UNDERSTANDING, DETECTING AND EXPOSING CONCURRENCY BUGS

BY

SHAN LU

B.S., University of Science and Technology of China, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Associate Professor Yuanyuan Zhou, Chair
Professor Sarita Adve
Assistant Professor Darko Marinov
Professor Josep Torrellas
Professor Mark Hill, University of Wisconsin at Madison

Abstract

Software is pervasive in our daily lives. Unfortunately, software bugs can severely affect the dependability and security of software systems. Among all types of software bugs, the concurrency bug is one of the most troublesome and important. Concurrency bugs widely exist in concurrent programs. They are difficult to detect and diagnose because of their unique non-determinism. In the real world, concurrency bugs have caused several disasters in the past and are generating increasingly severe problems in recent years with the prevalence of multi-core hardware and concurrent programs. Facing the challenge of concurrency bugs, this thesis proposes effective concurrency bug detection and concurrent program testing approaches based on a comprehensive characteristics study of real-world concurrency bugs.

This thesis makes three main contributions. The first contribution is a comprehensive characteristics study of real-world concurrency bugs. A good understanding of real-world bugs is always the foundation for addressing the software bug problem. This dissertation conducts the first comprehensive empirical study of concurrency bug patterns, manifestation conditions, and fix strategies based on a large number of concurrency bugs sampled from widely used open source C/C++ server/client applications. This characteristics study provides many motivation and guidelines for concurrency bug detection, testing and programming language design.

The second main contribution is the proposal of novel techniques to automatically infer programmers' synchronization intentions and detect important types of concurrency bugs. A fundamental problem in concurrency bug detection is determining what types of interleavings are intended and what are not. This thesis proposes novel techniques to automatically infer two types of important synchronization intentions: single-variable atomicity intentions and multi-variable correlations from source code and program execution. Based on these intention inference approaches, two bug detection tools, AVIO and MUVI, are designed to detect concurrency bugs that are common yet not well addressed: atomicity violation bugs and multi-variable concurrency bugs. Experiments have shown that AVIO and MUVI can accurately detect many bugs that existing techniques cannot detect. Previously unknown bugs in widely used open source concurrent programs are also detected.

The third main contribution is along the lines of exploring concurrent programs' interleaving space and exposing concurrency bugs. This thesis presents a hierarchy of interleaving coverage criteria. This hierarchy includes seven interleaving coverage criteria that are designed based on different concurrency bug models and provides guidance to interleaving space exploration. Guided by the coverage criteria research, a testing framework, CTrigger, is built to expose atomicity violation bugs. CTrigger's testing space, called unserializable interleaving space, is carefully designed to balance its complexity and bug-exposing capability. Within this testing space, CTrigger uses trace analysis to identify feasible and rare unserializable interleavings; it uses low-overhead execution perturbation to exercise these interleavings and effectively expose atomicity violation bugs. Experiments have shown that CTrigger can expose real-world atomicity violation bugs 100–1000 times faster than the common practice stress testing. In addition, CTrigger can reliably repeat the bugs that are exposed once 300 to more than 60,000 times faster than stress testing, which will greatly expedite the software failure diagnosis process.

To my parents, Yemiao Lu and Xintian Shi,
and my husband, Yi-Ting Chou

Acknowledgments

My first and biggest thanks go to my adviser, Professor Yuanyuan Zhou (YY). This dissertation would not exist without her. YY taught me not only how to do research, but also how to be a good person and a confident woman. From her, I got endless research inspiration and guidance. From her, I learned how to write papers, make presentations, manage time, collaborate with people, be organized and be strong. YY always believes in me when I do not have confidence in myself. YY always encourages me to move beyond my comfort zone to reach my full potential. I cannot imagine a better adviser than YY. I know I can never repay YY for what she has given to me. I just hope that I can make her proud in the future and I hope I am able to give something of what she gave me to my own students.

I would like to thank the other members of my dissertation committee — Professor Sarita Adve, Professor Mark Hill, Professor Darko Marinov and Professor Josep Torrellas — for their invaluable comments and constructive criticisms that greatly improved my thesis. It is really my honor to have them on my committee and I learned a lot from them.

Thanks also go to all members in the Opera research group. I want to thank Pin, Feng, Zhenmin, Zhifeng and Qingbo for setting up great examples for me, making me proud of the group from the very beginning, and helping me through my junior years. I want to thank Joe for his research enthusiasm, for tolerating my bad English, and for the coffee he bought me after our ISCA rejection :). I want to thank Soyeon for working with me through so many deadlines. My graduation would probably have been postponed if Soyeon was not such a good collaborator. I want to thank Lin. I would be much grumpier if I couldn't chat with her. I want to thank Weihang, Spiros, Chongfeng, Xiao, Chengdu, Rini, and Weiwei for working with me on several projects, offering me delightful discussion and unremitting assistance. I would also like to thank every one of the Opera members who have not been mentioned yet, including Vivek, Yoann, Ding, and Zuoning. You guys and YY have really made Opera Group a comfortable home for me. I will never forget our brainstorms (ice-creams), reading groups, games of Frisbees, end-of-semester parties, etc.

I would also like to thank many other people in the University of Illinois computer science department. Thank you, Professor Sam King, Professor Craig Zilles, Professor Laxmikant Kale, Professor Vikram Adve, and Professor Madhusudan Parthasarathy for giving me encouragement

and suggestions. Thanks to Sheila Clark for helping me through various administration stuff. Thanks to many friends in the system/architecture groups. It is especially nice to see you guys in the printing rooms at midnight before deadlines. Thanks for being around and making the past five years a happy research journey.

I also own my happy graduate student life in Urbana-Champaign to many people. I want to thank Jing Yu, Hong Cheng, Jing Jiang, Chao Liu, Changhao Jiang, Bin Tan, Chun-cheng, Chih-wei, Yun-tien, and Pei-hsi for many happy parties and dinners. I want to thank the Fighting Illini basketball and football teams :). Thanks for giving me the two most sensational seasons in my sports-fan life (2005 and 2007). Thanks for giving me the most inspiring game in my life (2005 Fighting Illini vs. Arizona). I feel deeply sorry to be aligned against Fighting Illini in the coming years. Allow me to say 'GO ILLINI!' one more time.

I would like to thank my dear parents for their unconditional love and support. It was my parents who first exposed me to science and encouraged me all the way along. My father has spent enormous efforts on my study. He taught me how to reason and how to be persistent. He encouraged me to pursue my dreams through hard work and careful planning. My mother always lets me know that I can come back to them for support no matter what happens. Dad and Mom, I am so lucky to have been your daughter. I love you so much.

I would also like to thank my grandparents, uncles, aunts, parents-in-law, and dear cousins. Thanks for giving me tremendous support, bless, encouragement and love through all these years.

Finally, I would like to thank my husband. I am grateful to the Illinois Ph.D. program that brought Yi-Ting and me together. I am afraid of nothing as long as I am with Yi-Ting. Yi-Ting, I love you!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software bugs severely affect the dependability of software systems and are one of the most important problems in computer science research. Among all types of software bugs, concurrency bugs are one of the most troublesome. Concurrency bugs widely exist in concurrent programs. Their unique non-determinism property makes them very difficult to detect and diagnose. In the past, they caused several disasters in the real world such as the Northeast Blackout of 2003. Nowadays, the urgency to address the concurrency bug problem is becoming even more important with the prevalence of multi-core hardware.

Facing the challenge of concurrency bugs, this dissertation proposes effective concurrency bug detection and concurrent program testing approaches based on a comprehensive characteristics study of real-world concurrency bugs.

1.1 Motivation

1.1.1 Concurrency Bugs

Software is pervasive in our daily life. Unfortunately, most software contains bugs. Previous studies have shown that software bugs cause about 25–35% of system down time [MS00, PBB+02, Sco98] and 50% of security vulnerabilities [CER]. Recent survey also shows that software bugs cost US economy sixty billion dollars annually (i.e., 0.6% of GDP) [Nat02].

Among all types of software bugs, concurrency bugs are one of the most troublesome types. Concurrency bugs are synchronization problems among the concurrent tasks in concurrent programs. Concurrent programs are software systems that conduct concurrent execution of multiple tasks. These tasks can interact with each other through either shared memory or message passing. This dissertation looks at shared memory based concurrent programs, which are common in current desktop and server environments, and assumes the Sequential Consistency (SC) memory model [Lam79].

Concurrency bugs widely exist in concurrent programs. The reason is that most programmers are trained to write sequential, instead of concurrent, programs. With the sequential thinking habit,

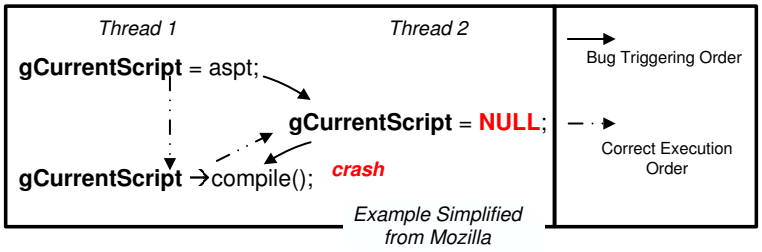


Figure 1.1: A typical concurrency bug. (This figure is simplified from a real bug in Mozilla Application Suite. Figure 4.1 shows more details of this bug.)

programmers are used to thinking about one thing at a time and, hence, can easily make mistakes when they reason about multiple concurrent execution components at the same time.

Figure 1.1 shows an example of typical concurrency bugs. This example is simplified from a real bug in Mozilla. It actually looks perfectly correct, if we consider the codes executed by thread 1 and thread 2 separately. For example, the code executed by thread 1 is a common pattern in sequential programs: store a value into a pointer variable, retrieve and dereference this value later. This is exactly how programmers with sequential thinking habit make mistakes: they easily forget that other threads, such as thread 2, could execute concurrently with thread 1 and access the shared pointer variable `gCurrentScript`. In this example, when thread 2 nullifies the pointer variable in the middle, the program crashes due to a null pointer dereference. From this simple example, we can see that sequential-thinking programmers could easily introduce concurrency bugs when they write concurrent programs. They will need a lot of help to address the concurrency bug problem.

Making things even worse, once introduced into the program, concurrency bugs are difficult to get rid of. Different from sequential programs and sequential bugs, concurrent programs and concurrency bugs have a unique and notorious property: non-determinism. The execution results of concurrent programs are determined not only by inputs, but also by the non-deterministic thread interleavings[1]. With the same input (a bug-triggering one), a concurrency bug may manifest in some runs and disappear in many other runs. For example, in Figure 1.1, even with the same input, the program execution may or may not fail, depending on which interleaving is followed (as shown by the solid lines and dotted lines in the figure). This non-determinism introduces a lot of trouble to in-house testing: a program with concurrency bugs can pass a thorough testing composed of a large number of inputs. As a result, many concurrency bugs sneak into production runs and manifest at users' site under special interleavings. In addition, the non-determinism in the

[1] An interleaving is an execution order among memory accesses from multiple threads. This dissertation considers interleavings following the sequential consistency memory model and does not consider additional interleavings that are possible under more relaxed memory consistency models.

manifestation of concurrency bugs also makes it inconvenient for users to report bugs and makes it hard for developers to diagnose software failures.

1.1.2 Impacts of Concurrency Bugs in the Real World

The challenging concurrency bug problem has existed for many years, but has never been well addressed.

In history, concurrency bugs have caused several disasters. In the 1980s, a concurrency bug in Therac-25, a radiation therapy machine, caused radiation overdoses and killed at least five patients, with more patients severely injured [LT93]. In 2003, a concurrency bug in the alarm system of GE energy management software finally left 50 million people in the northeastern America without power and cost around 6 billion dollars of financial loss [Sec].

Recently, the concurrency bug problem has become more severe. This change originates from a big shift in computer hardware: multi-core machines have replaced single-core machines as mainstream. This hardware change is due to the many inherent problems of single-core machines, such as the power wall problem and the memory wall problem. This change pushes parallel computing out of the high-performance computing community into the daily life of every computer users. In the meantime, it raises big problems for the software community. Existing concurrency bugs will manifest more frequently because multi-core machines provide many more interleaving opportunities to programs. The more frequent manifestation will yield more damages in practice. In addition, many new concurrent programs will enter the mainstream software community in order to leverage the multi-core resource. These new concurrent programs will inevitably produce many new concurrency bugs in our software.

1.2 Approaches to Address Concurrency Bugs

1.2.1 Directions to Address the General Software Bug Problem

Many different approaches have been proposed to address the software bug problem. Most of them are along one of the following lines.

Bug avoidance and software verification *Bug avoidance* includes techniques that help developers to write correct programs. Good programming languages and programming environments can effectively avoid some programming mistakes and decrease the total number of bugs in our software. *Software verification* includes techniques that try to prove the correctness of a program. Verification techniques can provide great confidence to developers when their results are positive.

Both bug avoidance and software verification techniques are frequently used in the early stage of software development. However, since making mistakes is human nature, it is impossible for programmers to always write perfectly correct programs, and it is the task for other avenues to address the remaining problems.

Software testing and bug detection These are two independent and also complementing directions to combat software bugs. *Testing* exercises the software and tries to identify conditions under which the software would fail. *Bug detection* analyzes source code or execution in order to figure out what and where the problem is inside a piece of software. These two directions can benefit from each other. Many bug detection techniques rely on testing techniques to expose bugs before they can analyze the problem. On the other hand, good bug detection tools can help developers to tell more accurately whether or not a testing has failed. Both techniques are critical and widely used in practice, consuming more than 30% of software companies' development resources [SDT].

Recovery and bug fixing *Recovery* techniques try to limit the damages caused by manifested software bugs and restore the software system to correct tracks. *Bug fixing* usually happens after a bug is pinpointed. Developers expend a lot of manual effort to fix a bug. Some times, this process can get feedback from software testing, especially regression testing, and bug detection.

Understanding bugs Finally, in order to develop good approaches along each of the above directions, good *understanding* of software bugs cannot be overlooked. The characteristics study of real-world bugs is the traditional way to achieve such an understanding.

Among the above discussed directions, this dissertation focuses on understanding, detecting, and exposing (software testing) concurrency bugs. The next section summarizes the state-of-the-art approaches to understanding, detecting and exposing concurrency bugs. More details along all directions will be discussed in Section 2.

1.2.2 State of the Art in Understanding, Detecting, and Exposing Concurrency Bugs

A good understanding of software bugs can provide useful guidelines and motivation for fighting software bugs from every direction. In the past, many empirical studies [CYC+01, OWB05, SC92, Z. 06] on general software bugs (not specific to concurrency bugs) have provided a lot of insights into memory bugs, semantic bugs, and some general features of all software bugs. Unfortunately, few studies have been conducted specifically to concurrency bugs. Recently, researchers

realizing the importance of such a study carried out preliminary work on concurrency bug patterns [FNU03]. However, their observations were built upon programs that were intentionally made buggy by students for this study; this cannot represent real-world concurrency bugs. With the limited understanding of concurrency bugs, we are in great need of a characteristics study based on a large number of real-world bugs to provide knowledge about the different types of real-world concurrency bugs, the typical conditions that expose them, and how they were fixed by developers, etc.

Concurrency bug detection has been studied for a long time. Most of the previous work focuses on data race, which happens when two conflicting accesses from different threads access the same shared variable without proper synchronization. Although many race detection tools have been proposed [C$^+$02, EA03, NM91, OC03, SBN$^+$97, YRC05], race condition is neither sufficient nor necessary for a concurrency bug (Chapter 4 will discuss this in more detail). Some recent work [FF04, XBH05] has jumped out of data race to investigate atomicity violation bugs. This is a very inspiring starting point. However, existing atomicity violation bug detection techniques are far from complete. In the meantime, there are also other types of important concurrency bugs that are not addressed. Finally, since the manifestation of concurrency bugs is highly dependent on the timing of thread interleavings, dynamic concurrency bug detection tools are expected to have low overhead and generate small perturbation. Towards this goal, hardware support [Prv06, PT03, ZTZ07] for race detection has been proposed. More research along this line to support general low-overhead concurrency bug detection is desired.

How to effectively expose concurrency bugs is an open problem. It is much more difficult than exposing sequential bugs because it demands exploring not only the input space but also the huge interleaving space of concurrent programs. Many coverage criteria were proposed in the past to provide guidelines to the exploration of interleaving space. However, previous proposals are either too complicated [KT96, TLK92] or based on heuristics [EFN$^+$02]. In the real world, the common practice to expose concurrency bugs is simply to run a program with each input many times, which is very inefficient. Recently, several inspiring concurrency bug-exposing frameworks have been proposed [BFM$^+$05, MQ07, Sen08]. These works focus the testing on a small set of representative interleavings by carefully perturbing the concurrent execution. These proposals can help expose some types of concurrency bugs, such as deadlock bugs. However, they are less helpful in exposing some other important types of concurrency bugs, such as atomicity violation bugs. The perturbation designed in some of these frameworks cannot fully leverage the multi-core computation resource and would greatly slow down the testing process.

In summary, prior work has made a lot of progress on fighting concurrency bugs. However, we are still in great need of a better understanding of concurrency bugs, more concurrency bug detection tools that go beyond data races, and effective concurrency bug-exposing frameworks.

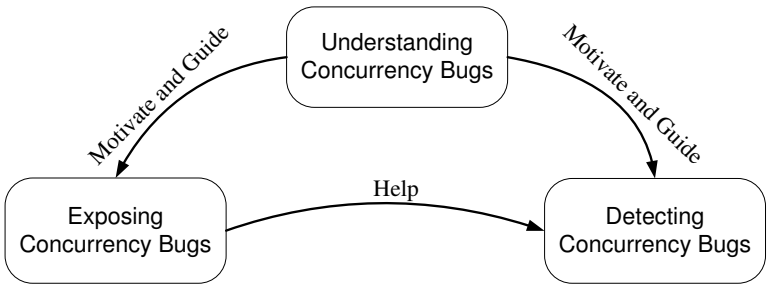


Figure 1.2: Interactions among the three components in this dissertation.

1.3 Dissertation Contributions

This dissertation works on three directions to address the concurrency bug problem: understanding real-world concurrency bugs through characteristics study, detecting important types of concurrency bugs that have not been well addressed before, and building a foundation towards practical and effective concurrent program testing. These three components of this dissertation interact and complement each other as shown in Figure 1.2. The work in this dissertation has already made impact: the characteristics study has helped more than 30 research groups to evaluate their software reliability research; part of the bug detection work (AVIO) is under technology transfer to Intel.

(1) Understanding concurrency bugs Addressing the concurrency bug problem requires approaches from many different directions and all these directions will significantly benefit from a deep understanding of real-world concurrency bugs.

This dissertation conducts a comprehensive characteristics study on a large number of real-world concurrency bugs collected from four widely used large open-source C/C++ applications: MySQL, Apache, Mozilla and OpenOffice. This study reveals many interesting findings about concurrency bugs' patterns (causes of concurrency bugs), manifestation conditions (conditions that are required to expose concurrency bugs), and fix strategies (how bugs are fixed by developers). It also directly motivates the following work on detecting and exposing concurrency bugs.

(2) Detecting concurrency bugs The goal of concurrency bug detection is to pinpoint what is wrong with the concurrent program execution, i.e., which (part of) interleaving is bad. To achieve this goal, a fundamental challenge is to figure out what types of interleavings are buggy (i.e., not intended by programmers). Different from all previous work, this dissertation proposes to automatically infer programmers' synchronization intentions and catch buggy interleavings that violate the intentions. Specifically, guided by the characteristics of real-world concurrency bugs, bug detection techniques for two types of important concurrency bugs are proposed as follows.

(a) AVIO: an invariant-based atomicity violation bug detection technique Atomicity violation bugs are one of the most common types of concurrency bugs. They are caused by violation to the atomicity of certain code regions. The biggest challenge in detecting them is to figure out which code regions are intended to be atomic. AVIO uses a novel solution to address this challenge. Specifically, a special type of program invariant that reflects programmers' atomicity intentions is proposed. AVIO automatically infers the atomicity intentions by learning invariants from correct execution. AVIO also has two designs to detect violations to the atomicity intentions (invariants): pure software design and hardware-supported design. The latter uses a simple extension to cache-coherence protocol and L1 cache to achieve negligible overhead. In the experiments with six real-world atomicity violation bugs in big server applications, AVIO can detect these bugs more accurately and more efficiently than existing concurrency bug detection tools. *AVIO is under technology transfer to Intel*.

(b) MUVI: a multi-variable concurrency bug detection technique Traditional race detection techniques focus on the synchronization among accesses to the same shared variable. Unfortunately, as indicated by the characteristics of real-world bugs, concurrent accesses to multiple correlated variables should also be synchronized. The problem of detecting this type of bug has never been addressed. In MUVI, a novel technique is proposed to automatically infer variable correlation based on source code analysis and data mining. Leveraging the inferred correlation information, the MUVI tool can effectively detect multi-variable concurrency bugs. In the evaluation of five real-world multi-variable concurrency bugs, MUVI correctly identified the root causes of four bugs, none of which could be identified by existing techniques. MUVI also found four previously un-known multi-variable concurrency bugs from a widely used open-source application: Mozilla. In addition, using the automatically inferred variable correlation information, MUVI found 39 (17 confirmed) semantic bugs that were caused by inconsistent update to correlated variables in the latest versions of Linux, MySQL, Mozilla, and Apache.

(3) Exposing concurrency bugs An effective way to expose concurrency bugs during software testing can greatly benefit concurrency bug detection and diagnosis. Unfortunately, the huge interleaving space makes it extremely challenging to exercise concurrent programs and expose concurrency bugs. Facing this challenge, this dissertation first proposes a hierarchy of interleaving coverage criteria and then builds a framework to expose atomicity violation bugs:

(a) A new hierarchy of interleaving coverage criteria The manifestation of a concurrency bug is always associated with certain interleavings. Unfortunately, in practice, the testing resource can afford exploring only a small portion of the whole interleaving space. Therefore, good coverage criteria are needed to guide the exploration of concurrent programs' interleaving space. This dissertation proposes a hierarchy of interleaving coverage criteria based on the characteristics of

real-world concurrency bugs. This hierarchy is composed of five layers and seven criteria. Each of these seven criteria is designed based on one concurrency bug model and can help select representative interleavings from a different perspective. These criteria together show a wide spectrum of design trade-offs between testing complexity (ranging from exponential to linear), and bug-exposing capability.

(b) CTrigger: a new testing framework to expose atomicity violation bugs Guided by the above study of coverage criteria, a testing framework called CTrigger is designed to expose atomicity violation bugs. CTrigger conducts testing through two phases: the first phase identifies representative interleavings and the second phase controls the execution to exercise the selected interleavings. Specifically, guided by one criterion that is proposed above, CTrigger focuses on unserializable interleavings that are highly correlated with atomicity violation bugs. Starting from unserializable interleavings, CTrigger trace analysis prunes out infeasible interleavings and ranks feasible interleavings based on their estimated occurrence probabilities, giving preference to rare interleavings. After this analysis, CTrigger carefully controls the concurrent execution to exercise the selected unserializable interleavings with a low overhead and expose hidden atomicity violation bugs. The evaluation with seven representative concurrent programs shows that CTrigger can expose the tested atomicity violation bugs 2–4 orders of magnitude faster than stress testing, the common practice. Some real-world server bugs do not manifest after days or even weeks of stress testing, yet can be exposed by CTrigger within only a couple of minutes. CTrigger also reliably repeats the bugs that are exposed once, mostly within 5 seconds, 300 to more than 60,000 times faster than stress testing.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes the background knowledge and previous work on concurrency bug characteristics study, concurrency bug detection, testing, and other related topics. Chapter 3 presents our characteristics study of real-world concurrency bugs. Chapters 4 and 5 focus on our two concurrency bug detection tools, AVIO and MUVI. Chapter 6 explains our design of a hierarchy of interleaving coverage criteria. Guided by this design, Chapter 7 presents the CTrigger framework that can effectively expose atomicity violation bugs.

The materials in some chapters have been published as conference and journal papers. The materials in Chapter 3 have been presented in [LPSZ08, LLQ+05]. The materials in Chapters 4 and 5 have been presented in [LPH+07, LTQZ06, LTQZ07], and some materials in Chapters 6 and 7 have been presented in [LJZ07, PLZ09].

Chapter 2

Background and Previous Work

This chapter discusses previous work on improving software dependability, with the focus on concurrent programs. Section 2.1 discusses previous work on understanding the characteristics of real-world software bugs. Section 2.2 and 2.3 discuss existing approaches on detecting concurrency bugs and testing concurrent programs. Finally, section 2.4 briefly discusses concurrent programming language design and multi-core deterministic replay works.

2.1 Understanding Bug Characteristics

In the past, many empirical studies on general software bugs (not specific to concurrency bugs) have been done. Their findings have provided useful guidelines and motivation for improving software reliability from different aspects, such as bug detection [CYC+01, SC92], fault tolerance [GKIY03], failure recovery [CC00], fault prediction and testing [OWB05], and so on. In a recent work [Z. 06], researchers also studied how the recent trends (e.g., the availability of commercial bug detection tools) affect the general bug characteristics such as distribution and fix time.

Unfortunately, few previous works have focused on concurrency bugs, probably because real-world concurrency bugs are hard to collect and analyze. For example, in a previous study [CC00], only 12 concurrency bugs were collected from three applications: MySQL, GNOME, and Apache. Under this situation, a previous concurrency bug pattern study [FNU03] had to ask students to purposely write concurrent programs containing bugs, which cannot well represent the real-world bug characteristics. Concurrency bug characteristics study using a large number of real-world bug samples is desired.

2.2 Detecting Concurrency Bugs

Many concurrency bug detection tools have been built. They can be categorized into three categories based on the different types of bugs they target: data race detection, atomicity violation detection, and deadlock detection. They are discussed one by one in the following sections.

2.2.1 Data Race Bug Detection

Data race is the type of concurrency bug that most previous concurrency bug detection work focused on. A data race occurs when two memory access instructions, at least one of which is a write, from different threads access the same shared variable without proper synchronization.

