBM. Main page - walawiki. `http://wala.sourceforge.net/wiki/index.php/Main_Page`, 2017.

[79] Jb Evain. Mono.Cecil. `https://github.com/jbevain/cecil`.

[80] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Notices*, volume 46, pages 389–400. ACM, 2011.

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

[82] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, volume 12, pages 221–236, 2012.

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

[84] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[85] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[86] René Just, Chris Parnin, Ian Drosos, and Michael D Ernst. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 287–297. ACM, 2018.

[87] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[88] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. *ACM SIGPLAN Notices*, 47(4):185–198, 2012.

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

[90] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS , London, UK*, 2012.

[91] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: crowdsourced data race detection. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP, Farmington, PA, USA*, 2013.

[92] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. NSDI, 2007.

[93] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[94] kubernetes-client. fix a race condition. `https://github.com/kubernetes-client/csharp/pull/212`.

[95] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[96] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

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

[98] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *OSDI*, pages 399–414, 2014.

[99] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, 2014.

[100] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ACM SIGPLAN Notices*, volume 51, pages 517–530. ACM, 2016.

[101] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S Gunawi, and Shan Lu. Dfix: automatically fixing timing bugs in distributed systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 994–1009, 2019.

[102] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[103] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Pcatch: automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[104] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[105] Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2009.

[106] Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

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

[108] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 155–162. ACM, 2019.

[109] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 419–431. ACM, 2018.

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

[111] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. Wids checker: Combating bugs in distributed systems. 2007.

[112] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. *ACM Sigplan Notices*, 44(6):75–86, 2009.

[113] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *ACM SIGPLAN Notices*, volume 47, pages 133–146. ACM, 2012.

[114] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, San Jose, CA, USA*, 2006.

[115] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[116] Brandon Lucia and Luis Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 39–50, 2013.

[117] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[118] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 378–393. ACM, 2015.

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

[120] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proceedings of POPL*, pages 378–391. ACM, 2005.

[121] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland*, 2009.

[122] IHS Markit. Businesses losing $700 billion a year to it downtime, says ihs. `http://news.ihsmarkit.com/press-release/technology/businesses-losing-700-billion-year-it-downtime-says-ihs`, 2016.

[123] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[124] Madan Musuvathi and Shaz Qadeer. Chess: Systematic stress testing of concurrent software. In *International Symposium on Logic-based Program Synthesis and Transformation*, pages 15–16. Springer, 2006.

[125] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. June 2007.

[126] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI), San Diego, California, USA*, 2007.

[127] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen S Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *OSDI*, pages 1–15, 2012.

[128] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[129] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 25–36. ACM, 2009.

[130] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[131] Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *PLDI*, 2012.

[132] Eli Pozniansky and Assaf Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, March 2007.

[133] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 521–530, New York, NY, USA, 2012. ACM.

[134] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Acm sigops operating systems review*, volume 39, pages 235–248. ACM, 2005.

[135] Radical. MessageBroker internal subscription(s) list is not thread safe. `https://github.com/RadicalFx/Radical/issues/108`.

[136] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 368–383, Berlin, Heidelberg, 2010. Springer-Verlag.

[137] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, New York, NY, USA, 2012. ACM.

[138] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125. Citeseer, 2014.

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

[140] Michiel Ronsse and Koenraad De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[141] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, 2014.

[142] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[143] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, pages 277–287. IEEE Press, 2012.

[144] Anirudh Santhiar and Aditya Kanade. Static deadlock detection for asynchronous c# programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[145] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[146] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[147] Koushik Sen. Race directed random testing of concurrent programs. *ACM Sigplan Notices*, 43(6):11–21, 2008.

[148] Sequelocity.NET. Race condition on TypeCacher. `https://github.com/AmbitEnergyLabs/Sequelocity.NET/pull/23`.

[149] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71. ACM, 2009.

[150] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 501–516, Savannah, GA, 2016. USENIX Association.

[151] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D Keromytis. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News*, 37(1):37–48, 2009.

[152] Jiri Simsa, Randy Bryant, and Garth A Gibson. dbug: Systematic evaluation of distributed systems. In *SSV*, 2010.

[153] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

[154] Ian Sommerville. *Software Engineering: (Update) (8th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[155] statsd.net. Race conditions when updating guage value. `https://github.com/lukevenediger/statsd.net/issues/29`.

[156] Aater M. Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS '09*, pages 253–264.

[157] System.Linq.Dynamic. Fix the multi-threading issue at ClassFactory.GetDynamicClass. `https://github.com/kahanu/System.Linq.Dynamic/pull/48`.

[158] Akito Tanikado, Haruki Yokoyama, Masahiro Yamamoto, Soichi Sumi, Yoshiki Higo, and Shinji Kusumoto. New strategies for selecting reuse candidates on automated program repair. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, volume 2, pages 266–267. IEEE, 2017.

[159] Theregister. Aws's s3 outage was so bad amazon couldn't get into its own dashboard to warn the world. `https://www.theregister.co.uk/2017/03/01/aws_s3_outage/`, 2017.

[160] Thunderstruck. Race condition in ConnectionStringBuffer singleton. `https://github.com/19WAS85/Thunderstruck/issues/3`.

[161] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, 2008.

[162] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *ASPLOS*, 2011.

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

[164] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices*, volume 50, pages 357–368. ACM, 2015.

[165] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Low-level detection of language-level data races with lard. In *ASPLOS*, 2014.

[166] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for software-defined networks. In *NSDI*, pages 719–733, 2017.

[167] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *OSDI*, 2010.

[168] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 221–234. ACM, 2005.

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

[170] Sheng Zhan and Jeff Huang. ECHO: instantaneous in situ race detection in the IDE. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA*, 2016.

[171] Jiaqi Zhang, Weiwei Xiong, Yang Liu, Soyeon Park, Yuanyuan Zhou, and Zhiqiang Ma. Atdetector: improving the accuracy of a commercial data race detector by identifying address transfer. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 206–215, 2011.

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

[173] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 19–33. ACM, 2017.