Three types of algorithms have been proposed to detect data races: lock-set algorithms, happens-before algorithms, and hybrid algorithms combining both lock-set and happens-before algorithms. The lock-set algorithm [C+02, DS91, EA03, NAW06, SBN+97] tracks the set of locks that are used to protect accesses to each memory location. It reports data race bugs whenever a memory location's lock set becomes empty. Happens-before bug detection [DS90, NM91, PK96] is based on Lamport's happens-before relation [Lam78]. Simply speaking, the happens-before partial order among memory accesses can be calculated based on synchronization operations. The happens-before algorithm reports data race bugs when there is no strict happens-before order between two conflicting memory accesses. Lock-set and happens-before algorithms each have their own advantages. The happens-before algorithm is more accurate, but can only detect races that occur during monitored runs. On the other hand, the lock-set algorithm can report some races that did not occur during the monitored runs. Unfortunately, the lock-set algorithm also incurs many more false positives than happens-before algorithm. Hybrid algorithms have been proposed [OC03, PS03, YRC05, NS07] to combine the advantages of these two algorithms. By conducting both lock-set and happens-before order analysis, hybrid algorithms can detect more bugs than the happens-before algorithm and introduce fewer false positives than the lock-set algorithm.

Apart from the false-positive and false-negative issue discussed above, performance is another critical issue. Many dynamic race detectors impact system performance by causing about 10 to 100 times slowdowns. This overhead can hardly be accepted even if the tool is used at the development site. Even worse, such a high overhead may prevent timing-sensitive concurrency bugs from happening. To address this problem, hardware support for race detection was proposed. Previous works have designed hardware extensions for fast happens-before race detection [MC91, Prv06, PT03] and fast lock-set race detection [ZTZ07]. These proposals can effectively improve the bug detection performance. However, their bug detection capability is similar to that of their software peers.

Overall, race detection has made a lot of progress in past years. However, it still suffers several fundamental problems. First, some data races are accepted by developers. It is hard to differentiate these races from true bugs, which leads to many false positives in bug reports. Second, it is hard to identify customized synchronization operations. This again leads to false positives. Finally, race-free does not mean concurrency bug-free. Even if a concurrent program has no data race,

it can still contain atomicity violation problems or other synchronization problems. This leads to false-negatives in concurrency bug detection. In addition, as will be discussed in Section 2.4, this false negative problem will be critical if advanced synchronization primitives, such as transactional memory, are adopted.

2.2.2 Atomicity Violation Bug Detection

Atomicity, also called serializability [1], is a property for the concurrent execution of several operations when their data manipulation effect is equivalent to that of a serial execution of them[2]. Atomicity violation bugs are introduced when programmers assume some code regions to be atomic, but fail to guarantee the atomicity in their implementation. Consequently, the assumed atomicity can be broken at run time and lead to program failure.

Many works [FQ03, SAWS05, WS05] have studied how to verify the atomicity of certain code regions. They use the state reduction theory of the right/left mover to detect atomicity violation upon specified code regions statically or dynamically. Recently, simpler definitions of atomicity have been discussed [VTD06, FM06] to ease the atomicity verification process. A limitation of these works is that they require programmers to specify all synchronization points, which is expensive, and all code regions that need to be atomic, which is impractical—if programmers can properly do this, they probably would not have introduced the atomicity violation bugs. This limitation is addressed to some extent by Atomizer [FF04], which gains knowledge of synchronization through the lock-set analysis [SBN+97]. In [BL02], *stale-value errors*, a subclass of atomicity violation bugs, are studied. Not requiring any manual annotation, SVD [XBH05] uses data dependency and control dependency to infer atomic regions. It provides an inspiring starting point for automatically inferring atomic regions. However, its inference mechanism still misses many potential atomic regions.

2.2.3 Deadlock Bug Detection

Deadlock occurs when threads circularly wait for each other to release the acquired resource (e.g., locks). Deadlock bugs have completely different properties than non-deadlock bugs such as those discussed earlier. In practice, many server applications simply use timeout-and-restart to detect and recover from deadlock. Lockset-based analysis [EA03, SBN+97] and speculation-based techniques [LELS05] are also proposed for more accurate deadlock detection.

[1] Specifically, serializability in this dissertation refers to view serializability.

[2] Atomicity has a slightly different meaning in database research community. The definition used in this dissertation follows the tradition in concurrent program research.

The techniques mentioned above are mostly dynamic techniques. There are also many static concurrency bug detection tools not mentioned. One category of static techniques leverages type systems [FF00, PFH06, AGEB08]. The other category is model checking [God97, HJM04, MQ07, QW04]. Compared with dynamic techniques, static bug detection does not have the problem of run time overhead and is not limited to program inputs. However, it usually has scalability problems and accuracy problems caused by pointer-alias issue, etc.

2.3 Testing Concurrent Programs

Software testing is a dominant technique that software developers heavily rely on to expose bugs before software release. Previous surveys have shown that about 30% of software development resources are spent on testing in software companies [SDT]. A lot of research has been done on coverage criteria design, test case selection, and testing framework building. This section focuses on previous work on testing concurrent programs. Some background of general testing will also be presented.

2.3.1 Software Testing and Coverage Criteria

A fundamental challenge in testing is the huge and sometimes infinite testing space, which includes the set of all program inputs, all environment configurations, all interleavings, and more. This challenge is huge in practice because the limited testing resources, in terms of people, hardware, and time, makes it impossible to exhaustively test every possible case. Therefore, testing engineers strive to design *representative* test cases to expose as many bugs as possible before software release. During this process, testing coverage criteria are greatly needed to help design and evaluate test case selection.

A *testing coverage criterion* (also called a *testing adequacy criterion*) is a model consisting of a set of testing requirements (also called program properties). It is designed based on certain testing domains and a class of target software bugs [GG75]. It can work as a metric [ZHM97] to measure the coverage and adequacy of a testing or a test case set. It can also work as a test case selector [BA82] to choose representative test cases or to filter out redundant test cases to expose software bugs in the most efficient way. A testing can claim to have achieved *full coverage* under a certain coverage criterion, iff the testing has satisfied all testing requirements (i.e., exercised all program properties) defined by the criterion. A coverage criterion is usually evaluated based on its testing complexity and bug-exposing capability. In practice, testing engineers select the most appropriate criterion based on their testing requirement and the amount of the testing resource.

2.3.2 Interleaving Coverage Criteria

Exposing concurrency bugs requires not only a bug-triggering input but also a bug-triggering interleaving. Unfortunately, exercising all interleavings associated with every input is infeasible in practice because the number of possible interleavings for each input is exponential to the program's execution size. Under this circumstance, a good interleaving coverage criterion is desired to help select representative interleavings to explore.

Previous research has proposed many different interleaving coverage criteria. Unfortunately, most of them require testing to cover an exponential number of interleavings [KT96, TLK92] and are therefore impractical. The proposal of concurrent data flow coverage [HM92, YP03] requires the testing to cover all feasible data flow combinations. It decreases the testing coverage requirements to polynomials of the program's execution size, but is still too large for practical testing. No previously proposed coverage criteria have been used in practice to test real-world big concurrent programs.

Previous research also studied how to conduct input generation to satisfy traditional coverage criteria, such as statement coverage, in concurrent programs [SA06]. Future research can combine these input generation techniques with interleaving testing.

2.3.3 Exposing Concurrency Bugs

The common practice to expose concurrency bugs is to run a program with each input test case for a long time (for servers) or for many times (for other types of applications). We refer to this as *stress testing*. Intuitively, it makes some sense because the non-deterministic nature of concurrent programs will help exercise different interleavings in different runs. Unfortunately, practice has shown that stress testing is neither efficient nor reproducible [MQ07] (more discussion about why stress testing is inefficient is offered in Section 7.3).

Recently, several inspiring approaches [BFM+05, EFN+02, MQ07, Sen08] were proposed to improve stress testing. All these approaches are targeted to reduce the exponential size of interleaving space into smaller sets of interleavings for practical testing to target.

ConTest [BFM+05] injects artificial delays at synchronization points (e.g., lock acquisition, lock release) in order to intensify the contention for synchronization resources. This would help expose deadlocks, but not data races or atomicity violation bugs, because the latter types are usually caused by programmers forgetting to use appropriate synchronization to protect shared memory accesses.

CHESS [MQ07] cleverly limits the number of context switches during an execution to 1–4 and therefore significantly reduces the number of interleavings to explore. However, it has to make a difficult trade-off between coverage and testing time. Even with a couple of context switches, there

are still a huge number of possible choices, including both which locations/threads to switch from and which locations/threads to switch to. The number of choices further increases polynomially if three or four switches are allowed. That is why CHESS allowed context switches only at synchronization points when it was used by Microsoft developers on real-world programs in order to be practical. Such a constraint will make the method less effective for exposing atomicity violation bugs and data race bugs, just like that in ConTest as discussed above.

Based on the same motivation, RaceFuzzer [Sen08, PS08] focuses on potential data races reported by race detectors. It attempts to force all the reported race interleavings during testing in order to separate false positives from true race bugs. While this approach is definitely useful to help users automatically filter out false positives in race bug detection, its bug-exposing capability significantly relies on the underlying data race detectors. If the detector does not have good coverage, RaceFuzzer would miss many bugs. Unfortunately, due to the inherent complexity of concurrent programs, there are still few race bug detectors that can achieve high coverage, especially for C/C++ programs and for atomicity violation bugs.

In addition, both CHESS and RaceFuzzer only select one thread to execute at a time, which can significantly slow down each test run and cannot fully take advantage of multi-core machines in testing. While it is possible to conduct multiple testing runs on the same machine, the contention for disk resource and network I/Os makes it impractical for I/O-intensive applications, such as server programs.

2.4 Other Related Works

Good concurrent programming languages can help programmers correctly express their intentions and therefore avoid certain types of concurrency bugs. Along this direction, many new programming language features have been proposed to support concurrent program development [VTD06, MZGB06, AGEB08]. Especially, there has been a lot of studies on transactional memory (TM) [ATKS07, AAK+05, HWC+04, HF03, Moi97, MBM+06, CTTC06, HM93] recently. TM provides programmers an easy way to specify which code regions should be atomic. The atomicity of the specified regions against other specified regions are protected through underlying TM hardware and software support.

How to survive concurrency bugs at production run is also an important topic towards improving the dependability of concurrent programs. Restart or checkpoint-retry [QTSZ05] techniques are quite effective for surviving concurrency bugs, as long as the bugs are detected in time. In recent Atom-Aid work [LDSC08], transactional memory is leveraged to help survive atomicity violation bugs that have escaped the in-house testing. By transparently adding transactions at run-

time, Atom-Aid can decrease the manifestation probability of an atomicity violation bug dramatically. These production-run surviving techniques can well complement the detection and testing techniques that are discussed earlier.

Deterministic replay techniques can significantly help the diagnosis of concurrency bugs. Multi-core deterministic replay requires recording a huge amount of memory access information at run time and faces the challenge of huge recording overhead. Previous works have proposed hardware and virtual machine support to speedup deterministic replay [NPC06, NPC05, XBH03, XHB06, HH08, MCT08, DLFC08]. Low overhead software implementation of multi-core deterministic replay is desired.

Chapter 3

Understanding the Characteristics of Real-World Concurrency Bugs

A good understanding of concurrency bugs is the foundation for addressing the concurrency bug problem. Unfortunately, as discussed in earlier chapters, there have been very few studies focusing on concurrency bugs. To address this problem, this chapter conducts a characteristics study of real-world concurrency bugs. This study will lay the groundwork for the research presented in later chapters. It will also provide guidance for future research regarding concurrent programs.

3.1 Overview

3.1.1 Motivation

Addressing the concurrency bug problem requires efforts from many directions, such as bug detection, software testing, and programming language design, among others. All of these directions have made progress over the past years. However, each of them is still facing many open questions:

(1) Concurrency bug detection Most previous concurrency bug detection research has focused on data race bugs [C+02, EA03, NM91, PT03, SBN+97, YRC05] and deadlock bugs [BLR02, EA03, SBN+97]. In order to improve concurrency bug detection, we want to know what types of concurrency bugs exist in the real world. It is especially important to understand whether there is any type that has not been addressed and whether the assumptions used by existing tools are valid. For example, most previous race detection tools focus on synchronization of access to each single variable. How many concurrency bugs are missed by this single variable assumption?

(2) Concurrent program testing and model checking As discussed in Section 2.3, concurrent program testing is greatly challenged by the huge interleaving space. It remains an open question whether (and how) we can selectively test a small number of representative interleavings and still expose most of the concurrency bugs. Finding answers to this question demands our knowledge about the *manifestation conditions* of real-world concurrency bugs. We need to know what conditions, in addition to what program inputs, are needed to reliably trigger a concurrency bug. Specifically, how many threads, how many variables, and how many accesses are usually involved

in a real-world concurrency bug's manifestation? This information will also help address the state explosion problem in concurrent program model checking.

(3) Concurrent programming language design Good concurrent programming languages can help programmers correctly express their intentions and therefore avoid certain types of concurrency bugs, as we described in Section 2.4. Along this direction, transactional memory (TM) [ATKS07, AAK+05, HWC+04, HF03, Moi97, MBM+06, MH06, RHP+07, CTTC06] is one of the popular trends. Although TM shows great potential, there are still many questions, such as (i) what portion of bugs can be avoided by using TM, (ii) what are the real-world concerns that TM design needs to pay attention to, and (iii) besides TM, what other programming language supports will be useful for programmers to write correct concurrent programs?

A better understanding of real-world concurrency bugs can help us answer these open questions —basically, we can learn from the common mistakes made by programmers in concurrent programs. For example, if many real-world concurrency bugs involve multiple shared variables, it will be necessary to study how to detect multi-variable concurrency bugs; if the manifestation of most real-world concurrency bugs are determined by the interaction between two threads, we can simply conduct pairwise testing for every pair of program threads; if many real-world concurrency bugs are hard to avoid using existing synchronization primitives, we should look for new language features to ease the process of writing concurrent program; if a certain type of information is frequently used by programmers to fix real-world concurrency bugs, bug detection tools can be extended to provide such information and thus become more useful in practice.

As reviewed in Section 2.1, most empirical studies in the past [CYC+01, SC92, Z. 06] have looked at general software bug characteristics, instead of focusing on concurrency bugs. Although these studies have provided many useful guidelines for improving software quality, they cannot answer these questions specific to concurrency bugs. Existing concurrency bug characteristic studies either have only a small number of concurrency bugs [CC00] or are based on bugs intentionally introduced for the study [FNU03]. A comprehensive study of the characteristics of real-world concurrency bugs is therefore desired.

3.1.2 Highlights

This chapter presents the first (to the best of our knowledge) comprehensive real-world concurrency bug characteristics study. Specifically, this study examines *bug patterns, manifestations, fix strategies* and other characteristics of real-world concurrency bugs. It is based on 105 randomly selected real-world concurrency bugs, including 74 non-deadlock bugs and 31 deadlock bugs, collected from four large and mature open-source applications: MySQL, Apache, Mozilla

Findings on Bug Patterns (Section 3.3)	**Implications**
(1)Almost all (97%) of the examined non-deadlock bugs belong to one of the *two simple bug patterns*: atomicity-violation or order-violation*.	Concurrency bug detection can focus on these two bug patterns to detect most concurrency bugs.
(2) About one third (32%) of the examined non-deadlock bugs are *order-violation bugs*, which are *not* well addressed in previous work.	*New* concurrency bug detection tools are needed to detect order-violation bugs, which are not addressed by existing atomicity violation or race detectors.
Findings on Manifestation (Section 3.4)	**Implications**
(3) Almost all (96%) of the examined concurrency bugs are guaranteed to manifest if certain partial order between *2 threads* is enforced.	Pairwise testing on concurrent program threads can expose most concurrency bugs, and greatly reduce the testing complexity.
(4) Some (22%) of the examined deadlock bugs are caused by *one thread* acquiring resource held by itself.	Single-thread based deadlock detection and testing techniques can help eliminate these simple deadlocks.
(5) Many (66%) of the examined non-deadlock concurrency bugs' manifestation involves concurrent accesses to *only one variable.*	Focusing on concurrent accesses to one variable is a good simplification for concurrency bug detection, which is used by many existing bug detectors.
(6) One third (34%) of the examined non-deadlock concurrency bugs' manifestation involves concurrent accesses to *multiple variables.*	*New* detection tools are needed to address *multiple variable concurrency bugs.*
(7) Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most *two resources*.	Pairwise testing on the acquisition/release sequences to two resources can expose most deadlock concurrency bugs, and reduce testing complexity.
(8) Almost all (92%) of the examined concurrency bugs are guaranteed to manifest if certain partial order among *no more than 4 memory accesses* is enforced.	Testing partial orders *among every small group* of accesses can expose most concurrency bugs, and simplify the interleaving space *from exponential to*
Findings on Bug Fix Strategies (Section 3.5)	**Implications**
(9) Three quarters (73%) of the examined non-deadlock bugs are fixed by techniques *other than* adding/changing locks.	Bug detection and diagnosis tools need to provide more bug pattern and manifestation information, besides lock information, to help programmers fix bugs.
(10) Many (61%) of the examined deadlock bugs are fixed by stopping one thread from using a resource (e.g., lock). Such fix can introduce non-deadlock	Fixing deadlock bugs might introduce non-deadlock concurrency bugs. Special help is needed to ensure the correctness of deadlock bug fixes.
Findings on Bug Avoidance (Section 3.5.4)	**Implications**
(11) Transactional memory (TM) can help avoid about one third (39%) of the examined concurrency bugs.	Transactional memory (TM) is a promising language feature for programmers.
(12) TM *could* help avoid over one third (42%) of the examined concurrency bugs, if some *concerns* are addressed.	TM design need to pay attention to some concerns, such as how to protect hard-to-rollback operations.
(13) Some (19%) of the examined concurrency bugs *cannot* benefit from basic TM, because of their bug patterns.	Better programming language features to help express *"order"* semantics in C/C++ programs are desired.

Table 3.1: Findings of real-world concurrency bug characteristics and their implications on bug detection, program testing and programming language design. (*: All terms and categories mentioned here will be explained in Section 3.2.)

and OpenOffice, representing both server and client applications. To understand each bug, its bug report, related source code, patches, and programmers' discussion are all carefully examined.

This study reveals many interesting findings and provides useful guidelines for concurrency

bug detection, concurrent program testing, and concurrent programming language design. The main findings and their implications are summarized in Table 3.1.

Although the examined applications and bugs were carefully selected to represent a large body of concurrent applications, this study does not intend to draw general conclusions about all concurrent applications. The results presented in this chapter should be taken with the specific applications and the evaluation methodology in mind (Section 3.2.3 discusses the threats to validity).

3.2 Methodology

3.2.1 Bug Sources

Applications: Four representative open source applications are selected for this study: MySQL, Apache, Mozilla, and OpenOffice. These are all mature (with 9–13 years development history) large concurrent applications (with 1–4 million lines of code), with well maintained bug databases. These four applications represent different types of server applications (database and web server) and client/desktop applications (browser suite and office suite). Concurrency is used for different purposes in these applications. Server applications mostly use concurrency to handle concurrent client requests. They can have hundreds or thousands of threads running at the same time. Client and desktop applications mostly use concurrency to synchronize multiple GUI sessions and background working threads.

Bugs: Concurrency bugs are *randomly* collected from the bug databases of the above applications. Since these databases contain more than five hundred thousand bug reports, in order to effectively collect concurrency bugs from them, a large set of keywords that are related to concurrency bugs are used, for example, 'race(s)', 'deadlock(s)', 'synchronization(s)', 'concurrency' 'lock(s)', 'mutex(es)', 'atomic', 'compete(s)', and their variations. From the thousands of bug-reports that contain at least one keyword from the above keyword set, about five hundred bug reports with clear and detailed root cause descriptions, source codes, and bug fix information, are *randomly* picked. Then, these randomly picked bugs are manually check to make sure that they are really concurrency bugs. After this step, 105 concurrency bugs are included into this study.

These 105 concurrency bugs contain two types: *deadlock bugs* and *non-deadlock concurrency bugs*. Since these two types of bugs have completely different properties and demand different detection, recovery approaches, they are separated in this study for the ease of investigation.

In summary, 105 concurrency bugs are collected and used in this study. These 105 bugs include 74 non-deadlock concurrency bugs and 31 deadlocks bugs. The details are shown in Table 3.2.

Application	Description	# of Bug Samples	
		Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Browser Suite	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Table 3.2: Applications and bugs examined in the characteristics study

3.2.2 Characteristic Categories

In order to provide guidance for future research on concurrent program reliability, this work focuses on three aspects of concurrency bug characteristics: bug pattern, manifestation, and bug fix

Definitions Related to Bug Pattern Study			
Dimension	**Category**	**Description**	**Abbr.**
Bug Pattern*	Atomicity Violation	The desired serializability among multiple memory accesses is violated. (i.e., a code region is intended to be atomic, but the atomicity is not enforced during execution.)	Atomicity
	Order Violation	The desired order between two (groups of) memory accesses is flipped. (i.e., A should always be executed before B, but the order is not enforced during execution.)	Order
	Other	Concurrency bugs other than the above two types.	Other
Definitions Related to Bug Manifestation Study			
Dimension	**Term**	**Definition**	
Bug Manifestation	Manifestation Condition	A specific execution order among a smallest set (S) of memory accesses. Enforcing this order, no matter how, guarantees the bug to manifest.	
	# threads involved	The number of distinct threads that are included in S.	
	# variable involved	The number of distinct variables that are included in S.	
	# accesses involved	The number of accesses that are included in S.	
Definitions Related to Bug Fix Study			
Dimension	**Category**	**Description**	**Abbr.**
Non-deadlock Fix Strategy	Condition Check	(1) While-flag; or (2) optimistic concurrency with consistency check.	COND
	Code Switch	Switch the order of certain statements in the source code.	Switch
	Design Change	Change the design of data structures or algorithms.	Design
	Lock Strategy	(1) Add/change locks; or (2) adjust the boundary of critical sections.	Lock
	Other	Strategies other than the above ones.	Other
Deadlock Fix Strategy	Give up resource	Not acquiring a resource (lock, etc.) for certain code region.	GiveUp
	Split Resource	Split a big resource to smaller pieces to avoid competition.	Split
	Change acquisition order	Switch the acquisition order among several resources.	AcqOrder
	Other	Strategies other than the above ones.	Other
Concerns in TM**	Very long code	A code region is too long to be put into a transaction.	Long
	Rollback Problem	Some I/O and system calls are hard to roll back.	Rollback
	Code Nature	Source code with certain design is hard to turn to transaction.	Nature

Table 3.3: Characteristic categories and definitions. (*: The bug pattern category is determined by the root cause, i.e., what type of synchronization intention is violated, *regardless of the possible fix strategies.* **: TM is short for transactional memory.)

strategy. Other characteristics, such as failure impact and bug diagnosis process, will be briefly discussed at the end.

(1) Along the *bug pattern* dimension, non-deadlock concurrency bugs are classified into three categories (atomicity-violation bugs, order-violation bugs and the other bugs) based on their root causes, i.e., what types of synchronization intentions (or assumptions) are violated. Detailed definitions are shown in Table 3.3. Here data race is not classified as a bug pattern. The reason is that the data race concept does not necessarily mean a bug and is orthogonal to the above three categories: some data races are accepted by developers while some data races may imply any of the above three categories of bugs (more discussion is in Section 4.1). Deadlocks are not further categorized as most of them are similar and simple.

(2) The *manifestation* dimension studies the required condition for each concurrency bug to manifest (denoted as *manifestation condition*, defined in Table 3.3), discussing how many threads, how many variables/resources, and how many accesses are involved in concurrency bugs' manifestation conditions.

(3) For the *bug fix strategy*, both the fixing strategies of the final patches and the mistakes of the intermediate patches are studied. How transactional memory can help avoid these bugs is also studied. All the related classification is shown in Table 3.3.

3.2.3 Threats to Validity

Similar to the previous work, real-world characteristic studies are all subject to a validity problem. Potential threats to the validity of our characteristics study are the representativeness of the applications, concurrency bugs used in our study, and our examination methodology.

As for application representativeness, this study chooses four server and client-based concurrent applications written in C/C++, which are the popular programming languages for these types of applications. I believe that these four applications well represent server and client-based concurrent applications, which are two large classes of concurrent applications. However, this study may not reflect the characteristics of other types of applications, such as scientific applications, operating systems, or applications written in other programming languages (e.g., Java).

As for bug representativeness, the studied concurrency bugs are *randomly* selected from the bug database of the above applications. They provide good samples of the fixed bugs in those applications. While characteristics of non-fixed or non-reported concurrency bugs might be different, these bugs are not likely as important as the reported and fixed bugs that are examined in this study.

In terms of the examination methodology, every piece of information related to each examined bug is carefully examined. It includes programmers' clear explanations, forum discussions, source code patches, multiple versions of source codes, and bug-triggering test cases.

Overall, while the conclusions in this chapter cannot be applied to *all* concurrent programs, I believe that this study does capture the characteristics of concurrency bugs in two large important classes of concurrent applications: server-based and client-based applications. In addition, most of these characteristics are consistent across all four examined applications, indicating the validity of the evaluation methodology to some degree. Additionally, this dissertation does not emphasize any quantitative characteristic results of this study. Finally, I warn the readers to take the findings together with above methodology and selected applications.

3.3 Bug Pattern Study

Different bug patterns usually demand different detection and diagnosis approaches. In Table 3.4, the examined non-deadlock concurrency bugs are classified into three categories based on their patterns: Atomicity, Order, and Other (definitions are described in Table 3.3). Note that the categories are distinguished from each other by the bug root cause of a bug, regardless of the possible bug fix strategies.

Application	Total	Atomicity	Order	Other
MySQL	14	12	1	1
Apache	13	7	6	0
Mozilla	41	29	15	0
OpenOffice	6	3	2	1
Overall	74	51	24	2

Table 3.4: Patterns of non-deadlock concurrency bugs. (There are three examined bugs, whose patterns can be considered as either atomicity or order violation. Therefore, they are put in both categories.)

Finding (1): Most (72 out of 74) of the examined non-deadlock concurrency bugs are covered by two simple patterns: *atomicity-violation* and *order-violation*.

Implications: Concurrent program bug detection, testing and language design should first focus on these two major bug patterns.

The Finding (1) can be explained by the fact that atomicity and pairwise order are the two most common synchronization intentions of programmers. However, it is not easy to enforce all these intentions correctly in implementation. As a result, atomicity-violation bugs and order violation bugs dominate the examined non-deadlock concurrency bugs.

It is very common for programmers to assume that a small code region is executed atomically because programmers think sequentially. For example, in Figure 3.1, programmers assume that if S1 reads a non-NULL value from `thd->proc_info`, S2 will also read the same value. However, such an *atomicity assumption* can be violated by S3 during concurrent execution, and it leads to a program crash.

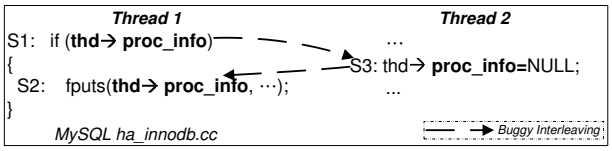


Figure 3.1: An atomicity violation bug from MySQL.

It is also common for programmers to assume an order between two operations from different threads. However, programmers may forget to enforce such an order. As a result, one of the two operations may be executed faster (or slower) than the programmers' assumption, and triggers the order bug. In the Mozilla bug shown in Figure 3.2, it is easy for programmers to assume wrongly that thread 2 would dereference `mThread` *after* thread 1 initializes it, because thread 2 is created by thread 1. However, in real execution, thread 2 may be very quick and dereference `mThread` *before* `mThread` is initialized. This unexpected order leads to program crash. Note that even though the bug can be fixed with locks, the root cause of the bug is a violation to programmers' order assumption, not atomicity assumption.

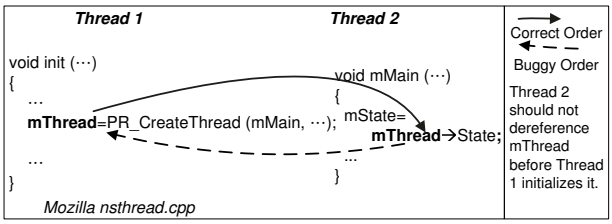


Figure 3.2: An order violation bug from Mozilla.

Concurrency bugs violating other types of synchronization intentions also exist, but are much rarer as shown in Table 3.4. Figure 3.3 shows an example. In one version of MySQL, programmers use a timeout threshold `fatal_timeout` to detect deadlock. The server will crash if any thread waits for a lock for more than `fatal_timeout` amount of time. However, when programmers set the threshold, they under-estimate the workload. As a result, users found that the MySQL server keeps crashing under heavy workload (with 2048 worker-thread setting). Such a performance-related assumption is neither related to atomicity nor to order [1].

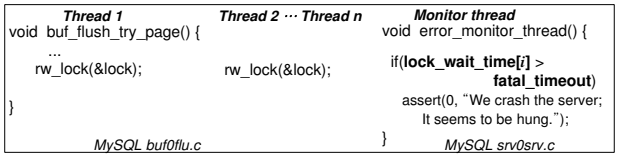


Figure 3.3: A MySQL bug that is neither an atomicity-violation bug nor an order-violation bug (simplified for illustration).

[1] This bug is fixed by limiting the number of worker-threads

Finding (2): A significant number (24 out of 74) of the examined non-deadlock concurrency bugs are order bugs, which are *not* addressed by previous bug detection work.

Implications: New bug detection techniques are desired to address order bugs.

As we discussed above, it is common for programmers to assume a certain order between two operations from two threads. Specifically, programmers can have an order intention i) between a write and a read (Figure 3.2) to one variable; ii) between two writes (Figure 3.4) to one variable; or iii) between two groups of accesses to a group of variables (Figure 3.5). In Figure 3.4, programmers expect S2 to initialize `io_pending` before S4 assigns a new value, `FALSE`, to it. However, the execution of the asynchronous read can be very quick and S4 may be executed before S2, contrary to the expectation of programmers. This causes thread 1 to hang. In another example shown in Figure 3.5, `js_UnpinPinnedAtom` frees all elements in the `atoms` array. This set of memory accesses to the whole array is expected to happen after `js_MarkAtom`, which may access some elements in `atoms`. However, this intention is not assured by the implementation, which becomes a bug.

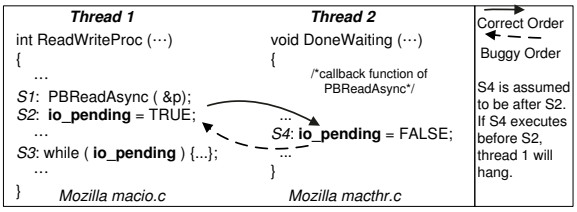


Figure 3.4: A write-write order violation bug from Mozilla.

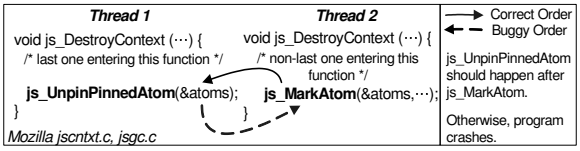


Figure 3.5: A Mozilla bug that violates the intended order between two groups of operations.

Note that the above order bugs are *different* from data race bugs and atomicity violation bugs. Even if two memory accesses to the same variable are protected by the same lock or two conflicting code regions are atomic to each other, the execution order between them still may not be guaranteed. We should also note that some order-violation bugs could be *fixed* using coarser-grained locking, as in example Figure 3.2 and Figure 3.4; some others cannot be fixed by locks, as in example Figure 3.5 and Figure 3.7 (will be discussed later). This is not related to the bug root cause, and does not affect the bug pattern classification.

Although important and common, order-violation bugs have not been well studied by previous research. Many order bugs will be missed by existing concurrency bug detectors, which mainly focus on race bugs and atomicity bugs. New techniques are desired for solving the order problems.

3.4 Bug Manifestation Study

Manifestation condition of a concurrency bug is usually a specific order among a set of memory accesses or system events. This section studies the characteristics of real-world concurrency bug manifestation, following the methodology defined in Table 3.3. Guidance for concurrent program testing and concurrency bug detection will also be discussed based on the observations.

3.4.1 How Many Threads are Involved?

Non-deadlock concurrency bugs					
Application	Total	Env.	>2 threads	2 threads	1 thread
MySQL	14	1	1	12	0
Apache	13	0	0	13	0
Mozilla	41	1	0	40	0
OpenOffice	6	0	0	6	0
Overall	74	2	1	71	0
Deadlock concurrency bugs					
Application	Total	Env.	>2 threads	2 threads	1 thread
MySQL	9	0	0	5	4
Apache	4	0	0	4	0
Mozilla	16	0	1	14	1
OpenOffice	2	0	0	0	2
Overall	31	0	1	23	7

Table 3.5: The number of threads/environments involved in concurrency bugs.

Finding (3): The manifestation of most (101 out of 105) examined concurrency bugs involves no more than two threads.

Implications: Concurrent program testing can pairwise test program threads, which reduces testing complexity without losing bug exposing capability much.

Finding (3) tells us that even though the examined server programs use hundreds of threads, in most cases, only a small number (mostly just *two*) of threads are involved in the manifestation of a concurrency bug.

The underlying reason for above observation is that most threads do *not* closely interact with many others: most communication and collaboration are conducted between two or a small group of threads. As a result, manifestation conditions of most concurrency bugs do not involve many threads. For examples, all of the bugs presented in Section 3.3, except the one shown in Figure 3.3,

are guaranteed to manifest if their execution follow certain partial orders (marked by dotted lines in the figures) between two threads.

We should note that this finding is *not* opposite to the common observation that concurrency bugs are sometimes easier to manifest at a heavy-workload (concurrent execution of many threads). In many cases, the manifestation condition involves only two threads. Heavy-workload increases the resource competition and context switch intensity. It therefore increases the possibility of hitting certain orders among the *two threads* that can trigger the bug. The manifestation condition itself still involves just two threads.

This finding implies that testing can focus on exercising different orders among accesses from every pair of threads. Such pairwise testing technique can prevent the testing complexity from increasing exponentially with the number of threads. At the meantime, few concurrency bugs would be missed.

There are also cases where the bug manifestation relies on not only memory accesses within the program, but also environmental events (as shown in column 'Env' in Table 3.5). For example, one Mozilla bug cannot be triggered unless another program modifies the same file concurrently with Mozilla. Exposing such bugs needs special system support.

Finding (4): The manifestation of some (7 out of 31) deadlock concurrency bugs involves only one thread.
Implications: This type of bug is relatively easy to detect and avoid. Bug detection and programming language research can try to eliminate these simple bugs first.

A one-thread deadlock bug usually happens when one thread tries to acquire a resource held by itself. Detecting this type of bugs is relatively easy, because intra-thread, instead of inter-thread, analysis would be sufficient to reveal many of them.

3.4.2 How Many Variables are Involved?

Are concurrency bugs synchronization problems among accesses to one variable or multiple variables? To answer this question, this section examines the number of variables (or resources) involved in the manifestation of each concurrency bug. The examination result is shown in Table 3.6.

Non-deadlock concurrency bugs			
Application	Total	>1 variables	1 variable
MySQL	14	6	8
Apache	13	4	9
Mozilla	41	15	26
OpenOffice	6	0	6
Overall	74	25	49

Deadlock concurrency bugs				
Application	Total	>2 resources	2 resources	1 resource
MySQL	9	0	5	4
Apache	4	0	4	0
Mozilla	16	1	14	1
OpenOffice	2	0	0	2
Overall	31	1	23	7

Table 3.6: The number of variables/resources involved in concurrency bugs.

Finding (5): 66% (49 out of 74) of the examined non-deadlock concurrency bugs involve only one variable.

Implications: Focusing on concurrent accesses to one variable is a good simplification for concurrency bug detection.

Finding (5) confirms the common intuition: more non-deadlock concurrency bugs are caused by contention among accesses to the same variable rather than different variables. The reason is that flipping the order of two accesses to different memory locations does not *directly* change the program state and therefore is less likely to cause problems. Figure 3.1, 3.2, and 3.4 are all examples of single variable concurrency bugs: their manifestation can be guaranteed by certain order among accesses to one variable.

This finding supports the single-variable assumption taken by many existing bug detectors. For example, data race bug detection [SBN+97, YRC05] checks the synchronization among accesses to one variable, instead of multiple variables.

Finding (6): A non-negligible number (34%) of non-deadlock concurrency bugs involve more than one variable.

Implications: We need *new* concurrency bug detection tools to address multiple variable concurrency bugs.

Multiple variable concurrency bugs usually occur when unsynchronized accesses to correlated variables cause inconsistent program state. Semantic connections among variables are common, and therefore multiple variable concurrency bugs are not rare.

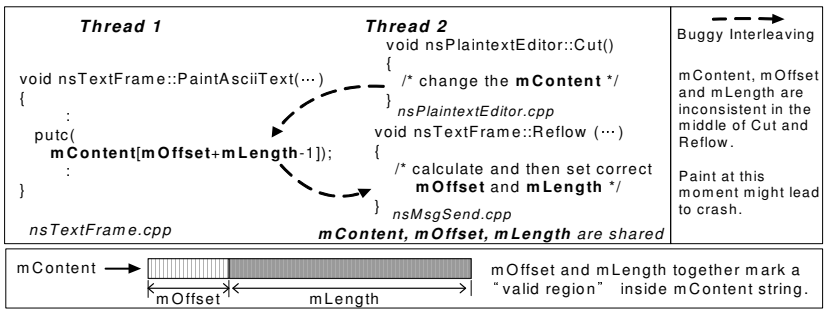


Figure 3.6: A multi-variable concurrency bug from Mozilla. (Accesses to three correlated variables, `mContent`, `mOffset` and `mLength`, should be synchronized.)

Figure 3.6 shows an example of multiple variable concurrency bug from Mozilla. In this example, `mOffset` and `mLength` together mark the region of useful characters stored in dynamic string `mContent`. Thread 1 and 2's concurrent accesses to these *three* variables should be synchronized, otherwise thread 1 might read inconsistent values and access invalid memory address. Here, controlling the order of memory accesses to any single variable *cannot* guarantee the bug to manifest. For example, it is all right for thread 1 to read `mContent` either before or after thread 2's modification to *all* of these three variables. The required condition for the bug to manifest is that thread 1 uses the *three* correlated variables in the middle of thread 2's modification to these *three* variables.

As discussed above, most existing bug detection tools only focus on single-variable concurrency bugs. Although this simplification provides a good starting point for concurrency bug detection, future research should not ignore the problem of multi-variable concurrency bugs.

The difficulty of detecting multiple variable concurrency bugs is that it is hard to infer which accesses, to different variables, should be well synchronized. Solving this problem will not only benefit automatic concurrency bug detection, but also provide useful hints for programmers to specify correct transactions or atomic regions for transactional memory or atomicity bug detection tools [FF04].

Finding (7): 97% (30 out of 31) of the examined deadlock concurrency bugs involve at most two resources.

Implications: Deadlock-oriented concurrent program testing can pairwise test the order among acquisition and release of two resources.

Among the examined deadlock bugs, only one bug is triggered by three threads circularly waiting for three resources. Leveraging this finding, pairwise testing on resources can prevent the testing complexity from increasing exponentially with the total number of resources.

3.4.3 How Many Accesses are Involved?

The above study has shown that the manifestation of most concurrency bugs involves only two threads and a small number of variables. However, the number of accesses from one thread to each variable can still be huge. Therefore, it is necessary to investigate how many accesses are involved in the bug manifestation.

Non-deadlock concurrency bugs						
Application	Total	1 acc.*	2 acc.	3 acc.	4 acc.	>4 acc.
MySQL	14	0	2	7	4	1
Apache	13	0	6	5	2	0
Mozilla	41	0	12	18	5	6
OpenOffice	6	0	2	3	1	0
Overall	74	0	22	33	12	7
Deadlock concurrency bugs						
Application	Total	1 acc.*	2 acc.	3 acc.	4 acc.	>4 acc.
MySQL	9	4	1	4	0	0
Apache	4	0	0	4	0	0
Mozilla	16	1	2	12	0	1
OpenOffice	2	2	0	0	0	0
Overall	31	7	3	20	0	1

Table 3.7: The number of accesses (or resource acquisition/release operations) involved in concurrency bugs. (*: "1 acc." only happens in deadlock bugs, when one thread waits for itself. The bug triggering does not depend on any inter-thread order.)

Finding (8.1): 90% (67 out of 74) of the examined non-deadlock bugs can deterministically manifest, if certain orders among at most four memory accesses are enforced.

Finding (8.2): 97% (30 out of 31) of the examined deadlock bugs can deterministically manifest, if certain orders among at most four resource acquisition/release operations are enforced.

Implications: Concurrent program testing can focus on exercising different orders within every small groups of accesses, instead of all memory accesses. This scheme can reduce the interleaving testing's target space from exponential to polynomial, with little loss of bug exposing capability.

The Finding (8.1) can be easily understood, because most of the examined concurrency bugs have simple patterns and involve a small number of variables. Most of the exceptions come from those bugs that involve more than two threads and/or more than two variables. The Finding (8.2) is also natural, considering that most of the examined deadlock bugs involve only two resources.

The above findings have significant implication for concurrent program testing. The challenge in concurrent program testing is that the number of all possible interleavings is exponential to the number of dynamic memory accesses, which is too big to thoroughly explore. Above finding implies a more effective design of interleaving testing: exploring all possible orders within every small groups of memory accesses, e.g., groups of 4 memory accesses, instead of all memory accesses. The complexity of this design is only polynomial to the number of dynamic memory accesses, which is a huge reduction from the exponential-sized all-interleaving testing scheme. Furthermore, the bug exposing capability of this design is almost as good as exploring all interleavings, missing only few bugs in this study. Chapter 6 will discuss in detail about how to leverage these characteristics and design a new hierarchy of interleaving coverage criteria.

A recent model checking work [MQ07] uses a heuristic to start the checking from interleavings with small numbers of context switches. This heuristic is supported by the study in this section.

Of course, enforcing a specific order among a set of accesses is not trivial. The program input and many accesses need to be carefully controlled. How to leverage above finding to enable practical and powerful concurrent program testing and model checking remains as future work.

3.5 Bug Fix Study

3.5.1 Fix Strategies for Non-deadlock Bugs

Before checking how the real-world bugs were fixed, my guess was that adding or changing locks should be the most common way to fix concurrency bugs. However, the characteristic result is contrary to my guess, as shown in Table 3.8.

Application	Total	COND	Switch	Design	Lock	Other
MySQL	14	2	0	5	4	3
Apache	13	4	2	3	4	0
Mozilla	41	13	8	9	9	2
OpenOffice	6	0	0	2	3	1
Overall	74	19	10	19	20	6

Table 3.8: Fix strategies for non-deadlock concurrency bugs (all categories are explained in Table 3.3).

Finding (9): Adding or changing locks is *not* the major fix strategy. It is used for only 20 out of 74 non-deadlock concurrency bugs that we examined.

Implication: There is no silver bullet for fixing concurrency bugs. Just telling programmers that certain conflicting accesses are not protected by the same lock is not enough to fix concurrency bugs.

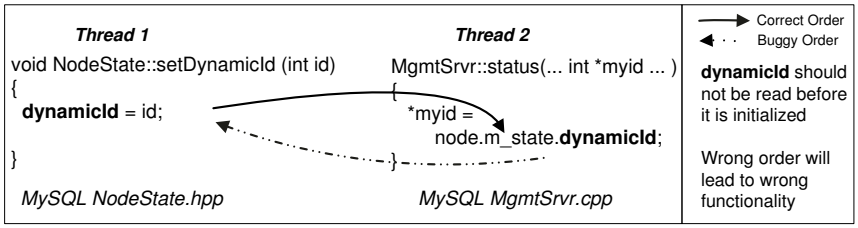


Figure 3.7: A MySQL bug that cannot be fixed by adding/changing locks.

There are two reasons for this controversy. First of all, locks cannot guarantee to enforce some synchronization intentions, such as A should happen before B. Therefore, adding/changing locks cannot fix certain types of bugs. Figure 3.5 showed such an example. Figure 3.7 is another simple example. Secondly, even if adding/changing locks can fix a bug, in many cases, it is not the best strategy, because it may hurt the performance or introduce new bugs, such as deadlock bugs.

In the following, we look at the different strategies, other than adding/changing locks, used by programmers. We will see that these strategies usually require deep understanding of program semantics. In the meantime, they usually have better performance than corresponding lock-based fixes, if existing.

(1) Condition check (denoted as COND). Condition check can be used in different ways to help fix concurrency bugs. One way is to use while-flag to fix order-related bugs, such as the bug shown in Figure 3.5. The other way is to add consistency check to monitor the bug-related program states. This enables the program to detect buggy interleavings and restore program states. For example, to fix the bug shown in Figure 3.6, the program does consistency check `if(strlen(mContent)>= mOffset+mLength)` before it executes `putc` function. The `putc` will be skipped if the consistency check fails. In another example shown in Figure 3.8, condition `(n!=block->n)` is checked to see whether the shared variable `block->n` has been overwritten since the last time it was read. If `n` is not consistent with `block->n`, the program rolls back and reads `block->n` again. Note that, above fix strategy does *not* eliminate the buggy interleaving, which is usually the purpose of lock-based fixes. Instead, it focuses on detecting buggy interleavings and makes sure the program states corrupted by the buggy interleavings can be recovered in time. It has better performance than corresponding lock-based fixes, if the bad interleaving rarely occur.

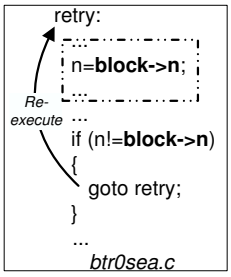


Figure 3.8: A MySQL bug fix.

(2) Code switch (denoted as Switch). Switching the order of certain code statements can fix some order-related bugs. For example, the order bug shown in Figure 3.4 is fixed by switching statements S1 and S2, so that S2 is always executed before S4.

(3) Algorithm/Data-structure design change (denoted as Design). This includes different types of algorithm and data structure changes that help to achieve correct synchronization. Some design changes are simple, just modifying a few data structures. For example, in the MySQL bug #7209, the bug is caused by unprotected conflicting accesses to a shared variable `HASH::current_record`. Programmers recognize that this variable does not need to be shared. They simply move the field `current_record` out of the class `HASH`, making it a local variable for each thread, and fix the bug. As another example, in Mozilla bug #201134, one thread needs to conduct a series of operations on a shared variable `nsCertType`. In order to enforce the atomicity of that series of operations, programmers simply let program read `nsCertType` into a local variable, conduct operations on the local variable, and store the value back to `nsCertType` at the end. Some design changes are more complicated, involving algorithm re-design. For example, in Mozilla bug #131447, programmers changed a message handling and queuing algorithm to tolerate special timing when a reply message arrives before its corresponding callback function is ready.

As we can see, fixing concurrency bugs is much more complicated than just adding or changing lock operations. Race detection tools can help programmers conduct those lock-related fixes, but this is not enough. It is desired to have more tools to help programmers figure out the bug pattern, the consistency condition associated with each bug, etc. For example, if programmers know that the bug is an order-violation bug and they also know what the consistency condition is, it is easy to come out with a *condition check* fix. This is the challenge for future research on concurrency bug detection and diagnosis.

3.5.2 Fix Strategies for Deadlock Bugs

Application	Total	GiveUp	Split	AcqOrder	Other
MySQL	9	5	0	2	2
Apache	4	2	0	2	0
Mozilla	16	11	1	3	1
OpenOffice	2	1	0	0	1
Overall	31	19	1	7	4

Table 3.9: Fix strategies for deadlock bugs (all categories are explained in Table 3.3)

Finding (10): The most common fix strategy (used in 19 out of 31 cases) for the examined deadlock bugs is to let one thread give up acquiring one resource, such as a lock. This strategy is simple, but it may introduce other non-deadlock bugs.

Implication: We need to pay attention to the correctness of some "fixed" deadlock bugs.

Table 3.9 summarizes the fixing strategies for deadlock bugs. As we can see, the most common strategy is 'GiveUp'. In many cases, programmers find it unnecessary or not worthwhile to acquire a lock within certain program context. Therefore, they simply drop the resource acquisition to avoid the deadlock.

However, this 'GiveUp' strategy could introduce non-deadlock concurrency bugs. In some of the examined bug reports, programmers explicitly say that they know the fix would introduce a new non-deadlock concurrency bug. They still adopt the fix, because they gamble that the probability for the non-deadlock bug to occur is small. In the future, techniques combining optimistic concurrency and rollback-reexecution, such as TM, can help fix some deadlock bugs. Of course, using these techniques should also be careful, because they might introduce live-lock problems.

3.5.3 Mistakes During Bug Fixing

Fixing bugs is hard. Some patches released by programmers are still buggy. In order to investigate the nature of buggy patches, all the distinct buggy patches of the 57 Mozilla concurrency bugs are checked [2]. Specifically, at first, all the intermediate (non-final) patches submitted by Mozilla programmers for these 57 bugs are gathered. Then these patches are manually checked to filter out non-bug-fixing patches, which only change comments or code structures for maintenance purpose.

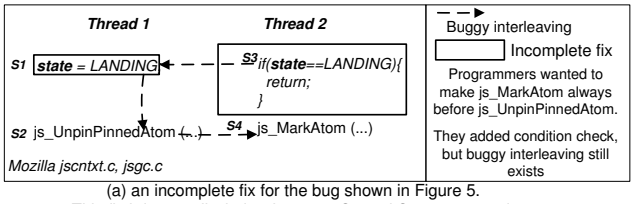
This study finds that 17 out of the 57 Mozilla bugs have at least one buggy patches. On average, 0.4 buggy patches were released before every final correct patch. Among all the 23 distinct buggy patches, 6 of them only decrease the occurrence probability of the original concurrency bug, but fail to fix the original bug completely (an example is shown in Figure 3.9). 5 of them introduce new concurrency bugs. The other 12 introduce new non-concurrency bugs. Programmers need help to improve the quality of their patches.

3.5.4 Discussion: Bug Avoidance

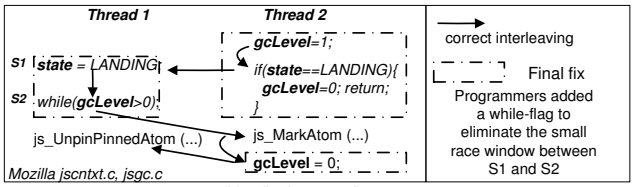
Good programming languages should help avoid some bugs during implementation. Transactional memory (TM) is a popular trend of programming language feature for easing concurrent programming. To estimate its benefit and understand what more are needed along this direction, the 105 concurrency bugs are studied. Specifically, the study checks how many of the 105 bugs can potentially be avoided with TM support and what are the issues that future concurrent programming language design needs to address.

Again, all the findings in this section should be interpreted with the evaluation methodology in mind, as discussed in Section 3.2.3. In addition, since different TM designs may have dif-

[2]This study focuses on Mozilla, because it has the best maintenance of patch update information.



(a) an incomplete fix for the bug shown in Figure 5.
This fix left a small window between S1 and S2 unprotected.



(b) a final correct fix.
Now the order between js_MarkAtom and js_UnpinPinnedAtom is enforced.

Figure 3.9: The process of fixing the bug shown in Figure 3.5. (Programmers want to make sure `js_MarkAtom` will not be called after `js_UnpinPinnedAtom`. They first added a flag variable `state` to fix the bug. However, that left a small window between S1 and S2 unprotected. They finally added a second flag variable `gcLevel` to completely fix the bug.).

ferent features, this discussion focuses on the basic atomicity and isolation properties of TM. Both the benefits and the concerns are discussed in general, based on some basic TM designs [AAK+05, HF03, Moi97]. It is definitely possible for advanced TM designs to address some of the concerns discussed here, which is exactly the purpose of this discussion: provide more real-world information and help improve the design of TM.

Application	Total	Can Help	TM might help(concerns:)			Little Help
			Long	Rollback	Nature	
MySQL	23	7	0	14	0	2
Apache	17	7	0	3	1	6
Mozilla	57	25	8	9	5	10
OpenOffice	8	2	0	4	0	2
Overall	105	41	8	30	6	20

Table 3.10: Can TM help avoid concurrency bugs?

Finding (11): TM can help avoid many concurrency bugs (41 out of the 105 concurrency bugs we examined).

Implication: Although TM is not a panacea, it can ease programmers correctly expressing their synchronization intentions in many cases, and help avoid a big portion of concurrency bugs.

Atomicity violation bugs and deadlock bugs with relatively small and simple critical code regions can benefit the most from TM, which can help programmers clearly specify this type of

atomicity intention. Figure 3.8 shows an example, where programmers use a consistency check with re-execution to fix the bug. Here, a *transaction* (with abort, rollback and replay) is exactly what programmers want.

> **Finding (12):** TM can potentially help avoid many concurrency bugs (44 out of the 105 concurrency bugs we examined), if some concerns can be addressed, as shown in Table 3.10.
>
> **Implication:** TM design can combine system supports and other techniques to solve some of these concerns, and further ease the concurrent programming.

One concern, not a surprise, is I/O operations. As operations like I/O are hard to roll back, it is hard to use TM to protect the atomicity of code regions which include such operations. Take the concurrency bug in Figure 3.1 as an example. Since S2 involves a file operation, TM might need non-trivial undo techniques to protect the S1–S2 atomic region.

Other concerns, such as atomic region size and special code nature, also exist. For example, the atomic code regions of several Mozilla bugs include the whole garbage collection process. These regions could have too large memory footprint to be effectively handled by hardware-TM.

Addressing many of the above concerns are feasible, but requiring higher overhead and complexity. For example, some of the rollback concerns can be addressed using system supports. Very long transactions can be addressed by combining software and hardware TMs.

> **Finding (13):** 20 out of the 105 concurrency bugs cannot benefit from the basic TM designs, because the violated programmer intentions, such as order intentions, cannot be guaranteed by the basic TM.
>
> **Implications:** Apart from atomicity intentions, there is also a significant need for concurrent programming language features to help programmers express order intentions easily.

Programmers' order intention is the major type of intention that cannot be easily enforced by the basic TM design or locks. In general, the basic TM designs cannot help enforce the intention that A has to be executed before B. Therefore, they cannot help avoid many related order-violation bugs [3]. Among all order-violation bugs, there is a sub-type of order intentions that is extremely hard to be enforced by basic TM designs: A must be either executed before B or not executed at all. In other words, programmers do not want B to wait for A. They simply skip A if B is already

[3] Some order-violation bugs can be avoided by TM. In those cases, order intentions can be enforced as side effects while TM enforces the atomicity of related code regions (an example is shown in Figure 3.2).

executed. For example, in one Mozilla bug, thread 1 keeps inserting entities to a cache and thread 2 would destroy the cache at some moment. Based on the description in the bug report, programmers do not want thread 2 to wait for thread 1 to finish all insertions. The program simply skips any insertion attempt after the cache is destroyed. This happens for 7 bugs.

In order to help avoid above 20 bugs, the semantic design, instead of implementation schemes, of the basic TM needs to be enhanced. Recently, some TM designs [CMC+06, HMPJH05] are equipped with rich semantics (such as watch/retry, retry/orElse) and can help enforce some of the above synchronization intentions. It is hoped that this bug characteristics study can help future research to decide the best TM design.

3.6 Other Characteristics

Bug impacts: Among our examined concurrency bugs, 34 of them can cause program crashes and 37 of them cause program hangs. This validates that concurrency bug is a severe reliability problem.

Some concurrency bugs are very difficult to repeat. In one bug report (Mozilla#52111), the reporter complained that "I develop Mozilla full time all day, and I get this bug only once a day". In another bug report (Mozilla#72599), the reporter said that "I saw it only once ever on *g* (never on other machines). Perhaps the dual processor of *g* makes it occur."

Test cases are critical to bug diagnosis. Programmers' discussions show that a good test case to repeat a concurrency bug is very important for diagnosis. In Mozilla bug report #73291, the programmers once gave up on this bug and closed the bug report, because they could not repeat the bug. Fortunately, somebody worked out a way to reliably repeat the bug, and the bug was fixed subsequently. In another Mozilla bug report (Mozilla#72599), the programmers finally gave up repeating the bug and simply submit a patch based on their "guessing", and this led to a wrong fix.

Programmers lack diagnosis tools. From the bug reports, we notice that many concurrency bugs are diagnosed simply by programmers reading the source code. For example, for 29 out of the 57 Mozilla bugs, the bug reports did not mention that the programmers ever leveraged any information from any tools, core dumps, or stack traces, etc. Sometimes programmers tried `gdb`, but could not get useful information. This study has never seen programmers mention that they used any automatic diagnosis tools. In contrast, in many bug reports about memory bugs, programmers mentioned that they got help from Valgrind, Purify, etc [LTW+06].

3.7 Summary

This chapter presents a (best-effort) comprehensive characteristics study of real-world concurrency bugs. This study focuses on bug pattern, bug manifestation, and bug fix strategy of concurrency bugs. The observation is made based on 105 real-world concurrency bugs randomly collected from 4 representative open-source programs: MySQL, Apache, Mozilla, and OpenOffice. The result of this study includes many interesting findings and implications for concurrency bug detection, testing and concurrent programming language design.

Future research can benefit from this study in various aspects. For example, future work can design new bug detection tools to address multiple-variable bugs and order-violation bugs; can pairwise test concurrent program threads and focus on partial orders of small groups of memory accesses to make the best use of testing effort; can have better language features to support "order" semantics to further ease concurrent programming.

The following four chapters will demonstrate how to take motivation and guidance from this characteristics study and improve the state-of-the-art of concurrency bug detection and concurrent program testing.

Chapter 4

Detecting Concurrency Bugs I — AVIO: Atomicity Violation Bug Detection

As discussed in Chapter 1, concurrency bugs widely exist in concurrent programs. Effective approaches to detecting these bugs are critical to the dependability of concurrent software systems. Existing techniques for detecting concurrency bugs mostly focus on data races, which is not enough to completely address the concurrency bug problem. This and the next chapters will present two concurrency bug detection techniques: AVIO and MUVI.

Motivated by previous characteristics study of real-world concurrency bugs (Chapter 3), AVIO and MUVI focus on two types of important yet not well studied concurrency bugs: atomicity violation bugs and multi-variable concurrency bugs. Different from all previous concurrency bug detection tools, AVIO and MUVI automatically infer what interleavings are intended by programmers and report bugs when those intentions are violated. AVIO, an atomicity violation bug detection tool (under technology transfer to Intel), will be presented in this chapter; MUVI, a multi-variable concurrency bug detection tool, will be presented in the next chapter.

4.1 Overview

4.1.1 Motivation

Atomicity, also referred to as serializability, is a property for several concurrently executed actions, when their data manipulation effect is equivalent to that of a serial execution of them[1]. Atomicity violation bugs are caused by violations to the atomicity of certain code regions. Atomicity violation bugs widely exist in the real world. In a previous concurrency bug characteristics study (Chapter 3), atomicity violation bugs contributed to about 70% of the examined non-deadlock concurrency bugs.

In previous works, atomicity violation bugs have not been well studied. Instead, most work has focused on data race detection[2]. This dissertation focuses on atomicity violation bugs instead of data race bugs for following reasons:

[1]The definition of atomicity in this dissertation follows the tradition in concurrent program research, and is slightly different from that in database research community. Serializability in this dissertation refers to view serializability.

[2]A data race occurs when two instructions (including at least one write access) from different threads access the same memory location without proper synchronization. More details are presented in Section 2.2.

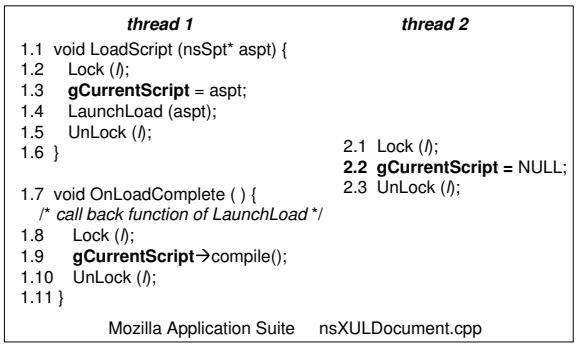


Figure 4.1: An example indicates that data race free does not guarantee correct synchronization. (This example is slightly simplified from a real bug in Mozilla Application Suite. When thread 2 violates the atomicity of thread 1's access to `gCurrentScript`, the program crashes.)

(1) Programmers' most common synchronization intention is atomicity, not data race free. Free of data race may not indicate correct synchronization. Figure 4.1 is a real bug example from the Mozilla Application Suite. As we can see, thread 1 stores a pointer into the shared script handler `gCurrentScript` and wants to retrieve the value to continue processing the script in a later step. However, thread 2 may nullify the `gCurrentScript` in the middle. This example does **not** contain any data race because there exists a lock l protecting every access to `gCurrentScript`. Unfortunately, it still contains a severe concurrency bug: an *atomicity violation*, which will lead to a crash once triggered.

Through the example highlighted in Figure 4.1, an important lesson is demonstrated: programmers, *who are used to sequential thinking*, frequently assume the **atomicity** of code segments. In this example, the two parts of script processing from thread 1 are expected to be atomic, never interfered by other accesses to `gCurrentScript`. Unfortunately, the assumed atomicity is not always satisfied in programmers' implementation. As a result, unserializable interleavings would occur at run time, violate programmers' assumptions, and manifest as concurrency bugs.

The prevalence of programmers' atomicity intentions is one of the reasons that atomicity violation bugs are common in practice. Using locks or transactions is just *one way* to ensure atomicity, but, as demonstrated, data race free does not guarantee proper atomicity.

(2) Data race is not a problem for future transaction-based concurrent programs, but atomicity violation still is. Recently, there has been an emerging trend toward the transactional memory programming model [ATKS07, AAK+05, HWC+04, HF03, Moi97, MBM+06, CTTC06]. Programmers using this model need not worry about data races, however, atomicity violations will still happen when programmers mistakenly put operations that should be atomic into different transactions.

(3) A data race is not always a bug. In many cases, programmers intentionally allow data races on non-critical variables for better performance [YRC05, NWT+07].

(4) Reliance on specific synchronization semantics in race detection causes many false positives. Since the data race concept is tightly bonded with synchronization operations, both the happens-before and locks-set algorithms demand prior knowledge about all synchronization primitives used in the program. Ignorance of non-lock synchronization primitives, such as *barrier*, conditional variables, and many user-defined synchronizations, have caused many false positives in previous work [C+02, SBN+97, YRC05], reducing their effectiveness for programmers.

Unfortunately, although the problem of atomicity violations has been known for years, few good solutions exist to address this important problem. The biggest challenge is that it is hard to know which code regions in a program are intended to be atomic and need to be protected. Most state-of-the-art techniques [FF04] rely on programmers' *annotations* to annotate atomic regions. Recently SVD approach [XBH05] uses data/control dependency to infer atomic regions. Although SVD provides a very inspiring approach, the dependency-based inference only covers a limited subset of atomicity violation bugs. The complicated dependency analysis also incurs large runtime overhead.

4.1.2 Highlights

AVIO is an innovative, comprehensive, invariant-based approach to detecting general atomicity violations bugs. The main idea of AVIO is to automatically discover from correct runs (i.e., training runs) those important code regions that are assumed to be atomic by programmers, and then to detect atomicity violation to those code regions and report bugs at run time.

AVIO's idea is based on the following two novel observations:

(1) AI Invariants There exists a unique type of program invariant that is frequently assumed by programmers: two consecutive accesses from one thread to the same shared variable are never unserializably interleaved by other threads (i.e., always atomic). This invariant is simple yet highly related to synchronization correctness. It reflects a type of fundamental synchronization intention of programmers: are conflicting accesses from other threads welcomed, forbidden, or do-not-care? If such an invariant is not guaranteed in the code implementation and gets violated at run time, a concurrency bug will happen (Section 4.2.1).

(2) How to extract AI Invariants? Generally speaking, as long as there are many different and correct execution samples, program invariants can be observed from these samples. Fortunately, this is exactly the advantage of concurrency bugs. Buggy concurrent programs mostly execute correctly even with bug-exposing inputs because the manifestation of a concurrency bug requires

special and usually low-probability interleavings. In addition, running a concurrent program multiple times, even with the same input, produces many different interleavings. Therefore, it is feasible and relatively easy to obtain AI invariants from correct runs (training runs) (see Section 4.2.3).

Based on above observation, AVIO's idea is instantiated in two designs: an AVIO-H in hardware and an AVIO-S in software. AVIO-H requires *simple* extensions to the cache coherence protocol and achieves negligible overhead. In contrast, AVIO-S is a pure software approach. It is slower but more accurate. With the AVIO-H and AVIO-S designs, AVIO can be used for two scenarios:

(1) Postmortem analysis: Programmers can use AVIO (e.g., AVIO-S) to diagnose the root cause of software failures. Given a failure to diagnose, a programmer can check which AI invariants collected from correct runs are violated in buggy runs.
(2) On-the-fly detection: Programmers can also extract AI invariants during in-house testing, and then use AVIO, especially AVIO-H, during production runs to detect atomicity violations and pinpoint code regions that lack atomicity protection.

The two implementations of AVIO are evaluated using six representative **real** atomicity violation bugs from two large real-world server applications, namely the Apache HTTP Server and the MySQL database server, and an extracted version of Mozilla, running on either real machines (for AVIO-S) or the whole-system simulator Simics [MDG+98] (for AVIO-H). Experimental results show that, compared to previous approaches, AVIO has the following unique advantages:

- **Detects a variety of atomicity violation bugs**. The experiments show that AVIO detects more tested real atomicity violations of various types than previous algorithms (SVD [XBH05], happens-before and lockset). The reason is that AVIO provides a more comprehensive coverage of different types of serializability violations. In addition, AVIO can detect atomicity violation bugs that are not addressed by data race detection.
- **Requires no annotations or specifications**. Unlike many previous approaches, AVIO does not require programmers to provide any specifications about synchronization primitives or atomic regions. Therefore, the AVIO idea applies to not only multi-threaded programs using standard lock-based synchronization, but also to those using application-specific synchronizations as well as future applications written in transactional memory models.
- **Few false positives**. Two factors help AVIO achieve lower false positives than many previous tools. First, AVIO does not rely on specific synchronization primitives and will not report code regions protected by customized synchronization as bugs. Second, sometimes programmers intend to have data races or unserializable interleavings. AVIO can automatically infer these intentions through training and therefore avoid reporting them as bugs. In the experiments, AVIO reported only a few (on average, 3-5) static false positives for our

evaluated applications. In contrast, previous methods have reported an average of 51 false positives, which significantly undermines their bug detection capability because programmers need to sift through 51 error reports in order to find one true bug.

- **Non-stringent requirements for training data generation**. Training is important in all invariant-based approaches [HL02, ZLL+04]. Fortunately, leveraging the non-deterministic characteristic of concurrent programs (i.e., different runs with the *same* input *automatically* produce different interleavings), training data generation in AVIO has unique advantages over previous invariant-based approaches. Results show that running fewer than 5 times for SPLASH-2 benchmarks and less than 100 requests for server applications are good enough to generate a reasonably accurate set of AI invariants (see Section 4.6.3 for a detailed sensitivity analysis).

- **Low overhead**. AVIO, in particular its hardware implementation AVIO-H, imposes very little (0.4–0.5%) overhead, orders of magnitudes smaller than software-based concurrency bug detection tools. The software implementation AVIO-S incurs 15-42 times the overhead, which is still lower than (or comparable with) previous software-based tools such as SVD (65X) [XBH05] and Valgrind-lockset (> 200X).

4.2 AVIO Idea

Terminology definitions For simplicity, unless otherwise mentioned, all accesses are to the same shared memory location. We refer to the thread whose atomicity is interrupted as the *local thread* and its accesses as *local accesses* or *local reads/writes* (note that this does NOT mean a local variable). We refer to the thread with the interleaving access as the *remote thread* and its accesses as *remote accesses* or *remote reads/writes*.

Note that, *interleaving* in general means an execution order among accesses from multiple threads. Our discussion about interleavings in this dissertation assumes the sequential consistency memory model. Additional interleavings that are possible under more relaxed memory consistency models are not considered. This chapter focuses on interleavings among two local accesses and some remote accesses. Specifically, local accesses A and B are *interleaved* by a remote access C when C is executed between A and B. A *serializable* interleaving is an interleaving among local and remote accesses that is equivalent to a serial execution of them.

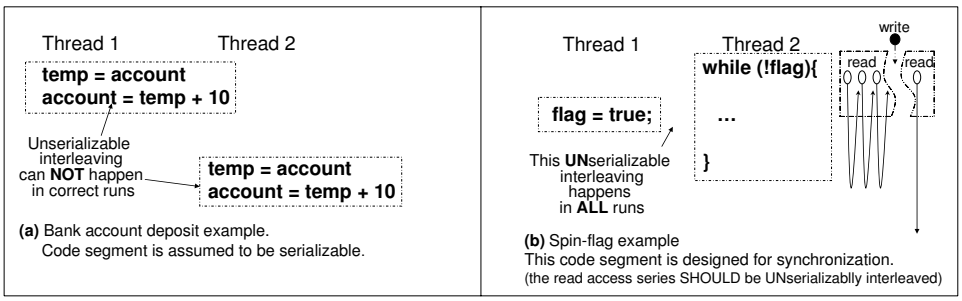


Figure 4.2: (a) An example with AI invariant; (b) An example without the AI invariant

4.2.1 Access-Interleaving Invariant

The *essence* of atomicity violation bugs is no different than other types of bugs: they are caused by a mismatch between the code implementation and the programmer intention. Specifically, programmers assume that a sequence of shared variable accesses is atomic, never interleaved by unserializable accesses, but the implemented code does not guarantee this property and thus bugs emerge.

Programmers' atomicity intention comes in different formats. The most common and fundamental one can be represented by a type of invariant that we refer to as an **Access-Interleaving invariant** (AI invariant). Such an invariant is held by an instruction if the access pair, composed of itself and its preceding local access to the same location, is never *unserializably* interleaved. We denote this instruction as **I-instruction** (invariant instruction) and the preceding access instruction as **P-instruction** (preceding instruction). Note that with an AI invariant, it is perfectly OK to have interleavings. Atomicity would be maintained as long as the interleavings are serializable. Section 4.2.2 will further discuss interleaving serializability.

Figure 4.2(a) gives a simple demonstration of an AI invariant using the classic banking account example. In this code, programmers assume that the read and modification of $account$ are always together and never be unserializably interleaved by a conflicting remote access. Otherwise, an atomicity violation can result in program misbehavior.

An AI invariant indicates programmers' atomicity assumption. Such assumption is the essence of concurrent execution correctness. Atomicity assumptions, as in the banking account example, are made during design and implementation by programmers who are more comfortable with sequential thinking. Assumptions may be enforced through locks, barriers, flags or other synchronization mechanisms such as transactions. Poorly enforced atomicity assumptions cause synchronization errors during some executions.

We should note that programmers do *not* assume all code regions to be atomic, nor does AI invariant held for every shared variable access instruction. Contrarily, some instructions often *allow*

unserializable interleavings. For example, in cases like flag-based synchronization implementation, programmers explicitly do not want an AI invariant. Automatically differentiating code that is or is not expected to have AI invariant would allow us to avoid many false positives. For example, Figure 4.2(b) [3] illustrates synchronization implemented using a flag variable. During execution, no AI invariant will be observed at the `flag` read access of the while loop, because unserializable interleavings happen in every run. Such AI non-invariant matches the programmers' intention in this example at this position: an unserializable interleaving by a remote access is *required* to ensure liveness.

Of course, AI invariant is not the only format of programmers' atomicity assumption, but it is the most common and fundamental one. Other assumptions involving multiple shared variables are rarer and can be potentially extended from AI invariants. We will discuss them in Section 4.7.

In summary, atomicity violation bugs are code regions that are expected to be atomic but implemented as non-atomic. At run time, serial execution or serializable interleavings definitely maintain the atomicity; meanwhile, unserializable interleavings do NOT necessarily violate correctness. Which part of code needs to be atomic and serializable depends on programmers' intentions and is well indicated by AI invariants. Therefore, if we can automatically extract AI invariants, this knowledge can then be used to detect atomicity violation bugs by monitoring "unexpected" unserializable interleavings in code segments where AI invariants should hold.

4.2.2 Serializability Analysis

Not all interleavings are unserializable, and serializable interleavings do not lead to atomicity violation. In this section, we first analyze what interleavings are serializable and what are not. There are totally eight ways that two consecutive local accesses to the same shared variable can be interleaved by a remote access. Table 4.1 describes every cases, explaining why each case is serializable, with equivalent serial accesses, or unserializable, with a bug example.

Among the eight cases, four (cases 0, 1, 4, 7) are serializable interleavings while the other four (case 2, 3, 5, 6) are not. We have an example bug for each unserializable case in which programmer's assumptions about atomicity is violated. For example, Figure 4.3 gives a real bug from the Apache httpd server whose root cause is a case 2 unserializable interleaving. Figure 4.4 shows a real bug example from the MySQL database server for case 5. Similarly, case 3 and case 6 are exemplified by the examples shown early in Figures 4.1 and Figure 4.2(a). But note that, as discussed in the previous section, unserializable interleavings are not necessarily bugs (Figure 4.2(b)) unless they violate programmers' assumptions.

[3] Strictly speaking, the while-flag in the figure is incorrect without memory fences or hardware-supported atomic instructions. Since this issue does not affect our discussion of AI invariant, we ignore it in the figure.

Interleav -ing	Case #	Description	Serializability	Equivalent serial accesses	Problems (for unserializable cases)	Bug Example
read^p $read_r$ read^i	0	two reads interleaved by a read	serializable	read^p read^i $read_r$	N/A	N/A
write^p $read_r$ read^i	1	read after write interleaved by a read	serializable	write^p read^i $read_r$	N/A	N/A
read^p $write_r$ read^i	2	two reads interleaved by a write	*unserializable*	N/A	The interleaving write gives the two reads different views of the same memory location	Apache Figure 4.3
write^p $write_r$ read^i	3	read after write interleaved by a write	*unserializable*	N/A	The local read does not get the local result it expects	Mozilla Figure 4.1
read^p $read_r$ write^i	4	write after read interleaved by a read	serializable	$read_r$ read^p write^i	N/A	N/A
write^p $read_r$ write^i	5	two writes interleaved by a read	*unserializable*	N/A	Intermediate result that is assumed to be invisible to other threads gets exposed	MySQL Figure 4.4
read^p $write_r$ write^i	6	write after read interleaved by a write	*unserializable*	N/A	The local write relies on a value that is returned by the preceding read and becomes stale due to the remote write	Bank account Figure 4.2 (a)
write^p $write_r$ write^i	7	two writes interleaved by a write	serializable	$write_r$ write^p write^i	N/A	N/A

Table 4.1: Eight cases of access interleavings. (Accesses in each case are towards the same variable. Besides read/write, subscript r denotes remote interleaving access; superscript i and p denotes one access and its preceding access from the same thread. In this chapter, normal instructions and invariant-related instructions are differentiate by lower-case i/p and upper-case I/P.)

Above we get the unserializable condition, composed of four cases, for *single* interleaving remote access. Extending it, we get following similar four-case unserializable conditions with *multiple* remote accesses to the same shared variable taken into account. *This condition will be used in the rest of the chapter, guiding AVIO bug detection* (For illustration, interleaving remote accesses are put in parentheses; $*$ denotes zero or multiple interleaving read or write accesses; superscript i and p stand for one access and its preceding access from the same thread):

- Case2: $r^p[*w_r*]r^i$, two local reads are interleaved by *at least one* remote write, so they may have different views.
- Case3: $w^p[*w_r*]r^i$, a local read after write is interleaved by *at least one* remote write. Due to this remote write, the read would fail to get the local result that it expects.

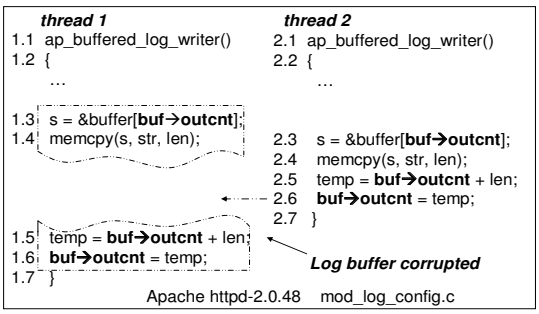


Figure 4.3: A real bug from Apache httpd server caused by the case 2 unserializable interleaving. (In this example, the two read accesses on lines 1.3 and 1.5 belong to the same buffer filling operation and are intended to have the same view of buf→outcnt. However, they can be interleaved by the write access on line 2.6, under which the Apache server log associated with `buffer` would be corrupted. This example also contains a potential case 6 unserializable interleaving between 1.5 and 1.6.)

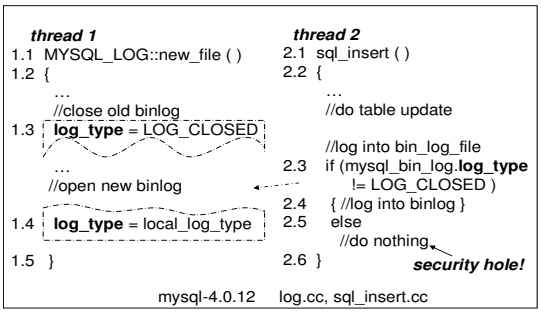


Figure 4.4: A real bug from MySQL database server caused by the case 5 unserializable interleaving. (In this example, the write accesses on lines 1.3 and 1.4 are interleaved by the read access on line 2.3. As a result of this interleaving, thread 2 reads an intermediate value produced by thread 1, which causes the database operation from thread 2 unrecorded in the log and generates a security vulnerability.)

- Case5: $w^p[r_r*]w^i$, a local write after write is interleaved by a remote access sequence that *starts with* read, making the local intermediate result visible to a remote thread.
- Case6: $r^p[*w_r*]w^i$, a local write after read is interleaved by *at least one* remote write. It makes the previous reading result stale.

4.2.3 Automatically Extract AI Invariants

A challenging question is how to obtain AI invariants, knowing which code regions do not welcome unserializable interleavings. In this section, we describe the high level idea used in AVIO. The detailed process will be given in section 4.3.2.

Obviously, we cannot expect programmers to provide such invariants because atomicity violations usually occur in code segments where programmers are not consciously aware of their

assumptions. Similarly, we cannot use lockset analysis to extract AI invariants without suffering from the same limitations (discussed in Section 4.1) as previous lockset based algorithms.

To automatically learn a programmer's intention, the best way is to study the program's behavior in correct execution: if a code segment is always serializable in correct runs (runs where no bug manifests), it is probably assumed to be so always. In other words, we can *statistically* "learn" a program's AI invariants through training. Specifically, to collect and analyze access interleavings from a set of correct runs (**training runs**), we can see which shared accesses (such as the one in Figure 4.2(b)) allow unserializable interleavings, and which shared accesses *never* have unserializable interleavings.

The feasibility of the above idea depends on how well the training can be: (1) How to ensure training is dominated by *correct* runs (correctness issue)? (2) How to get *sufficient different* training samples (sufficiency issue)? These two issues are critical in all invariant-based techniques [ECGN00, HL02, ZLL+04]. Fortunately, two unique and "notorious" properties of concurrency bugs make training in AVIO easier than general invariant training. In other words, we have turned the negative "troublesome" bug characteristics into positive characteristics in detecting these types of bugs.

First, the **correctness** issue is addressed by two facts: (1) Concurrency bugs manifest very infrequently, even with bug-exposing inputs. Their manifestation usually requires specific access interleaving, a notorious feature that makes concurrency bug very hard to reproduce for postmortem diagnosis. Practical experience with real bugs shows that, even with bug-triggering inputs, usually it still takes hundreds, thousands, or more of repeated executions to trigger a bug. As a result, we can easily get correct-dominated training. (2) Existing infrastructure and research in software testing can be leveraged to label training runs as correct or incorrect. In particular, according to a previous work [CC98], most concurrency bugs are fail-stop. Furthermore, software testers usually have various methods (beyond crashes or hangs) during in-house testing to determine the correctness of test runs. Additionally, assertions and automatically extracted predicates [LAZJ03] can further help to filter out incorrect training runs. The AI extraction algorithm can also be designed to tolerate a small percentage of unfiltered incorrect training runs.

Second, the **sufficiency** issue is addressed by the fact that concurrent execution is **non-deterministic** due to the underlying thread interleaving. As we all know, both multi-processor execution and operating system thread scheduling have a lot of randomness. As such, even with just one input, we can easily get a large number of distinct access interleavings. For example, in our experiments, 100 runs of a SPLASH-2 benchmark with just one input always generate 100 different traces. Of course, better design together with interleaving perturbation can potentially achieve more effective training. More discussion about how to effectively exercise the interleaving space can be found in Chapter 7.

Benefiting from the *non-determinism*, training in AVIO's postmortem analysis (usage model 1) is very easy. Just running the program with the bug triggering input many times, and we will get sufficient access-interleaving training results. This is a big advantage over traditional invariant-based tools. As for on-the-fly-detection (usage model 2), the capability of AVIO is related to its path coverage, which is a problem for *all* dynamic bug detection tools, not only for invariant-based techniques. In AVIO, if the training does not cover a particular code block, no AI invariant is available there and false negatives may occur. We need to rely on a reasonable branch coverage of in-house testing suite and AVIO can be extended to actively learn new invariants during detection. Value coverage is less of a concern, because AI invariants are associated with instructions and interleavings, not with data addresses or values. With a different input, an instruction may access data with a different value, but the programmer's assumption about the desired atomicity associated with this instruction remains the same.

The above analysis indicates it is feasible to extract AI invariant by training. Our experiments further validate this. All the real server bugs detected by AVIO are based on training with just one input and fewer than 100 training requests. The training input value is also flexible, as indicated in our experimental input sensitivity study (section 4.6.3).

4.3 AVIO Algorithms

AVIO automatically extracts AI invariants from off-line testing runs, and then detects potential violations to the extracted AI invariants during monitored runs. As the AI invariant extraction algorithm is based on the detection algorithm, we will first describe the detection algorithm and then present the extraction algorithm.

4.3.1 Detection Algorithm

Suppose that we already have a set of AI invariants, which gives a list of I-instructions. Then an AI invariant violation is an unserializable interleaving between an I-instruction and its preceding local access instruction (P-instruction) to the same shared variable. Based on our serializability analysis in section 4.2.2, to detect any such unserializable interleaving, the detection process can simply follow the binary decision diagram in Figure 4.5, which summarizes all the four unserializable interleaving cases.

The decision diagram in figure 4.5 clearly shows that AVIO needs four pieces of information to tell unserializable interleavings from serializable ones. These four pieces of information are: access type of the current instruction (i.e., I-instruction Type); access type of the preceding local instruction to the same memory location (P-instruction Type); interleaving remote write informa-

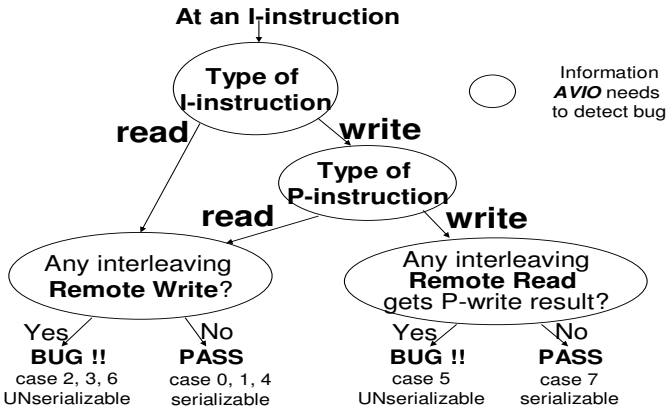


Figure 4.5: AVIO bug detection procedure (This diagram can be better understood when referring to table 4.1).

tion and interleaving remote read information. With these four pieces of information, AVIO can easily detect violations to AI invariants. We will show later in Section 4.4 how these four pieces of information are collected in both our hardware and software implementations of AVIO.

4.3.2 Extraction Algorithm

The goal of the AI invariant extraction, referred to as AVIO-IE, is to extract AI invariants from multiple correct runs.

Interestingly, AVIO-IE can be easily implemented by leveraging the AI invariant violation detection process. Specifically, the extraction process is a series of correct runs with the AVIO detection enabled. As shown in Figure 4.6, initially the set of AI invariants, $AISet$, includes all global memory accesses in the target program. Then it runs the program on top of AVIO multiple times. At the end of each run, AVIO reports "violations" to the current $AISet$ in this run. A violation at an instruction i indicates that an unserializable interleaving is encountered in the current *correct* run (labeled by the testing oracle). Therefore, there is no true AI invariant at i; i should be removed from $AISet$. This process will repeat many times until $AISet$ remains unchanged for the last m runs, where m is adjustable. In Section 4.6, we will show sensitivity results of the number of training runs. Finally we filter out never-executed instructions and return $AISet$.

To tolerate a small percentage of incorrectly labeled training runs (i.e., an incorrect run is labeled as correct), AVIO-IE can introduce an invariant filtering threshold T. Only when an invariant is violated in more than T training runs that pass the testing oracle, this invariant is removed from the $AISet$. This technique can avoid some actual invariants being filtered due to some incorrectly labeled training run, but at the cost of potentially more false positives in violation detection. So the best way is for programmers to adjust the threshold parameter based on the accuracy of their testing oracles as well as their false positive tolerance level.

```
                                                              Script
AVIO-IE (ProgramBinary P)
{
  AISet = all global memory accesses in P;
  while (AISet is changing in the last m iterations) {
        ViolationSet = RunOnceWithViolasDetection (P, AISet);
        AISet = AISet – ViolationSet;
  }
  AISet = AISet – NonTouched Instructions;
}
```

Figure 4.6: The process of extracting AI invariants in AVIO.

4.4 Two AVIO Implementations

To study the trade-offs between hardware and software, we implement our AVIO idea and algorithms in two different approaches: a software-only approach AVIO-S and a hardware-assisted approach AVIO-H. As the AI invariant extraction is done during in-house testing, it is less overhead critical. Therefore, extraction is implemented based on AVIO-S.

4.4.1 Hardware AVIO (AVIO-H)

AVIO-H Overview

The hardware implementation, AVIO-H, takes advantage of existing cache coherence protocol and achieves negligible overhead and little execution perturbation with simple hardware extensions as shown in Figure 4.7. AVIO-H currently assumes a CMP machine with a physical-address indexed private-L1 cache and a unified-L2 cache hierarchy, using an invalidation-based cache coherence protocol. Extending it to other multiprocessor architecture such as SMP and other cache coherence protocols is relatively straightforward, especially since our detection algorithms described in Section 4.3 are not implementation specific.

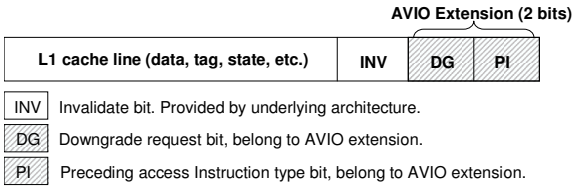


Figure 4.7: AVIO-H's extension to each L1 cache line.

First, AVIO-H appends each L1 cache line with two new access-information bits. These new bits, together with the existing invalidate (INV) bit used by the cache coherence protocol, provide enough information to perform the AVIO detection algorithm described in Section 4.3:

- **PI bit** (Preceding access Instruction bit): This bit provides the "Type of P-instruction" information. It is set to 1 at each local read to the corresponding cache line and is unset at each local write.
- **DG bit** (Downgrade bit): This bit provides information to find out whether the previous local write's result has been read by a remote thread. Interestingly, in existing invalidation-based cache coherence protocols, such an action is associated with a *Downgrade* request sending from the reader to the recent writer. Therefore, AVIO-H just needs to set the DG bit upon a *Downgrade* request and unset the bit after each local access.
- **INV bit**: This bit already exists in current cache coherence hardware. It provides information about any "interleaving remote write" after the previous local memory access. In existing invalidation-based cache coherence protocol, interleaving remote writes will invalidate all other L1 caches' copies. Therefore, AVIO-H just needs to check the INV bit to see whether a remote write has happened.

Second, the hardware cache coherence protocol is extended to support the above information bits and violation detection. Finally, we add special instruction encodings for I-instructions (reads and writes) and a special bit in the L1 cache access command to indicate when a memory instruction is an I-instruction. Using these extensions, we can easily implement the detection protocol in hardware as shown in figure 4.8.

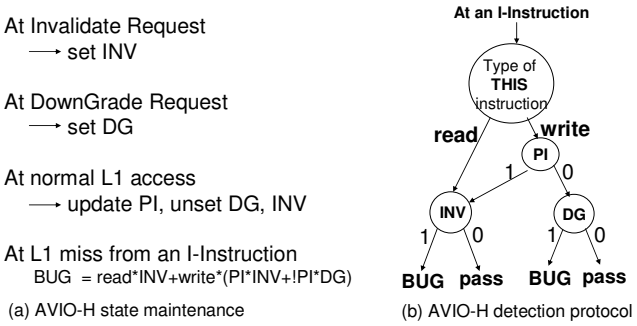


Figure 4.8: AVIO-H state maintenance and bug detection (a hardware version of figure 4.5).

Complexity and Overhead Both the state maintenance and bug detection in the AVIO-H have very simple logic, as shown in figure 4.8. Interestingly, further studying the detection protocol indicates that unserializable interleaving only happens when the original cache coherence protocol cannot use the local copy and needs to contact L2 to get the most-up-to-date copy and/or exclusive write permission. Therefore, AVIO-H's detection process is triggered only when an I-instruction cannot be satisfied by its local L1 cache.

The whole detection phase has small space overhead and negligible time overhead. The extra space is just two bits per L1 cache line, less than 0.4% overhead. Because the invariant check is conducted only when an I-instruction has to go to the shared L2 cache, the check is not in the critical path–the simple detection protocol can be hidden by the L2 cache access latency. Only when a bug is found, AVIO-H needs to impose overhead recording it.

Design Issues of AVIO-H

After describing AVIO-H's basic mechanism, this section discusses some design issues. Some of these issues have no effect on AVIO-H, while some others can, in rare cases, affect AVIO-H's accuracy in ways similar to previous hardware data race detectors. All these issues are specific to our hardware implementation, and do not affect our software implementation (described in Section 4.4.2).

Recording and reporting atomicity violations After detecting an AI invariant violation, AVIO-H marks the I-instruction in the reorder buffer and sends a signal when this instruction retires. Therefore, no bug is reported for speculative instructions. AVIO-H supports two bug reporting options: either break the execution with an exception, or only record the I-instruction's PC and accessed address to a memory location specified by the software.

Cache line displacement and context switch Recent access history of a cache line may be lost when it is displaced. This problem is also encountered in most previous hardware race detectors [Prv06, PT03] and was simply ignored because it only results in false negative in very rare cases. This is especially true for AVIO because AI invariants focus on two *consecutive* accesses to the same memory location from the same thread. Intuitively, these two accesses are nearby (which is why programmers forget to protect them in the first place) and therefore the probability for them to be interleaved by a displacement of an involved cache line is very small. In addition, we can always postpone displacing such a cache line by first evicting a private (e.g., a stack) cache line. Similarly, context switches can also create some false positives in AVIO-H as well as most previous hardware race detectors, and its probability is also very low for similar reasons. These issues can be addressed in the future by thread-id tagging or employing directory as victim buffer in directory-based cache coherence protocol.

Load-store queue and write-coalescing Some read access may be invisible to AVIO-H if they hit the load-store queue. Fortunately, if this access is an I-instruction, a hit in load-store queue definitely indicates *no remote interleaving* between this read and previous local access, so it is perfectly fine that AVIO-H is not checking this "invisible" access. For the same reason, write-coalescing also has no effect on bug detection. But if this access is a P-instruction right before an I-instruction, it can lead to a two-fold effect. On the positive side, AVIO-H may thus be enabled to

detect atomicity violation to a larger code region, since it mistakes an access sequence $w^{P1}r^{P2}r_rw^I$ by $w^{P1}r_rw^I$, with r^{P2} hit the load-store queue and invisible to L1 cache. On the negative side, $w^{P1}r^{P2}w_rw^I$ may be mistaken as $w^{P1}w_rw^I$, and AVIO-H may miss the bug. In summary, in most cases, the load-store queue has no effect; in a very small percentage of cases, it may either help or harm AVIO-H in detecting some bugs. Note that similar issues are faced by previous hardware race detectors. Previous solution forces global memory accesses to go through the lower memory hierarchy [RL98]. Since this issue rarely has bad effect on AVIO-H, we did not choose this solution in our current prototype.

Strict/weak consistency model, out-of-order access and execution issues Different memory consistency models may cause different memory access orders for the same concurrent program. However, it does not affect the bug detection. No matter what the access order is, what AVIO-H sees is the *actual* order executed on hardware. Out-of-order execution similarly has no effect on AVIO-H. The only exception here is when prefetching results are finally discarded *and* the access interleaving matches case 5, which is a low probability event, there may be some false positives.

False sharing due to cache line granularity In our design AVIO-H uses a cache line as the unit for information keeping and bug detection. It may introduce some false sharing, an issue also faced by previous hardware race detectors [PT03]. It can be solved by simply using a smaller granularity (e.g., word) at the expense of increasing space overhead and bus traffic. This problem can also be alleviated by using profiling to find false sharing and then using a compiler to automatically add paddings. Such processes have already become a standard optimization to reduce unnecessary cache coherence traffic and cache misses for performance reasons.

Other sources of cache line invalidation In AVIO-H, we use each cache line's INV bit to record remote write access information. In addition to cache coherence invalidation, this bit can also be set by other sources such as DMAs. This does not interfere AVIO-H's bug detection capability. Atomicity violation would still be correctly reported even though it may be from a DMA operation.

Support for SMT Our current AVIO design is based on CMP/ SMP. To support SMT, AVIO needs a simple extension: tag L1 cache lines with thread-ids.

Compatibility with certain processor and cache coherence protocol The cache management policy required by AVIO is general: an invalidation-based cache coherence protocol. However, there may be some real processor that is incompatible with AVIO's current prototype. In that case, AVIO needs simple extension to the cache coherence protocol to get some bug detection required information, such as downgrade information.

4.4.2 Software AVIO (AVIO-S)

To study the trade-offs between efficiency and accuracy, we also implemented the AVIO techniques purely in software.

Like AVIO-H, the key task of AVIO-S is to collect and maintain access information required by the AVIO detection protocol. Specifically, for all global memory, AVIO-S maintains the most recent local and remote access history information, and then uses it to check for possible violations at I-instructions. In software, various access information is collected by binary instrumentation at every global memory access and maintained in an access-table data structures. Each thread has an access-table, holding the type information of its latest access to each global memory location. There is also a global access-owner-table, holding the identifier of the thread that most lately wrote to each global memory location. At each memory access from I-instruction, the P-Instruction Type can be obtained from the local-access table; and the information about remote write and read can be inferred and bookmarked by comparing local thread-id with the owner-id.

Once an atomicity violation is detected, as in AVIO-H, AVIO-S will either stop the program and raise an exception, or log all the debugging information and continue the execution. Debugging information, such as the address of the three involving instructions, P-instruction, I-instruction and the remote interleaving access, can be recorded in the global and local access tables.

4.4.3 Trade-offs between AVIO-H and AVIO-S

AVIO-H and AVIO-S each have their own advantage and disadvantages. First, AVIO-S is cheaper because it does not require any hardware extensions. Second, AVIO-S is also more accurate because: (1) AVIO-S's detection granularity is very flexible, ranging from a byte to a cache line with a word as default. Therefore, AVIO-S suffers much less from the false sharing problem than AVIO-H. (2) Since AVIO-S monitoring and detection are done by instrumented code, it is not affected by the cache displacement, load-store queues, context switches, or other hardware-related issues.

However, as a trade-off, AVIO-S incurs much higher overhead and run-time perturbation, which comes from two sources. The first is monitoring overhead–each global access is instrumented to update the access information of the accessed data. The second is detection overhead. At each I-instruction, AVIO-S needs to detect possible violations to the corresponding AI invariant. To reduce overhead, hashing is used for quick locating the information tables. Since the global owner-table can be accessed by all threads, spin-locks are used for fast synchronization. All these optimizations are helpful in reducing overhead. However, as we will show in the experimental results (section 4.6), even though AVIO-S' performance is better than several other software concurrency bug detectors, the overhead is still much higher (4 orders of magnitude) than AVIO-H. Even if static analysis may further optimize AVIO-S, it is still too hard to reach the level to fit for

production run as AVIO-H. In addition, the bug detection capability may also be affected by the larger execution perturbation from AVIO-S.

4.5 Methodology

Our software implementation, AVIO-S, is implemented using the PIN binary instrumentation tool [LCM+05] and runs on a *real machine* with four Intel Pentium processors. Our hardware implementation, AVIO-H, is implemented on the Simics [MDG+98] whole system simulator, based on the SimFlex timing model [HSW+04] because it can run a real OS on the top, allowing us to run real-world server programs in a realistic simulation environment. Specifically, we use a full system, cycle-accurate, x86 simulator that models a 4-core CMP in-order x86 machine. Non-memory operations have a fixed one cycle latency and memory operations go through the cache and memory hierarchy. The parameters of the architecture are shown in Table 4.2. With AVIO-H, we assume a 0.4% extra slow down on whole chip frequency due to the 0.4% larger L1 cache and 500 cycle penalty at each bug report to stall pipeline and prepare debugging information.

CPU	2.0 GHz in-order; 1 issue width each core
L1 cache (private)	32K, 4 way, 64B/line, 2 cycle latency
L2 cache (shared)	1M, 8 way, 64B/line, 10 cycle latency
Memory	200 cycles latency
Cache coherence protocol	Derived from Piranha [BGM+00] CMP cache coherence protocol

Table 4.2: AVIO-H Simulation configuration.

Two sets of applications are used in our experiments. The first set is used to evaluate AVIO's bug detection capability. Unlike many previous hardware race detection studies [Prv06, PT03] that evaluated with manually injected bugs, we use six *real* atomicity violation bugs which were unintentionally introduced by the original programmers in two large real-world server applications (Apache and MySQL) and Mozilla[2]. Table 4.3 shows the buggy applications and the description of the six real bugs. For these applications, we evaluate whether the bug can be detected and how many false positives are reported during the bug manifestation run.

In the second set, we use several well known SPLASH-2 benchmarks to evaluate AVIO's overhead and false positive. SPLASH-2 has also been used in many previous works [PS03, PT03, RL98] to evaluate false positives because they have few concurrency bugs.

Besides comparing our two implementations AVIO-S and AVIO-H, we directly compare the false positives, negatives and overhead with an enhanced lockset algorithm implemented in Val-

[2]Since our instrumentation and simulation tools do not support Mozilla's graphic user interface, we use an extracted version of the real bug in Mozilla based on its nsXULDocument.cpp file.

Application	BugNo.	Bug description
Apache HTTP server (253K LOC)	#1	Unprotected buffer length read and write corrupt log file (Figure 4.3)
	#2	Unprotected reference counter write-read causes null pointer reference
MySQL DB server (688K LOC)	#1	Unprotected database bin log close and open cause some actions not logged (Figure 4.4)
	#2	Unprotected query-id set and read crashes database server
	#3	Unprotected 'delete table' query and logging causes database log disorder
Mozilla-extract[2]	#1	Unprotected script handler set and read causes null pointer reference (Figure 4.1)

Table 4.3: Applications and atomicity violation bugs evaluated in AVIO.

grind [NS07], which we will refer to as Val(grind)-Lockset algorithm. In addition, we also compare indirectly with the happens-before algorithm and the SVD algorithm [XBH05] by analytically evaluating whether each bug can be detected by them based on our understanding of these two algorithms. In terms of false positive and overhead, we refer to previous papers: happens-before has similar level of overhead with lock-set algorithm; SVD reports an up to a 65X server application overhead and 1-60 static false positives for the same server applications, MySQL and Apache.

To demonstrate the less stringent requirement of AVIO on training runs, *we do not use same inputs for detection and training in our experiments*. To extract AI invariants, we examine multiple access interleavings during 100 training runs (or 100 server requests) for each application. The invariant filtering threshold T is set to 0. In addition, we also conduct sensitivity studies on the number of training runs for both server applications and SPLASH-2 benchmarks. The result shows that no more than 100 server requests or 5 training runs are enough to obtain reasonably accurate AI invariants for all the tested applications.

4.6 Experimental Results

4.6.1 Functional Results

(1) Bug detection capability AVIO detects more tested real bugs than the three alternatives (Val-Lockset, happens-before and SVD). Specifically, as shown in Table 4.4, AVIO can detect five out of the six tested bugs, while the three alternatives can detect only one or three.

MySQL bug3 requires atomicity among accesses to multiple global variables. These variables have no data or control dependency with each other, but must be consistent for semantic reasons. As a result, it is not detected by any evaluated tool. Besides this bug, the Lockset algorithm cannot detect the Mozilla-extract bug, because it is data-race free, as explained in Figure 4.1. For

the same reason, the happens-before algorithm also fails to detect it. SVD cannot detect MySQL Bug1, because it is an atomicity violation involving a write-after-write access pair, with no true data dependency or control dependency within it. The pair are therefore not be put into one computation region and consequently not checked by SVD. Similarly, Apache Bug2, MySQL Bug2 and Mozilla-extract are atomicity violations with write-then-read access pairs, which are also not checked automatically by SVD.

In contrast, AVIO's bug detection capability is more comprehensive because, unlike race detectors, it does not rely on synchronization primitives; unlike SVD, it can detect atomicity violations with write-read and write-write dependencies based on our serializability analysis.

(2) False positives Table 4.5 shows that AVIO introduces only 1–11 static and 1–17 dynamic false positives on server applications, much fewer than the Lockset algorithm, which has on average 51.5 static and 118.5 dynamic false positives. Similarly, for the bug-free SPLASH-2 benchmarks, AVIO-S has no false positive and AVIO-H has only an average of 1.25 static and dynamic false positives, while Lockset has 8.25 static and 26313 dynamic false positives on average.

The reason for lockset algorithm's high false positive rate is that, as discussed in Section 4.1, it incorrectly reports all shared accesses that are correctly synchronized using non-lock based methods, such as barriers and flag-synchronizations, as bugs. Even though we do not evaluate false positives with the happens-before algorithm, we expect that the results will be similar because the happens-before algorithm would use similar knowledge of synchronization primitives to order execution segments and thereby suffer the same problem as the lock-set algorithm we evaluated.The large number of false positives in the previous algorithms requires much effort from programmers to sift through manually.

In contrast, AVIO reports many fewer false positives because it does not rely on any synchro-

Application	Bug Detected				
	AVIO-H (Hardware)	AVIO-S (Software)	Val-Lockset	Happens -before	SVD
Apache #1	Yes	Yes	Yes	Yes	Yes
Apache #2	Yes	Yes	No	No	No*
MySQL#1	Yes	Yes	Yes	Yes	No*
MySQL#2	Yes	Yes	Yes	Yes	No
MySQL#3	No	No	No	No	No*
Mozilla-extract	Yes	Yes	No	No	No*

Table 4.4: Bug detection results for server/client applications. (* Since not evaluated in the SVD paper, these four bugs are evaluated based on our understanding of the SVD algorithm. Specifically, SVD cannot detect these bugs because these bugs involve write-write, write-read dependencies, or accesses to multiple unrelated variables.)

Benchmark	Dynamic False Positive			Static False Positive		
	AVIO-H	AVIO-S	Val-Lockset	AVIO-H	AVIO-S	Val-Lockset
Apache #1	6	5	6	3	2	6
Apache #2	1	1	23	1	1	20
MySQL#1	4	4	107	4	4	79
MySQL#2	17	6	338	11	6	101
Average	7	4	118.5	4.75	3.25	51.5
fft	1	0	4098	1	0	6
fmm	4	0	389	4	0	12
lu	0	0	65026	0	0	5
radix	0	0	35740	0	0	10
Average	1.25	0	26313	1.25	0	8.25

Table 4.5: False positives for server applications and bug-free SPLASH-2 benchmarks. (Dynamic false positives are dynamic instances of false positives reported during execution; static false positives are static code segments incorrectly reported as bugs. Since Mozilla-extract is extracted from Mozilla by us, its false positive number is not objective and not reported here.)

nization primitives. Instead, it bases its detection on access interleavings, which are more essential and fundamental to atomicity violation bugs. Correctly synchronized accesses are not reported as bugs no matter what synchronization methods are used because they do not violate AI invariants. Moreover, AVIO can easily differentiate benign atomicity violations from true bugs because benign violations do not have any AI invariants (i.e., these code segments actually welcome unserializable interleavings). Therefore, AVIO does not report bugs at these code points.

AVIO still has a few false positives. For software AVIO, the false positives are due to insufficient training. Since server applications are very complicated, some correct interleavings do not occur during our short training (only 100 client requests). For hardware AVIO, apart from insufficient training, the reason for most false positives is false sharing at the cache line granularity. Unlike the lockset algorithm, AVIO never has huge numbers of dynamic false positives even when the static false positive rate is comparable to the Lock-set algorithm's, because most of AVIO's false positives occur due to rare interleavings or code paths.

(3) Comparison of AVIO-S and AVIO-H Functionality As AVIO-S has a much smaller granularity than that of AVIO-H, AVIO-S is more accurate. For example, it incurs up to 5 fewer static false positives than AVIO-H. But as shown in the next section, this improved accuracy is achieved with much higher overhead.

Benchmark	Bug Detection Execution Slow Down		
	AVIO (Hardware)	AVIO (Software)	Valgrind-Lockset
fft	0.5%	42X	1217X
fmm	0.4%	19X	660X
lu	0.4%	23X	661X
radix	0.4%	15X	236X
Average	0.4%	25X	694X

Table 4.6: Overhead results for SPLASH-2 benchmarks.

4.6.2 Overhead Results

AVIO has low detection overhead due to the hardware support and the simple detection algorithm we use. As shown from our experiments on SPLASH-2 benchmarks (table 4.6), AVIO-H imposes only 0.4-0.5% overhead, clearly feasible for production run use. Without hardware support, the software implementation AVIO-S imposes an average of 25 times slow down. Although this is acceptable for in house testing or postmortem analysis with deterministic replay support [NPC05, XBH03], it is too high for production runs. We expect its overhead would still be substantially higher than AVIO-H even after aggressive static analysis optimizations.

Though much worse than AVIO-H, our software implementation AVIO-S still outperforms these previous software approaches. As shown in Table 4.6, the Valgrind-lockset imposes an average of 694 times slow down[3]. As the original SVD paper [XBH05] reports, SVD imposes a factor of 65 times slow down to server applications. Such performance advantage is mainly due to the simplicity of our bug detection algorithm. In the future, static analysis can be used to further improve the performance of AVIO-S.

In summary, AVIO-S would be a good choice for off-line bug detection and diagnosis, while AVIO-H can be employed for production runs.

4.6.3 Training Sensitivity

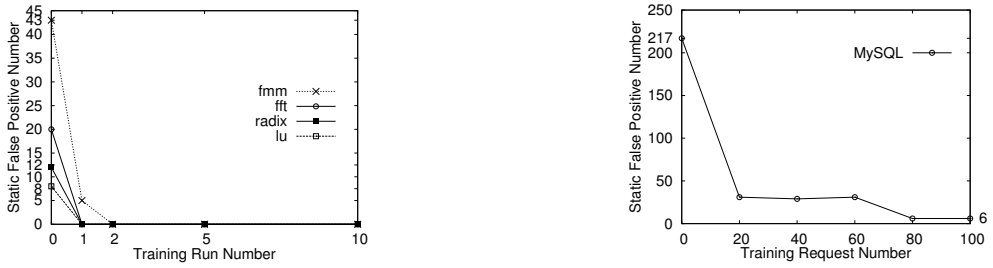
Our sensitivity study results show that for most applications, a few runs are sufficient to get a set of reasonably accurate AI invariants, which are also robust to different inputs. Figure 4.9 shows the number of false positives reported when we use the invariants generated from a different number (1-10) of training runs for the SPLASH-2 benchmarks, or a different number (1-100) of requests for MySQL.

In all four SPLASH-2 benchmarks, the false positives drop to 0 when we use the invariants extracted from more than two training runs. Similarly, with MySQL, most false positives are

[3]Part of the slow-down is from Valgrind's code emulation mechanism.

eliminated after just 100 training requests. Such results indicate that AVIO's training requirements are not stringent.

As mentioned before, in all our experiments, the inputs used in detection runs are different from those in training runs. Therefore, our results also show that training with one input can be used to guide bug detection with other inputs. Of course, similar to other invariant-based approaches [HL02, ZLL+04] and general dynamic bug detectors, AVIO can only generate invariants from exercised code.



(a) SPLASH-2 Benchmarks (b) MySQL 4.1.1-alpha

Figure 4.9: Training effects on static false positives for SPLASH-2 benchmarks and MySQL server. (Note that Figure (a) and (b) use scales for x-axis.)

4.7 Discussion: Limitations of AVIO

AVIO is definitely not a panacea. the current AVIO prototypes suffer from the following limitations and require more work in the future to enhance AVIO to address these problems.

(1) Bugs that are not exposed during the monitored run. Like many previous dynamic race and memory bug detectors, AVIO reports only those bugs that manifest in the monitored run and may miss potential bugs that do not happen during that run. It would be ideal if AVIO can *predict* non-exposed bugs like lockset algorithm. However, lockset algorithm focuses on data race and cannot effectively detect atomicity violation. It also suffers from high false positive rates, less applicability to future transactional memory programs, etc. In the future, AVIO can use static and dynamic analysis to infer potential interleavings during detection, so that it can be less sensitive to scheduling. Advanced concurrency test generation techniques can also help AVIO to counter this problem. *This dissertation will study this topic in Chapter 7.*

(2) Atomicity violations involving multiple variables. Like most previous tools, including Lockset, happens-before and SVD, AVIO focuses on single variable related bugs and cannot detect concurrency bugs that involve multiple shared variables, such as the MySQL bug3. Fortunately, real-world concurrency bug characterization experience (in Chapter 3) indicates that concurrency

bugs involving single variables are more typical and can serve as the building blocks of multi-variable ones. In order to extend AVIO to detect multiple variable atomicity violation. In many cases, we can simply compose multi-address regions from several single-address atomicity regions. We just need to extend the detection protocol to look out when serialization of one variable's accesses conflicts with that of another variable. In more complicated and challenging cases when multi-addresses are correlated by high-level semantics, like the MySQL bug3, our AI-Invariant needs to be extended to consider multiple variable access interleaving invariant. *How to systematically detect concurrency bugs involving multiple variables will be studied in next Chapter.*

(3) Training Overhead. Training will also add some overhead to the whole AVIO bug detection process. Fortunately, the training results *can* be reused. Even when the code is changed, we can still save the training effort of unchanged part and only redo the training for those change-related or not well trained parts. For each training run, the overhead is similar to that of a detection run (reported in section 4.6). The training run number depends on the sufficiency requirements. As shown in section 4.6.3, as long as the related code region is covered, usually small number of training runs (less than 100 requests in our server applications) would provide sufficient interleaving samples.

4.8 Summary

This chapter has presented an innovative, invariant-based approach called AVIO to detect atomicity violations. By automatically extracting AI invariants and detecting violations of these invariants at run time, AVIO can detect a variety of atomicity violation bugs. AVIO has been implemented in two different ways, a software-only implementation (AVIO-S) and a hardware implementation (AVIO-H). AVIO is evaluated using two real-world server applications (Apache and MySQL) with five representative *real* bugs, one extracted-version of Mozilla with one real bug in it, and several SPLASH-2 benchmarks. Experimental results show that AVIO detects more bugs with much fewer false positives than previous algorithms. Comparing the two implementations of AVIO, AVIO-H incurs very little (0.4–0.5%) overhead while AVIO-S introduces fewer false positives.

Atomicity violation bugs are one of the most common types of concurrency bugs. They will likely become more common with the emerging transactional memory programming model, because concurrent programs written with such models will suffer more from atomicity violations instead of data races (explained in Section 4.1). Therefore, atomicity violation bug detection will become increasingly urgent. In this chapter, AVIO identifies a simple AI-invariant that can reflect the essence of most atomicity violation bugs. It also provides a fundamentally different and novel idea to infer atomicity intentions and detect atomicity violation bugs. In addition, hardware AVIO-H is the *first* design to us hardware support to detect atomicity violations.

Chapter 5

Detecting Concurrency Bugs II — MUVI: Multi-Variable Concurrency Bug Detection

As discussed in Chapter 3, concurrency bugs involving multiple variables are common yet poorly studied. Most previous concurrency bug detection tools, including all race detection tools and atomicity violation bug detection tools like AVIO, only look at single-variable concurrency bugs and ignore multi-variable ones.

This chapter first discusses the program semantics underlying multi-variable concurrency bugs and then presents MUVI. MUVI automatically infers the correlation among multiple variables and uses that information to detect multi-variable concurrency bugs.

5.1 Overview

5.1.1 Motivation

Previous concurrency bug detection tools mostly focused on concurrency bugs that involve only one variable. For example, AVIO (Chapter 4) only considers the atomicity of instructions that access the same shared variable. Similarly, race detection tools focus on the synchronization among accesses to the same variable, not different variables.

However, concurrency bugs caused by unsynchronized accesses to multiple variables (short for multi-variable concurrency bugs) do exist. They actually contribute to a significant percentage of all concurrency bugs. Based on the characteristics study in Chapter 3, about one third of real-world non-deadlock concurrency bugs are multi-variable concurrency bugs.

To understand why there exist so many multi-variable concurrency bugs, we start from a real-world bug example. Figure 5.1 (a–b) shows an example from Mozilla. `cache→table` and `cache→ empty` are two related variables. The former is an array and the latter is a boolean variable indicating whether or not the former array is empty. The two functions `js_FlushPropertyCache` and `js_PropertyCacheFill` both operate on these two variables. In this example, thread 1 executes `js_FlushPropertyCache`, nullifying the whole `table` array and setting the `empty` flag to true. Unfortunately, these two actions can be interleaved by `js_PropertyCacheFill` from another thread. As a result, `empty` is false, but `table` is already

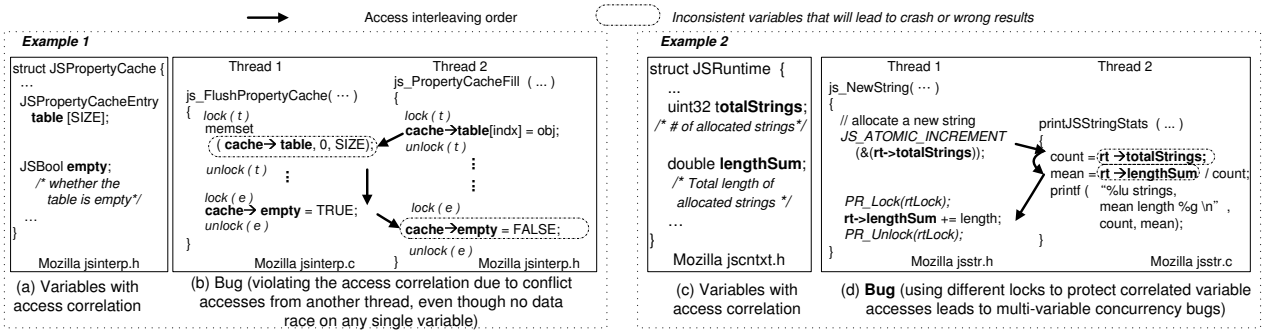


Figure 5.1: Two real examples of multi-variable access correlation and related concurrency bugs. ((a–b) shows a real bug example from Mozilla-0.8. Note that this type of bug cannot be correctly detected by data race or single-variable atomicity violation detection. Previous race detectors will either fail to report the true root cause when there is no lock or fail to report any problem if `cache→table` and `cache→empty` are separately protected as indicated. (c–d) shows a *new* concurrency bug detected by MUVI from Mozilla-0.9. Although accesses to each individual variable are well synchronized, the accesses to two correlated variables are not protected in the same critical section and cause bugs.)

cleaned. Subsequent execution will reference objects in this empty table based on the empty-flag's value (FALSE) and cause Mozilla to crash.

As we can see, the above bug is neither a race bug nor a single-variable atomicity violation bug. Even if the two accesses to `cache→table` are well synchronized and the two accesses to `cache→empty` are also synchronized, the bug still exists, as shown in Figure 5.1.

The root cause of this type of bug is that unsynchronized concurrent accesses violate the semantic relationship between variables, such as `cache→table` and `cache→empty`. Figure 5.1(c–d) shows another example of multi-variable concurrency bug. In this example, similarly concurrent execution could violate the semantic relationship between `rt→totalStrings` and `rt→lengthSum` and cause problems.

We refer to this semantic relationship between variables as **variable access correlation**. Variables with access correlations are related to each other through semantics and need to be accessed in a consistent manner. Some of them need to be updated together consistently and some of them need to be accessed together to give the program a consistent and complete view. When unsynchronized accesses violate such consistency, the program misbehaves. This is where multi-variable concurrency bugs come from.

Multi-variable concurrency bugs are important and common because the program semantic they violate (i.e., variable access correlation) is common and fundamental. Variables in our programs are not independent of each other. For example, we frequently use one variable (e.g., `empty`) to represent the constraint or status of another variable (e.g., `table`), use multiple variables (e.g., `totalStrings` and `lengthSum`) to represent different aspects of one complex

entity, and so on. These programming practices make variable correlations common in software. Consequently, unsynchronized concurrent accesses could easily violate these correlations and cause program failure.

Figuring out variable access correlation is the biggest challenge to detecting multi-variable concurrency bugs. On the one hand, without the knowledge of variable access correlation, bug detection can only blindly check the synchronization among all memory accesses, which is not only extremely slow but also unacceptably inaccurate. On the other hand, existing techniques cannot effectively extract variable access correlation. It is too tedious to have programmers write down all the correlations. Unfortunately, traditional compiler analysis cannot automatically catch such semantic information because many correlated variables are just *semantically* correlated and do not necessarily have data dependencies, such as the variable `empty` and the variable `table` shown in Figure 5.1 (a).

5.1.2 State of the Art

Multi-variable concurrency bugs have not been well studied. Most of the existing concurrency bug detection tools separately look at accesses to each single variable. For example, most existing race detection tools [DS91, SBN+97, NM91, C+02] only check the synchronization among accesses to the same shared variable. Some coarser-granularity race detection tools [vPG01, YRC05] analyze accesses to the same data structure (object), but not necessarily the same variable, together. However, they cannot solve the problem because they cannot deal with variable correlations across data structures. Furthermore, they cause many false positives because many fields from one data structure are not correlated and need not to be accessed within the same critical region [YRC05]. Atomicity violation bug detection tools [FF04] check the atomicity of certain code regions, which could include accesses to multiple variables. However, when inferring atomic code regions, existing techniques like AVIO (Chapter 4) and SVD [XBH05] still focus on the single variable. For example, AVIO's AI-invariants are associated with each single variable only.

Tools that focus on single-variable concurrency bugs cannot handle the multi-variable concurrency bug problem well. First of all, multi-variable concurrency bugs exist even if accesses to every single variable are well synchronized. Previous tools would completely miss these bugs, as shown in Figure 5.1(b). Figure 5.1(c–d) shows a new multi-variable concurrency bug detected by MUVI in Mozilla. In this example, programmers have used locks to protect all updates to `rt->totalStrings` and `rt->lengthSum`. However, since different locks are used, the correlated updates are still not protected in the same atomic region so the bug still occurs.

Second, if accesses to one of the correlated variables are not synchronized properly, previous tools may detect it but they would suggest an *inaccurate* root cause and result in two problems.

The first problem is that programmers may simply ignore the bug because the real severity of the bug is not pointed out. The second problem is that the programmers may provide an incorrect or incomplete fix, protecting each single variable separately, instead of the right fix, protecting accesses to the correlated variables *together within the same atomic region*. Such a fix will pass the checking of previous tools, but the bug still exists!

The only piece of previous work (to the best of our knowledge) [AHB03] that tries to automatically detect multiple-variable involved data race bugs uses a lock-based heuristic: if two variables, `x` and `y`, are ever accessed within one lock critical-section, they should never be separately accessed in different critical sections throughout the program. Although this work raises the issue about multi-variable concurrency bugs, its solution does not work well: all the reported bugs in their experimental results turned out to be false positives. The reason is that two variables being accessed inside one critical section once does not imply that they should always be accessed in the same critical section. Actually, they might coincidently appear in one critical section that is set up for other nearby accesses.

As we can see, with the increasing popularity of concurrent programs driven by the multi-core reality, it is important to address this fundamental limitation in concurrency bug detection.

5.1.3 Highlights

This chapter proposes an innovative and practical approach called MUVI (MUlti-Variable Inconsistency) to *automatically* identify multi-variable access correlations from programs, and detect multi-variable concurrency bugs. Apart from detecting concurrency bugs, MUVI can also detect inconsistent-update semantic bugs, which are a type of sequential bugs similarly caused by programmers forgetting to maintain variable access correlations. MUVI ideas and implementations are evaluated using several large open-source applications, including Linux, Mozilla, MySQL and PostgreSQL. Specifically, it makes the following contributions:

(1) The first tool (to the best of our knowledge) to automatically identify the commonly existent multi-variable access correlations in large programs. MUVI combines static program analysis and data-mining techniques to automatically infer multi-variable correlations. Experimental results with Linux, Mozilla, MySQL, and PostgreSQL (with 0.8–3.6 million lines of code) show that MUVI identifies a total of 6,449 multi-variable access correlations efficiently (within 19–175 minutes) with an accuracy of around 83%.

The automatically inferred variable correlations can be used to detect program bugs, such as multi-variable concurrency bugs and sequential semantic bugs, as we will demonstrate later. Apart from bug detection, they can also be used in other ways: (1) They can be stored in a specification database so that programmers can refer to it to avoid mistakes and encapsulate correlated accesses

to improve the code modularization. (2) They can be used to automatically annotate the source code and support other tools. For example, the recently proposed AutoLocker [MZGB06] can use this correlation information to assign the same lock to correlated variables. It can also help provide variable grouping information needed in Colorama [CMvPT07] and AtomicSet [VTD06].

(2) Address the fundamental multi-variable problem in previous concurrency bug detection Leveraging the automatically inferred variable access correlation, two classic race detection methods (lock-set and happens-before) are extended by MUVI to detect multi-variable related data races. How to extend other concurrency bug detectors such as AVIO (Chapter 4), RaceTrack [YRC05], and RacerX [EA03] to deal with multi-variable concurrency bugs are also discussed. Experiments with five real-world multi-variable concurrency bugs show that MUVI extensions successfully help previous tools to identify the correct root causes of four out of the five tested bugs. MUVI extensions also detect *four new multi-variable concurrency bugs* in Mozilla that have never been reported before. None of the nine bugs could be identified correctly by the original race detectors without the MUVI multi-variable extensions.

(3) The first tool to automatically detect multi-variable inconsistent update bugs Based on the inferred multi-variable correlations, MUVI also automatically analyzes the source code to detect places where programmers update one variable and forget to update its correlated variables. Experimental evaluation shows that MUVI has detected a total of 39 (22, 7, 9, and 1, respectively) new bugs from the latest version of Linux, Mozilla, MySQL and PostgreSQL, with 17 bugs already confirmed by the corresponding developers based on our bug reports. Moreover, MUVI has also detected 20 places with bad programming practices that can easily introduce bugs later. The false-positive rate of MUVI inconsistent update bug detection is reasonable (41%, on average).

The remainder of this chapter is organized as follows. Section 5.2 gives a detailed discussion about variable correlations. This is followed by the approaches to automatically infer variable correlations in Section 5.3. Section 5.4 discusses how to extend existing concurrency bug detection tools with the inferred correlations to detect multi-variable concurrency bugs. Section 5.5 shows another usage of variable correlation: detecting inconsistent-update semantic bugs. At the end of this chapter, methodology and experimental results are presented (Section 5.6 and Section 5.7).

5.2 Variable Correlations

Correlation, originating from statistics, means a pair of items' departure from independence. Many items in real-world are not independent among each other, neither are program variables in software. Variable access correlation is inherent in program semantics. The following shows the typical semantic reasons of variable access correlations:

	ID	source	Variable definitions	# of functions they are together (not)

Variables **With** access correlation	a	Linux net-device.h	struct net_device_stats {
			u64 rx_bytes; /* #of received bytes */
			u64 rx_packets; /* #of received packets*/
			}
				49 (1)

	b	PgSQL time.h	struct tm { int tm_sec; /* second */ int tm_min; /* minute */ } /* time */	25 (0)
	c	Linux fb.h	struct fb_var_screeninfo { u32 red_msb; /* red */ u32 blue_msb; /* blue */ u32 green_msb; /*green*/ u32 transp_msb; /*transparency*/ } /* for color display */	11 (1)

	d	Linux libiscsi.h	struct iscsi_session {
			spinlock_t lock; /* lock */
			int state; /* critical data */
			}
				20 (0)

	e	Linux list.h	struct hlist_node { struct hlist_node *next; /* next */ struct hlist_node **pprev; /* pevious */ } /* linked list */	32 (0)
	f	MySQL mysql-test.c	struct st_test_file* cur_file; struct st_test_file* file_stack; /* cur_file points to the top of stack */	69 (0)

Variables **WithOUT** access correlation	g	Linux net-device.h	struct net_device_stats {
			u64 rx_bytes; /* #of received bytes */
			u64 tx_aborted_erros; /* #of transfer aborts*/
			}
				4 (68)

	h	MySQL sql_class.h	Class THD {
			NET net; /* client connection descriptor */
			uint db_length; /*length of database name*/
			}
				3 (87)

Table 5.1: Six examples of variables with access correlation and two examples with no access correlation. (The last column shows the the numbers of times the variables are accessed together within the same function and separately, denoted by the parenthesis, in different functions.)

- *Constraint Specification:* A variable specifies a constraint, a property or a state of its correlated peers. For example, in Figure 5.1(a), `cache->empty` describes the state of `cache->table`.
- *Different Representation:* Two variables represent the same information in different ways. As shown in Table 5.1(a), `rx_bytes` and `rx_packets` record the incoming network traffic in different units: number of bytes and number of packets. They are accessed together [1] in 49 functions except for one, which is a new inconsistent update bug detected by MUVI and confirmed by the Linux developers.
- *Different Aspects:* The correlated variables specify different aspects of a complex data to emulate correlated real-world entities. For example, `tm_min` and `tm_sec` in Table 5.1(b) represent the minute and the second of a certain moment. They are accessed together in 25 functions and never separated. Table 5.1(c) shows four correlated fields, `red_msb`, `blue_msb`, `green _msb` and `transp_msb`, that represent the red, blue, green and transparency information for color screenplay. They are accessed together in 11 functions with only one exception, which is also a

[1] More formal discussion of *together*ness is in Section 5.3.1.

Constraint	Definition	Example
read(x) ⇒ read(y)	Every read to x semantically requires a read to y	Figure 5.1: read to `cache->table` has to be preceded by checking `cache->empty`
write(x) ⇒write(y)	Every write to x semantically requires a write to y	Table 5.1(a): update to one statistic variable (`rx_bytes`) always occur with update to its peer variable (`rx_packets`), vice versa.
write(x) ⇒AnyAcc(y)	Every write to x semantically requires an access to y	Table 5.1(d): write to `state` has to check or grab the lock `lock`
AnyAcc(x) ⇒AnyAcc(y)	Every access to x semantically requires an access to y	Table 5.1(c): accesses to `fb_var_screeninfo`'s `green, blue, read, transp` fields are always together.

Table 5.2: Examples of access constraints in correlations. (`AnyAcc` means either read or write. There are totally nine types of access constraints. Here we only show four types for demonstration. The other five types are (1) read(x) ⇒write(y), (2) read(x) ⇒AnyAcc(y), (3) write(x) ⇒read(y), (4) AnyAcc(x) ⇒read(y), and (5) AnyAcc(x) ⇒write(y).)

recently confirmed new bug detected by MUVI.

- *Implementation-demand:* The correlated variables cooperate with each other in order to implement a specific functionality of the program. Table 5.1(d) shows that the field `lock` is used to protect the critical data `state` in structure `iscsi_session`; therefore accesses to `state` are always together with accesses to `lock`. Similar examples can be seen in Table 5.1(e) (double-linked list data structure) and Table 5.1 (f) (stack data structure).

Obviously, not any two variables from a program are access-correlated. For global variables, such claim is intuitive. For multiple fields from the same structure, this also holds. The (g), (h) in Table 5.1 provide two examples: although the fields in each pair belong to the same structure, they do not have access correlation as they are accessed together in only 3–4 functions and are accessed separately in 68 or 87 functions.

Since correlated variables are connected semantically, violating an access correlation poses the risk of breaking the semantic connections and consistency, and might threaten the program correctness, as demonstrated in the bug examples shown in Section 5.1 and 5.7.

Access constraints in correlations

Access correlations do not necessarily mean that correlated variables need to be *updated* together. As shown in Table 5.2, sometimes, correlated variables are always accessed (either read or write or both) together; sometimes, reading a variable should be preceded by checking (reading) another variable; in some other cases, writing one variable requires checking (reading) *or* writing its correlated variables.

Similarly, some multi-variable access correlations are not necessarily symmetric. Modifying one may require modifying or checking the others accordingly, but the other way around is not

necessarily true. In the example (d) shown in Table 5.1, updating `state` requires accessing the lock variable, but accessing `lock` does not need to modify `state`.

Based on the above two observations, there are nine different types of access constraints for two correlated variables (four of which are illustrated in Table 5.2). For simplicity of description, for two variables x and y, we use the following notation to represent an access correlation formally:

$$A1(x) \Rightarrow A2(y)$$

where $A1$ and $A2$ can be any of the three: "read", "write" or "AnyAcc" (either read or write). For example, an access correlation, `write`(x) ⇒`read`(y), means that every time x is modified, the program needs to read the value of y together. Similarly, `AnyAcc`(x) ⇒`AnyAcc`(y) means that if x is accessed (either read or written), y needs to be accessed together. The "togetherness" notion is defined in the next section.

5.3 Variable Correlation Analysis

Variable access correlations are typically too many and tedious for programmers to specify manually. Therefore, if we can automatically infer such access correlations, we can use them as specifications, annotations and help bug detection. This section presents how MUVI automatically infers variable correlations from programs.

5.3.1 Correlation Analysis Overview

Similar to much previous work on extracting information and invariants from source code [ECH+01, KTB+06, LZ05] and dynamic execution [ECGN00, HL02], we assume that the target program is reasonably mature, i.e., it is not at its initial development stage. Almost all open source and commercial software meet this requirement, so it does not significantly limit the applicability of our work.

Since our goal is to extract multi-variable *access* correlations, not arbitrary correlations, we base our correlation analysis on variable access patterns and examine what variables are usually read or written together. For example, if every time when variable x is updated, variable y is also read together, it is very likely that some underlying program semantic (access-correlation) connects them together. In this case, we claim that `write`(x) ⇒`read`(y).

"Access Together" Definition: A non-trivial question that immediately emerges is how we claim that two accesses are "together". There could be many possible measures. For example, we can use source code distance (measured in terms of lines of code) or dynamic execution distance (measured

in the dynamic instruction trace) as a metric. Although no single measure is absolutely the best in all cases, in our scenario, dynamic execution distance is obviously not a good measure. The reason is that two correlated accesses can easily be separated by a loop or a function invocation, and thus have a large dynamic distance. In contrast, static code distance does not suffer from this limitation. It is also more aligned with programmers' coding process, which is usually centered around semantic correlations and functionalities.

Obviously, simply counting the absolute source code line gap between accesses is not enough, because that naive counting will consider two accesses in two adjacent functions as "together", which is unreasonable. Therefore, a good static distance based measure should also consider code structures such as basic blocks, functions, and files. Comparing all these units, a basic block is too small, while a file is too large. A function is the right unit since, from the programmers' point of view, a function is usually the basic unit to perform a certain task, which fits the correlation semantic.

Based on all these considerations, MUVI defines "access together" as: if two accesses (reads or writes) appear in the same function with less than *MaxDistance* statements apart, these two accesses are considered *together*, where *MaxDistance* is an adjustable threshold.

Our current prototype uses a cutoff threshold to determine whether two accesses are *together*. It is also conceivable to use a scalar metric, ranging from 0 to 1, to measure the "togetherness". Such scalar metric can also be used for ranking and false positive pruning.

"Access Correlation" Definition: Now we can formally define access correlation: x has access correlation with y (i.e., A1(x) $\Rightarrow$A2(y)), iff A1(x) and A2(y) appear together at least *MinSupport* times and whenever A1(x) appears, A2(y) appears together with at least *MinConfidence* probability, where *MinSupport* and *MinConfidence* are tunable parameters, A1 and A2 can be, respectively, any of the three: read, write or AnyAcc, and together-ness is defined as above.

Correlation Inference Steps: MUVI's correlation analysis is then to find out all A1(x) $\Rightarrow$A2(y) from the target program. It is conducted in three steps:

(1) *Access Information Collection:* MUVI parses the source code and collects each function's variable access information, including the set of variables accessed within each function, the access types and locations in source code. The information is stored in an `Acc_Set` database.

(2) *Access Pattern Analysis:* MUVI uses a frequent pattern mining technique to process the `Acc_Set` database and finds out all the variable sets that frequently appear together. This step produces a pool of variable access correlation candidates.

(3) *Correlation generation, pruning and ranking:* The last step starts from the correlation candidates produced at step 2. It uses the detailed information stored in `Acc_Set` to generate, prune and rank different types of correlations.

5.3.2 Access Information Collection

MUVI conducts flow-insensitive and inter-procedural static analysis to collect variable access information from every functions. In other words, the goal of this step is to compute the `Acc_Set` for each function in the program. To achieve this goal, MUVI needs to address several issues:

(1) which variables are we interested in? Theoretically, all variables could be involved in some correlation relationships. However, correlations involving structure/class fields and global variables are usually more common and more important than those short-lived correlations involving only scalar local variables. Therefore, MUVI considers two types of variables: global variables and structure fields (regardless of which object instance the field is associated with), represented by structure/class-name::field-name (e.g. `JSPropertyCache::empty`). These structures can be globally defined, locally defined or dynamically allocated. For simplicity of description, we refer to both global variables and structure fields as "variables" in the remainder of the chapter.

(2) what detailed access information do we need? For each variable access, we need the following detailed information to conduct further analysis: access type (read or write) for classifying different types of correlations; source code position (file name and line number) for measuring the "togetherness" of two accesses; whether an access is from a function itself or its callee functions for pruning purposes.

(3) how to handle function calls? A function can access a variable directly (referred to as a *direct access*) or via its callees (referred to as an *indirect access*). The `Acc_Set` of a function should include both direct and indirect accesses. Otherwise, some access correlations would be missed, especially in cases when a variable is read or written inside some utility functions, such as `get()` or `put()`, for the purpose of encapsulation. To achieve this, MUVI first builds a call graph of the target program and then traverses the call graph bottom up starting from the leaf nodes. All direct accesses made in a function are added to this function's `Acc_Set`. If function F calls function $f1$, $f1$'s direct accesses are also added to F's `Acc_Set` (as shown in Figure 5.2), but we do not propagate $f1$'s accesses any further along the call chain (i.e., not to F's callers). The rationale is that, if two accesses are several functions apart in the call chain, they are unlikely to be correlated (otherwise it is difficult for programmers to maintain the code). If we propagate $f1$'s direct accesses too many levels upward, we can easily introduce many false correlations. In future work, this scheme can become more flexible: allow propagations across many levels of function calls, but give more weights to more direct accesses and less weights to less direct ones.

For a direct access a in a function $f1$, its source code position is the line number where this access is made. However, when this access is propagated to $f1$'s caller F, the access's source code position in F's `Acc_Set` is the source line where F calls $f1$ so that F's direct accesses are still relatively close to this access a.

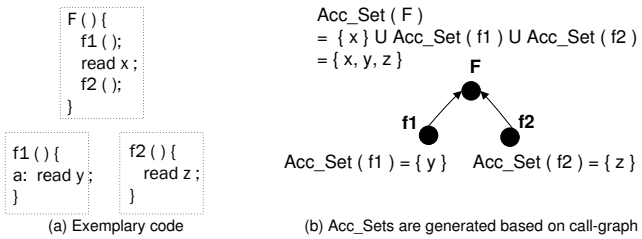


Figure 5.2: `Acc_Set` collection for an example call graph.

Issues and Extensions The above algorithm used in our current access information collection is context-insensitive, i.e., it does not consider the caller's effect during the analysis of `Acc_Set`. To be context-sensitive, we will need a parameterized `Acc_Set` summary for each function, so that different call sites of a function will get different `Acc_Set` instantiations.

Since MUVI relies on source code parsing, we cannot get information regarding accesses made inside a library whose source code is unavailable. For those common and important library calls (e.g., strlen, strcpy, memset), we can manually specify their `Acc_Sets` (i.e., the access type of a library call to its parameters). We can also extend our analysis to extract correlations from binary code, which remains as future work.

Since MUVI analyzes access correlations for fields in structures (regardless the object instance), pointer aliasing of structure objects is not an issue. For global variables, pointer aliasing could affect the accuracy, but the empirical results suggest that its effect is insignificant.

5.3.3 Access Pattern Analysis

The goal of this step is to identify variables that are accessed in the same function (i.e., appear in the same `Acc_Set`) for more than a threshold number of times. Each set of variables that satisfies this property is referred to as an access pattern. Note that a pattern is not an access correlation. Instead, it may imply a set of candidate access correlations of different types, such as `read`(x) $\Rightarrow$`read`(y), `read`(x) $\Rightarrow$`write`(y), etc.

Given the `Acc_Set` of each function, different approaches can be used to extract such access patterns. One solution is to count the number of `Acc_Sets` containing both x and y for every pair of variables (x, y). Although this solution is relatively simple, it cannot scale to large programs with millions of lines of code, such as Linux. Moreover, it would be hard to extend this algorithm to consider access correlations that involve more than two variables such as `write`(x) & `write`(y) $\Rightarrow$`read`(z).

Since access correlations involving more than two variables do exist in real-world programs (e.g., Table 5.1(c) and more examples in Section 5.7.4), we do not use the above method. Instead,

we leverage a well-studied data mining technique: frequent itemset mining [GZ03]. Frequent itemset mining examines a database where each entry is an itemset (i.e., a set of items) and tries to *efficiently* discover which sub-itemsets (subsets of an itemset) are *frequent*, i.e. contained in more than a threshold (called *MinSupport*) number of database entries. For example, in an itemset database $\mathbb{D}$,

$$\mathbb{D} = \{\{w, y, z\}, \{v, w, y, z\}, \{w, x, y\}\},$$

if *MinSupport*=3, the mining result will show that itemsets $\{w\}$, $\{y\}$, $\{w, y\}$ are frequent. If *MinSupport*=2, itemsets $\{w\}$, $\{y\}$, $\{z\}$, $\{w, y\}$, $\{w, z\}$, $\{y, z\}$, $\{w, y, z\}$ are frequent.

The specific frequent itemset mining algorithm used in MUVI is called *FPclose*. It is one of the most efficient frequent itemset mining algorithms. The details of FPclose algorithm can be found in [GZ03].

We apply FPclose to our `Acc_Set` database, consisting of the `Acc_Sets` of all functions from the target program. The FPclose algorithm outputs the frequent sub-itemsets, i.e., access patterns—sets of variables that are accessed (regardless of their access types) in more than *MinSupport* number of functions. For example, at the threshold *MinSupport* = 10, the non-correlated variable examples shown in Table 5.1(g)(h) in the previous section will not be selected as candidates, since they are accessed in only a few functions together.

5.3.4 Correlation Generation and Pruning

In this final step, MUVI takes the access patterns to generate correlations, prune false positives and rank the results. Basically, given a pattern such as (x, y), it may indicate a total of 18 correlations in the form A1(x) ⇒A2(y) or A1(y) ⇒A2(x), where A1 and A2 can be read, write, or AnyAcc. For each of the above possibilities, MUVI determines whether the access correlation holds by mainly considering two basic metrics, *support* and *confidence*, plus some other considerations.

- *Support*. Given a correlation C: A1(x) ⇒ A2(y), its support, denoted as *support*(C), is the number of functions in which A1(x) and A2(x) are together (based on the definition of togetherness in Section 5.3.1). Such a function is called a *supporter* of this correlation. If a correlation candidate has fewer than *MinSupport* number of supporters, it is pruned out.
- *Confidence*. The confidence of a correlation C: A1(x) ⇒ A2(y) measures the conditional probability that, given A1(x)'s presence in a function, A2(y) is performed nearby in the same function. It is calculated via *support*(C) / *support*(A1(x)), where *support*(A1(x)) is the number of functions that perform A1(x). Obviously, even if a correlation candidate has many supporters, a low confidence would make the correlation not trustworthy. Therefore, MUVI uses a threshold *MinConfidence* to prune out correlation candidates with too low confidence.

- *Other considerations*. In addition to the above two metrics, we also differentiate direct function supporters from indirect function supporters to improve the accuracy of correlation analysis. The former directly access variables involved in the correlation, while the latter access the involved variables via their callee functions. Clearly, direct supporters carry more weight than indirect supporters. Due to this concern, MUVI counts the number of direct supporters and prunes out correlation candidates with lower than a threshold *MinDirectSupport* number of direct supporters. It is also conceivable to give different weights to direct and indirect supporters when counting the support and confidence. Furthermore, we also prune out false correlations caused by popular variables, such as `stdout` and `stderr`. These variables are accessed in many functions, and therefore can easily be falsely inferred as correlating to many other variables. To address this problem, we prune out correlations that involve variables that appear in more than a threshold number of functions.

Ranking Large software will have a long list of correlations. In order to help programmers prioritize their efforts, MUVI ranks the correlation results based on *support* and *confidence*. Specifically, when the *support* is large enough, indicating that its *confidence* is statistically reliable, we rank the correlations based on their *confidence*; if the *support* is not large enough, we rank the correlations based on *support*.

Parameter setting Setting above threshold parameters needs to consider the tradeoffs between false positives and false negatives. Our default parameter setting (section 5.6) should provide a good initial balance point for most applications as shown in our experiments on four applications (Section 5.7). Users can start with the default setting and tune it based on their empirical experience. If users have concerns with the false positive number, they can increase the threshold parameters, such as *MinConfidence* and *MinSupport*. If users can tolerate more false positives, they can decrease the parameters to get more inference results.

Parameter setting also depends on the targeting program properties and how users plan to use the inferred correlation. For example, small applications can use relatively small *MinSupport*. If we want to use the correlations as strict requirements and report all violations to them as bugs, we would better be conservative and pick large parameters. If we only use the correlations as hints to help with bug detection, such as in the case of the multi-variable concurrency bug detection in Section 5.4, we can be more aggressive and choose relatively small parameter values.

5.4 Multi-variable Concurrency Bug

As discussed in Section 5.1, a significant portion of concurrency bugs are caused by concurrent execution inconsistently accessing multiple correlated variables. This section discusses how to

leverage the automatically inferred variable access correlations to extend existing concurrency bug detection techniques and detect multi-variable concurrency bugs.

5.4.1 Extending Race Detectors

Multi-variable concurrency bugs cannot be solved by single variable race detection. As shown in the real bug examples shown in Figure 5.1 and later in Figure 5.5, multi-variable concurrency bugs could occur when there is no race on each single variable.

The basic idea of multi-variable extension to previous race detectors is as follows. For any pair of conflicting accesses to the same variable, in addition to examining their locksets or their order, we also examine if either access has correlated accesses, and whether their correlated accesses share the same lock or are also ordered with respect to the conflicting accesses.

Background of race detectors As discussed in Section 2.2, race detectors are the most common types of concurrency bug detectors. In the following, we briefly review the two classic race detection algorithms: the lock-set algorithm [DS91, EA03, SBN+97, YRC05] and the happens-before algorithm [DS90, NM91, PK96]. We will demonstrate how to extend them in the next subsection.

The lock-set algorithm reports a data race bug when it finds that there is no common lock held during accesses to a shared memory location. To perform such a check, the algorithm maintains the set of locks currently held by each thread (called the thread *Lock Set*), and the set of locks that have been used to protect each variable so far (called the *Candidate Set*). A candidate set is initialized with all possible locks, and updated upon every access to the corresponding variable by intersecting with the thread lock set. A data race is detected when the candidate set is empty. In our study, we extend an existing dynamic implementation (by open source developers) of this algorithm in Valgrind [NS07].

The happens-before algorithm [DS90] detects data-race bugs by comparing the logic timestamps of accesses from different threads to the same shared variable. If the timestamps do not indicate a happens-before order among these accesses, a race is reported. Here the logic timestamp is calculated based on thread interaction and synchronization. In our study, the basic happens-before algorithm is implemented using a binary instrumentation tool, PIN [LCM+05].

Multi-Variable Extensions to Lock-Set Following the above basic idea, the multi-variable extension to the lock-set algorithm, referred to as lock-set$_{\text{MV}}$, is as follows. For every pair of accesses, A1(x) and A2(x) (one is a write), from different threads to the same shared variable x, we check if they are protected by at least one common lock (the basic lock-set algorithm), *and also* check if

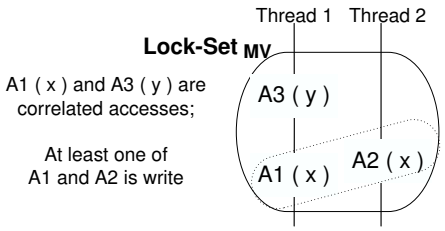


Figure 5.3: Multi-variable extension to the lock-set algorithm.

any correlated access A3(y), of A1(x) or A2(x), is also protected by a common lock with A1(x) *and* A2(x), as shown in Figure 5.3.

Note that we require only *correlated accesses*, instead of all accesses, to x and y to be protected by a common lock with their conflicting accesses. For example, if x and y only have a write-write correlation, then only those write accesses to x and y that appear together are checked.

An implementation challenge is that the access correlations are inferred from source code in the format of (global) variable names and structure/class field names, while the Valgrind lockset implementation works at binary code level. Translation is therefore required to bridge this gap. For global variables, we get their memory addresses from the compiler and feed them to lock-set$_{\text{MV}}$ prior to the detection run. For shared structure/class-objects, their dynamic allocation and deallocation raise extra challenges. In order to dynamically update the mapping from structure/class fields to memory locations for lock-set$_{\text{MV}}$, we use source-to-source code translation to wrap `malloc`-like and `free`-like functions with structure type information. Lock-set$_{\text{MV}}$ dynamically intercepts these wrappers to get the correlation information.

Multi-Variable Extensions to Happens-before Extending the happens-before algorithm to consider variable correlations is quite similar with our extension to the lock-set algorithm. The logic timestamp calculation for each memory access is the same as that in the traditional happens-before algorithm. The difference is that happens-before$_{\text{MV}}$ will compare the logic timestamp between not only accesses to the same memory location but also correlated accesses. If the timestamp comparison shows no happens-before relation with the above accesses, meaning that they can happen in arbitrary orders, happens-before$_{\text{MV}}$ reports a bug.

Extending other race detection tools Extending other dynamic race detectors, in particular those hybrid ones such as RaceTrack [YRC05], is straightforward, since they combine the lock-set and happens-before algorithms. Extending static race checkers such as RacerX [EA03] and Chord [NAW06] is even easier, since MUVI's correlation information is collected from source code, and thus is much easier to feed to static checkers than to dynamic ones.

5.4.2 Extending Atomicity Violation Detection

Even though we only implemented and evaluated the multi-variable extensions to lock-set and happens-before race detectors, it is straightforward to follow the same idea to extend other types of concurrency bug detectors, e.g., atomicity violation bug detectors.

As discussed in Chapter 4, atomicity violation bugs are important concurrency bugs. An atomicity violation bug occurs when certain code region's programmer-intended atomicity, also called serializability [2], is not maintained during execution.

An important and challenging problem in atomicity violation detection is to infer which code regions are intended to be atomic. Existing solutions rely on either manual annotation [FF04, MZGB06, VTD06] or inference based on *single*-variable-centric access patterns [LTQZ06, XBH05].

The access correlations extracted by MUVI can serve for atomicity violation detection well. An access correlation essentially indicates that the correlated accesses need to be done atomically. Taking the concurrency bug in Figure 5.1 as an example, correlation `write(cache->empty)` ⇒`write(cache ->table)` indicates that the writes to the two variables need to be atomic. The violation to this atomicity is exactly the root cause for that real bug from Mozilla.

The multi-variable atomicity information above can well complement existing tools like AVIO (Chapter 4) and SVD [XBH05]. These tools infer atomic regions based on 'code unit' composed of two consecutive accesses from one thread to *one* shared variable or read-write data dependency, respectively. Both of them would miss above atomic region composed of accesses to `cache->empty` and `cache->table`, where two different variables with no data-dependency are involved. In the following, we demonstrate how to extend AVIO to take advantage of the MUVI access correlation information.

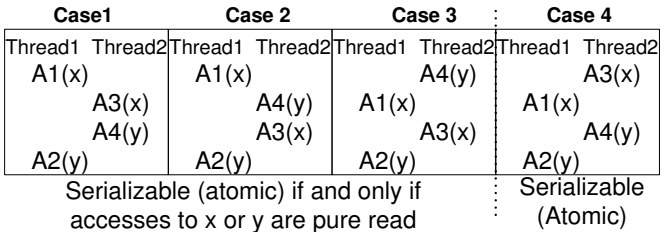


Figure 5.4: Serializability analysis based on the four cases of access interleaving on two variables. (There are totally 64 combinations of two threads accessing two variables, x and y. They can be summarized into 4 cases as above, where A1, A2, A3 and A4 represent write or read accesses. Case 4 is always atomic; Case 1–3 are atomic when there is no write access to x or there is no write access to y.)

[2] A property for several concurrently executed actions, when their data manipulation effect is equivalent to that of a serial execution of them.

The first step of our extension is straightforward. AVIO and its access-interleaving invariant focuses on the atomicity of code units composed of consecutive accesses to the same variable; MUVI can extend this to include correlated accesses to different variables. Afterward, the challenge is to decide the atomicity (serializability) of concurrent accesses to two variables. In Section 4.2.2, AVIO did such a serializability analysis by analyzing 8 different cases of single variable access interleaving. The task here is much more complicated, because two variables create many more possible access interleaving combinations. Based on the analysis of totally 64 different combinations of two-variable access interleaving, the atomicity condition is summarized in Figure 5.4. Future work can uses this atomicity condition to check whether a code unit is executed atomically or not.

5.5 Inconsistent Update Bug

Violating access correlations can lead to not only concurrency bugs, but also sequential bugs. This section presents how MUVI detects inconsistent-update bugs.

What is an inconsistent update? Inconsistent update bugs are caused by violations to write $\Rightarrow$AnyAcc access correlations. That is, sometimes, the programmer updates one variable, but forgets to update or check its correlated variable. As a result, the memory states of the correlated variables become inconsistent. Such mistakes can be easily made by programmers due to careless programming or miscommunication (as demonstrated by many bugs detected by MUVI in the latest version of Linux, Mozilla, etc). We do not consider violations to access correlations that start with a read access, because read does not directly change memory state and usually does not cause severe damages by itself.

How to detect? Based on MUVI's correlation analysis results, the basic algorithm of detecting inconsistent updates is now straightforward. For any write(x)$\Rightarrow$AnyAcc(y) correlation, we examine the violations to it. All the functions that only update x without accessing y are treated as inconsistent update bug candidates.

Ranking and pruning We first prune out likely false bug candidates based on caller–callee consideration. Given a bug candidate function F, which misses the access to y, if y is accessed in F's caller or callee functions, it is unlikely to be a bug. In our current prototype, MUVI checks two-level caller and callee functions. Of course, we can examine more levels, but our empirical results with several large software find it unnecessary.

After pruning, we rank the remaining bug candidates based on the following considerations:

(1) Violations to the strong write $\Rightarrow$write correlations are ranked at the top. Intuitively, write $\Rightarrow$write provides the most rigorous consistency requirements. If an update to y is "required" after

each update to x, the violation, which neither updates nor checks y, is most likely to cause memory state inconsistency.

(2) The more violations a correlation has, the lower rank it gets. Similar to previous rule inference work [KTB+06, LZ05], if there are too many violations to a correlation, it is unlikely for those violations to be all bugs. The rationale is that in mature software, programmers are less likely to introduce many bugs with the exact same root cause.

(3) The more trustworthy a correlation is, the more likely a violation to it is a bug. After the previous two steps, we rank the remaining violations based on the ranks of the corresponding correlations.

As an example, Linux function `velocity_receive _frame` violates the access correlation described in Table 5.1(a), write (`net_device_stats::rx_packets`) ⇒write (`net_device_stats ::rx_bytes`). Since it is the only violation to a highly ranked write⇒write correlation, it is ranked number 1 in our bug report, and it has been confirmed as a true bug by the Linux developers.

Discussion Of course, MUVI inconsistent update bug detection cannot solve all the multi-variable inconsistency problem. Since MUVI only considers access types (read or write) and not specific variable values, both false positives and false negatives could occur due to special variable values.

It is possible that when `x` is assigned a certain value, the update to `y` is unnecessary. For example, in MySQL, `SHOW_VAR::type` describes the type of data stored in `SHOW_VAR ::value`. Usually, they are updated together. However, when `type` is assigned to be `UNDEF`, there is no need to update or check `value`. It is also possible that although two correlated variables are updated together, the values assigned to them are inconsistent. Both are out of the scope of our approaches and can potentially be solved by combining value invariant techniques such as DIDUCE [HL02] and DAIKON [ECGN00] with our variable correlation analysis.

5.6 Methodology

Evaluated Applications We have evaluated MUVI correlation analysis and inconsistent update bug detection using the **latest** versions of four applications: Linux (drivers), Mozilla , MySQL, and PostgreSQL as listed in Table 5.3.

In order to evaluate MUVI's multi-variable concurrency bug detection capability, we use five known real-world [3] bugs from Mozilla and MySQL, as shown in Table 5.4.

Platform All of our experiments are conducted on a machine with a 2.4GHz Pentium processor,

[3] Since the race detectors are dynamic, we need to reproduce the bug during the execution to examine whether the detectors can catch them. Interestingly, MUVI also found *four* **new** *multi-variable concurrency bugs* never reported before.

Application	Version	LOC	Description
Linux (drivers)	2.6.20	3.6M	Operating System
Mozilla-Firefox	2.0.0.1	3.4M	Web browser
MySQL	5.2.0	1.9M	Database Server
PostgreSQL	8.2.3	832K	Database Server

Table 5.3: Applications (latest versions) used in MUVI correlation analysis and inconsistency bug detection.

BugId	App.	Description
Moz-js1	Mozilla-suite v0.9	Wrong-ordered concurrent updates make empty table's *empty* flag false; leads to system crash (Figure 5.1)
Moz-js2	Mozilla-suite v0.8	Wrong-ordered read/write to `gcPoke` flag leads to reading wrong `liveAtoms`; garbage collection fails
Moz-imap	Mozilla-Thunderbird v1.7	Wrong-ordered concurrent updates make URL-in-progress flag true, but URL string is NULL; system crashes
MySQL-log	MySQL-v4.0.16	Un-atomic read to log-file's name and log-file are interleaved by remote thread switching log files; file logging fails
MySQL-blog	MySQL-v3.23.56	Un-atomic table deletion and logging are interleaved by remote thread's table insertion and logging; security problem (Figure 5.6)

Table 5.4: Multi-variable concurrency bugs evaluated with lock-set$_{MV}$ and happens-before$_{MV}$. (It does not include the four *new* multi-variable concurrency bugs detected by MUVI. One of the four new bugs is shown in Figure 5.1(d); another will be shown in the next section.)

512 KB L2 cache, 1GB of memory, running Linux 2.4.20 as the OS. We extend the EDG [Gro] compiler front-end for static code analysis.

Parameter setting and sensitivity analysis The default parameters in MUVI variable access correlation analysis are set as follows: *MinSupport* is 10, *MinDirectSupport* is 5, *MinConfidence* is 0.8, and *MaxDistance* is 10 lines of code. We choose these values based on our sensitivity analysis. In the next section, we show the parameter sensitivity study for two critical parameters, *MinSupport* and *MinConfidence*. We fix all other parameters using the default settings and change the targeting parameter (*MinSupport* or *MinConfidence*) to measure the accuracy of inferred access correlations. We will also conduct experiments to discuss how the parameter setting would affect the bug detection results and discuss the effect of our function call handling.

Accuracy measurement In our evaluation, we *separately* measure the accuracy (false positives and false negatives) of variable access correlation analysis, multi-variable concurrency bug detection, and inconsistent update bug detection.

5.7 Experimental Results

5.7.1 Variable Access Correlation Analysis

Table 5.5 shows the variable access correlation analysis results. As we can see, variable access correlations are very common in real applications: totally 6449 access correlations (include only `AnyAcc⇒AnyAcc`) are inferred, with 5954 variables and 1467 structures involved. The analysis is efficient. For 3–4 million lines of code as Linux kernel has, it takes MUVI only 3 hours to infer 3353 access correlations.

To evaluate the accuracy of the correlations inferred by MUVI, it is too time-consuming to examine all 6449 correlations. Therefore, we take an approach similar to previous work [LLMZ04] by randomly sampling 100 correlations from each application and manually verifying whether they are true. The results show that the false positive rate is reasonably low, around 17% on average.

The above results indicate that MUVI can efficiently and reasonably accurately infer access correlations, which can be stored in a database as a specification for reference by programmers or leveraged by other tools such as AutoLocker [MZGB06], DAIKON [ECGN00] and Colorama [CMvPT07].

False positives of access correlation inference MUVI still has around 17% false correlations. They come from two major sources. (1) Macro and inlined functions are replicated by the compiler pre-processor and result in redundant supporters. Pruning these supporters requires special treatment of these macros and inlined functions. (2) In a few cases, variables are just coincidentally accessed together for many times, but there is no real correlation between them. This is especially true for some variables that get most of their support from read together. It is possible to prune out some of them by giving more weight to write-write patterns.

False negatives of access correlation inference Similar to previous work, MUVI is definitely not a panacea. It will miss variable access correlations in following cases: (1) true correlation with low supports. This is a common problem for almost all statistics based techniques [ECH+01,

App.	#Access-Correlations	#Involved Variables	#Involved Structures	%False Positive	Analysis Time
Linux	3353	3038	587	19%	175m2s
Mozilla	1431	1380	394	16%	157m40s
MySQL	726	703	209	13%	19m25s
PostgreSQL	939	833	277	15%	98m23s
Total	**6449**	**5954**	**1467**	**17%***	**450m30s**

Table 5.5: Variable correlations inferred by MUVI. (The correlations presented here only include `AnyAcc⇒AnyAcc` type. The other types are presented in Table 5.8. *: The false positive here means the average false positive.)

Bug	Lock-set$_{MV}$			Happens-before$_{MV}$		
	Detect Bug?	False Pos.	Overhead*	Detect Bugs?	False Pos.	Overhead*
Moz-js1	Y	1	39.9%	Y	1	21.2%
Moz-js2	Y	2	39.8%	Y	5	1.0%
Moz-imap	Y	0	13.2%	Y	0	1.0%
MySQL-log	Y	3	6.5%	Y	6	5.0%
MySQL-blog	N	0	5.9%	N	1	3.2%
Note: In addition to the above existing concurrency bugs, we detected four *new* multi-variable concurrency bugs that have never been reported before.						

Table 5.6: Multiple-variable concurrency bug detection results. ('Y' means root causes correctly identified; 'N' means the opposite. None of these five bugs' root causes can be correctly identified without MUVI extension. The numbers of false positives are the *additional* static false positives introduced by MUVI, excluding those introduced by the original race detectors. *: The overhead is the *extra* monitor-run overhead upon the original race detectors.)

LZ05]. To solve this problem, we might need to look for other sources of correlation evidence. (2) *conditional correlation*. Some access correlations might only exist within certain program state (an example, Figure 5.6, will be discussed in Section 5.7.2). Current MUVI prototype cannot distinguish program contexts and phases, and would miss such correlations.

5.7.2 Concurrency Bug Detection

Overall Table 5.6 shows the evaluation results on five real-world multi-variable concurrency bugs. Both lock-set$_{MV}$ and happens-before$_{MV}$ can correctly identify the root causes of four tested multi-variable concurrency bugs. None of these tested bugs' true root causes, i.e. multi-variable concurrency bugs, can be identified by the original lock-set or happens-before algorithms without MUVI's extensions.

Furthermore, MUVI also detects four *new* multi-variable concurrency bugs that have never been reported before. We have already shown an example in Figure 5.1(d). Figure 5.5 shows another new bug we find from Mozilla. In this bug, the function in thread 1 is invoked with lock protection, but the function in thread 2 is not. Actually, were not the correlation between `table->entryCount` and `table->removedCount`, the single-variable race between the two threads seemed acceptable. But with MUVI's multi-variable extension, the race detectors correctly identify the real problem: the consistency of the *correlated* accesses is broken by remote conflict accesses!

False positives of multi-variable concurrency bug detection MUVI extension also introduces a small number (0-6) of false positives. Since the original lock-set algorithm reports races more aggressively than happens-before, our extension introduces slightly more *additional* false positives

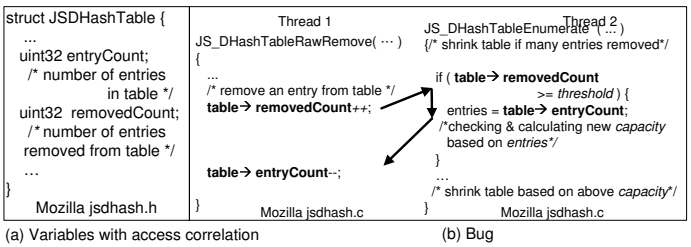


Figure 5.5: A new multi-variable concurrency bugs found by MUVI in Mozilla. (Thread 2 interleaves thread 1's update to `table->entryCount` and `table->removedCount`. As a result, thread 2 reads inconsistent values, and the `table` may not be correctly shrunk. If only considering the race on each single variable, the programmer can easily get confused. Fortunately, benefiting from MUVI's multi-variable extensions, the detectors can identify the correct root cause.)

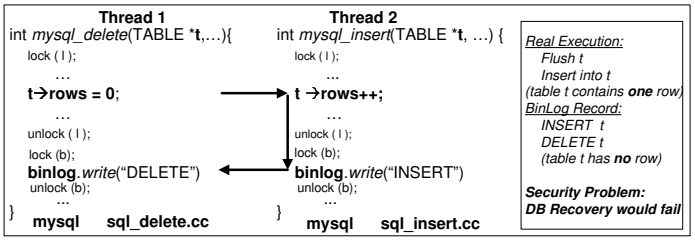


Figure 5.6: The false negative (bug MySQL-blog in Table 5.4) of Lock-set$_{MV}$ and Happens-before$_{MV}$. (The correlation between `t→rows` and `binlog` is *conditional*, and is therefore missed by MUVI. Specifically, t→rows can be accessed many times, and usually binlog need not be accessed together. Only at the end of a request processing when t→rows is given a final value, does binlog need to be consistently modified.)

to happens-before than to lock-set.

MUVI's false positives come from two sources: wrong correlations and benign multi-variable races. Benign multi-variable races exist due to special program semantics. Some of the benign multi-variable races can be pruned by sophisticated atomicity violation analysis like what we did for AVIO$_{MV}$ in section 5.4.

False negatives of multi-variable concurrency bug detection MUVI extension misses one tested bug (bug MySQL-blog), as shown in Figure 5.6. The reason of this false negative is that MUVI cannot detect the *conditional correlation*, which only holds within certain program contexts, associated with this bug. How to combine program contexts with variable correlations remains as future work.

Performance Our MUVI extension only adds a small percentage of extra overhead on the original race detectors: 5.9%-40% for lock-set; and 1%-21% for happens-before. Note that, due to the completely different implementations (one using Valgrind and the other using PIN), the extra overhead of lock-set$_{MV}$ and happens-before$_{MV}$ are incomparable. Since the original implementations already

incur too large overheads (more than 10X) to be used in production runs, our additional overheads have no major impact.

5.7.3 Inconsistent Update Bug Detection

Overall Table 5.7 shows the inconsistent update bugs detected by MUVI. Out of the MUVI inconsistent update bug reports, we manually examined the top 100 ones, and identified 39 true bugs (17 of them have been confirmed by the corresponding open source developers). All of these bugs are **new bugs in the latest versions** of Linux, MySQL, Mozilla, and PostgreSQL. Almost all of the detected bugs are semantic bugs, and therefore cannot be detected by existing tools such as memory bug detection tools. Three examples of confirmed bugs detected by MUVI are shown in Figure 5.7.

App.	#MUVI Bug Report	#New Bugs Found	#New Bugs Confirmed	#Bad program-ming	#False Positives	False pos. sources		
						S1	S2	S3
Linux	40	22	12	5	13	6	3	4
Mozilla	30	7	0	8	15	8	7	0
MySQL	20	9	5	3	8	5	2	1
PgSQL	10	1	0	4	5	5	0	0
Total	100	**39**	**17**	20	41	24	12	5

Table 5.7: Inconsistent update bugs detected by MUVI. (#New bugs confirmed means that the bugs are already confirmed by the corresponding developers after we reported these errors. "S1" stands for semantic exception, "S2" for wrong correlation, and "S3" for no future read.)

Bad programming practices Besides true bugs, there are quite a few violations that are bad programming practices that do not cause problems now but can easily introduce bugs later. For example, in PostgreSQL, pointer `PGconn::inStart` points to the starting point of a message, pointer `PGconn::inCursor` is used as a cursor pointing to a position inside a message during the message reading. In one function, the current message is discarded and therefore `PGconn::inStart` is moved to the next message, but `PGconn::inCursor` is not changed, still pointing inside the discarded message. Fortunately, in all other places in the program, the use of `inCursor` is always preceded by a checking of `inStart`, therefore such dangerous inconsistent update does not lead to a bug. However, future code revision may easily introduce a bug as a programmer may assume that these variables are always consistent. Therefore, such cases reported by MUVI can help programmers to clean up the code and improve the software quality.

False positives of inconsistent update bug detection Although MUVI has pruned some false alarms using inter-procedural analysis and confidence filtering, there are still some false positives caused by the following reasons (the breakdowns are also shown on Table 5.7):

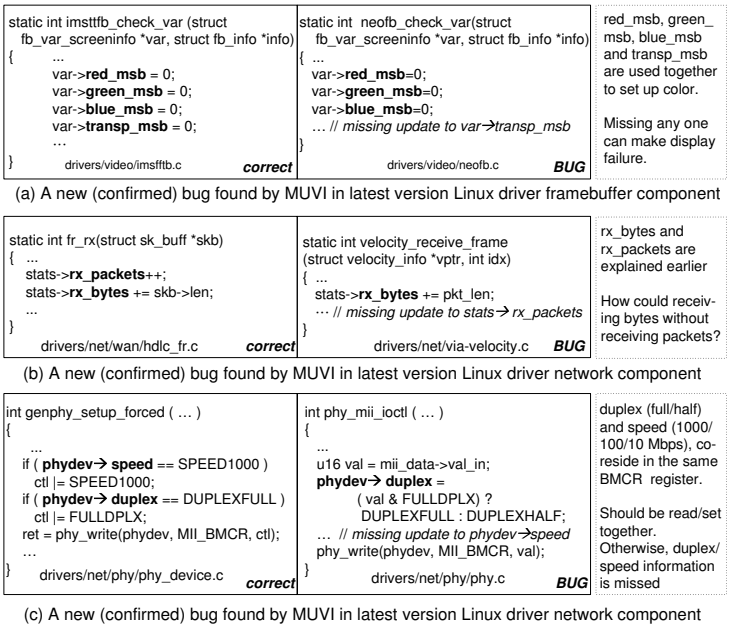


Figure 5.7: Examples of new inconsistent update bugs detected by MUVI in the latest version of Linux. (They are confirmed by the developers.)

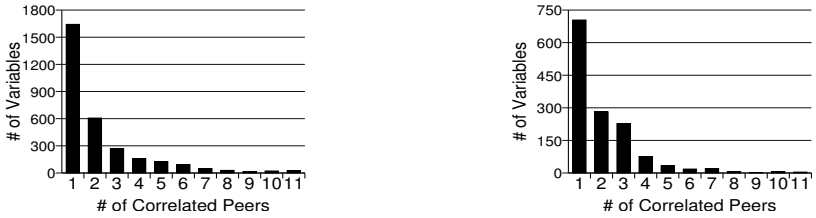
(1) Semantic exceptional cases. Even if the correlations are correct, they can still be violated in cases of special semantic requirements. For example, in Mozilla, `nsHTMLReflowMetrics::height` and `nsHTMLReflowMetrics::width` denote the height and width of an HTML object. They are always read and updated together. However, in function `AdjustForCollapsingCols`, since only col(umns) are collapsed, the program only updates width, but not height.

(2) Wrong correlations. All of the correlations inferred by MUVI are directly fed to bug detection, so wrong correlations result in around one third of the bug false positives.

(3) No future reads. It does not cause problems when a function modifies a variable without accessing the correlated peers if there is no future read to that variable. However, such an assumption about no future read needs to be carefully maintained. Therefore, such false positives can be treated as warnings as they are still helpful to programmers.

Among the above three sources of false positives, it is the easiest to solve the issue (3). False positives caused by it can be pruned by some compiler analysis, such as liveness analysis. Pruning false positives caused by issue (2) requires improving the accuracy of access correlation inference as discussed earlier. The issue (1) contributes the most to the false positives in our experiments. Solving this issue requires automatic inference of special program semantics, which is very challenging and remains as our future work.

False negatives of inconsistent update bug detection The false negatives of MUVI inconsistent



(a) Linux (b) Mozilla

Figure 5.8: Distribution of correlated variables with different number of peers in Linux and Mozilla. (The results with the other two applications are similar.)

update bug detection would come from two main sources: (1) some true correlations are missed by MUVI access correlation analysis; (2) some true bugs might be ranked low in MUVI bug reports and are therefore missed by programmers. In our current prototype, this happens when there is relatively big number of violations or small number of supports. Part of this problem can be solved by better ranking algorithms, which we will study in the future.

5.7.4 Distribution of Variable Correlations

How many correlated peers? Figure 5.8 shows the distribution of the variables with different numbers of correlated peers. The results from Linux and Mozilla show that most variables are only correlated with a small number of peers: around half of the variables are correlated with only one variable and around 20% are correlated with two variables.

This result indicates that *access correlations do not exist between any two random variables*. Even though most structures contain many fields, only those fields that have true semantic connections have access correlations.

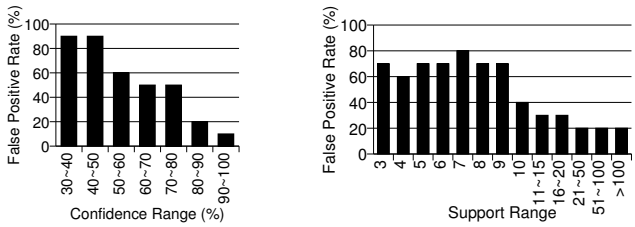
How many correlations in different type? Table 5.8 shows the number of correlations in different types. Around half of the correlations are asymmetric and therefore we need to differentiate the direction in correlation analysis. Table 5.8 also shows other types of access correlations MUVI finds. The distribution shows that most of the correlations are either read together or written together. read ⇒write and write ⇒read are relatively rare. Variables with these two types of correlations usually also support read ⇒read or write ⇒write. This indicates that the correlated variables should be either used or updated consistently.

	Sym.	Asym.	rr	rw	wr	ww
Linux	1595	1758	1141	325	408	1113
Mozilla	651	780	697	151	237	341
MySQL	316	410	339	61	81	161
PostgreSQL	586	353	365	112	131	269

Table 5.8: Directions and access types of correlations. (rr is read ⇒read, rw is read ⇒write, and so on.)

5.7.5 Sensitivity Analysis

In order to demonstrate how to choose parameters in MUVI, we perform a sensitivity study on *MinConfidence* and *MinSupport*. We only show the results of MySQL variable correlation inference in Figure 5.9. The other applications also share similar trends.



(a) Confidence (b) Support

Figure 5.9: Parameter sensitivity results for MySQL. (The false positive rates are measured by examining 10 randomly selected access correlation candidates with the specified *confidence* and *support* values.)

Figure 5.9 (a) shows how the false positive rate of MUVI variable correlation inference changes with different *MinConfidence* (all other parameters are fixed as default values). The false positive rate dramatically decreases from higher than 50% to around 20% when the confidence reaches 80%. Therefore, we choose 0.8 as the default *MinConfidence*.

Figure 5.9 (b) shows how the false positive rate of MUVI variable correlation inference changes with different *MinSupport* (all other parameters are fixed as default values). Similarly, we can see the dramatic change of false positive rate around the support range of 10, and therefore we choose 10 as the default *MinSupport*.

Parameter setting also affects the bug detection results. Here, we show the results from MUVI inconsistent update bug detection on Linux and MySQL based on different *MinDirectSupport* for demonstration. Comparing against the bugs detected by MUVI under the default setting (*MinDirectSupport*=5), Figure 5.10 shows the number of bugs detected under different *MinDirectSupport* (all other parameters use default values). More bugs would be missed with larger *MinDirectSupport*. For example, with *MinDirectSupport* 30, only 1 (out of the total 22) Linux inconsistent update bug and 3 (out of the total 9) MySQL bugs can be detected.

Apart from the parameter setting, other MUVI design also needs to consider the false positive–negative tradeoff. In our current prototype, not only direct accesses but also indirect accesses of one-level callee functions are considered in the Acc_Set (Section 5.3.2). Although this design choice results in a few more false positives, it also allows us to extract more true correlations. Take MySQL variable correlation inference as an example. Our experiment shows that MUVI can infer 51 more true correlations [4] than the alternative design where only direct accesses are considered

[4]Based on our manual examination.

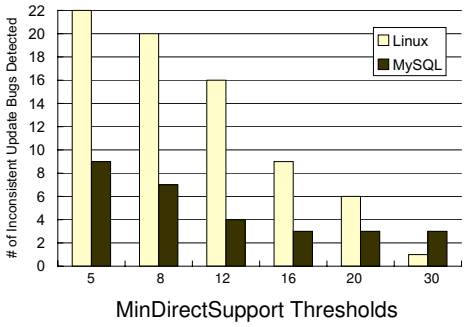


Figure 5.10: The number of Linux and MySQL inconsistent update bugs that are detected at different setting of *MinDirectSupport*. (the set of bugs detected by MUVI default setting is used as baseline)

(all other settings are exactly the same), with a reasonable number (15) of additional false positives.

5.8 Summary

This chapter proposes an innovative approach, MUVI, that combines source code analysis and data-mining techniques to automatically infer variable access correlations and detect related bugs, especially multi-variable concurrency bugs. MUVI extracts 6449 access correlations from Linux, Mozilla, MySQL and PostgreSQL with high (83%) accuracy. Based on these correlations, MUVI detects 39 new bugs (17 already confirmed) from these applications. MUVI extensions to two representative existing race detectors (lock-set and happens-before) correctly identify the root causes of four tested real-world multi-variable concurrency bugs and also detect four new multi-variable concurrency bugs that have never been reported before.

The MUVI work has gone beyond bug detection and looked at a fundamental semantic property. Results have indicated that multi-variable access correlation is a common and important program semantic property in various real-world programs and their violations can cause important concurrency and semantic bugs in operating system and server code. The access correlations inferred by MUVI can also be used to automatically annotate programs for other tools such as AutoLocker [MZGB06] and Colorama [CMvPT07].

Along the direction of concurrency bug detection, MUVI can well complement AVIO (Chapter 4) by extending it to detect multi-variable atomicity violation bugs. In terms of idea, MUVI and AVIO share a common theme: they both automatically infer underlying semantic information. Their inference approaches are different. MUVI uses a data-mining algorithm and works on source code. Instead, AVIO uses a simpler algorithm and works on execution trace. This difference is determined by their different problem natures.

MUVI is only a beginning in the multi-variable correlation research. It can be extended in four

aspects: (1) Detect other types of multi-variable related bugs, such as read inconsistency, multi-variable atomicity violation bugs, etc. (2) Improve MUVI correlation analysis and bug detection accuracy via better code analysis. (3) Extend MUVI to analyze dynamic traces to get run-time correlation. (4) Evaluate more real-world applications.

Chapter 6

Exposing Concurrency Bugs I — Interleaving Coverage Criteria Design

As discussed in Chapter 1, effectively exposing bugs before software release is critical. Without bugs being exposed, bug detection tools cannot accurately analyze the errors in the software and developers would lose the opportunity to eliminate the bugs from production software.

The following two chapters focus on the challenging problem of testing concurrent programs and exposing hidden concurrency bugs. This chapter looks at a general problem: how to explore the huge interleaving space. Following the characteristics of concurrency bugs, this chapter designs a hierarchy of coverage criteria to guide the interleaving space exploration. The next chapter looks at the concrete problem of exposing atomicity violation bugs and presents an effective testing framework: CTrigger.

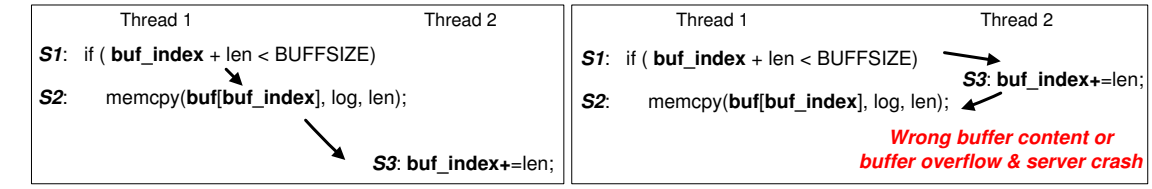
6.1 Overview

6.1.1 Motivation

Program interleaving is a unique domain of concurrent programs. An *interleaving* is an order among memory accesses from several concurrent execution components (i.e., threads or processes). Different runs of a concurrent program under one input could non-deterministically take different interleavings. All possible interleavings that an execution could take compose an *interleaving space*. Concurrent programs' interleaving space is critical and also challenging to handle.

Interleaving is important to concurrent programs because it affects the concurrent execution results. Unlike sequential programs whose execution is solely determined by inputs and environment, concurrent programs' execution is greatly affected by the interleavings. As shown in Figure 6.1, *one* program input could generate *different* results by taking different interleavings.

Interleaving is, therefore, critical to the dependability of concurrent programs. Different from that in sequential programs, bug-triggering inputs alone *cannot* guarantee the manifestation of concurrency bugs in concurrent programs. As shown in Figure 6.1, a concurrency bug might only manifest under a special interleaving. Therefore, in order to expose hidden concurrency bugs, we need to explore not only a program's input space, but also its interleaving space associated with



(a) A non-bug triggering interleaving, which almost always occurs (b) A bug triggering interleaving, which rarely occurs

Figure 6.1: An example of the interleaving effect on concurrent execution. (This code snippet is simplified from a real concurrency bug in Apache. The interleaving order among S3, S1, and S2 determines whether or not this bug will manifest.)

each input, as tried in the real world and previous research [EFN+02].

Unfortunately, exploring the interleaving space of concurrent programs is very challenging. In the real world, concurrent programs have huge interleaving spaces, factorial to the execution for each input. Facing such a huge space, real-world software development resources can afford to check only a small portion of it. How to make the best use of limited real-world resources and effectively explore the huge interleaving space has been an open problem for a long time, and has severely obstructed the improvement of concurrent program dependability.

6.1.2 State of the Art

Generally speaking, good *coverage criteria* are needed to support effective exploration of a large program space. Good coverage criteria can serve as the guidance for selecting representative states to focus on. Successful examples of coverage criteria include statement coverage [Bei90] and data flow coverage[CPRZ89, FW88], both of which are widely used to select representative inputs in software testing.

In order to effectively test concurrent programs and expose concurrency bugs, good *interleaving coverage criteria* are needed. Unfortunately, existing interleaving coverage criteria [BFM+05, FFLM96, TLK92] are quite limited. Many of them are either too complicated, with exponential complexity, or not based on solid concurrency bug models. Furthermore, in order to provide software testers with more choices and researchers with a better understanding of the design trade-offs, we need a *hierarchy* of interleaving coverage criteria covering a wide spectrum of testing complexity and bug-exposing capability, which unfortunately does not exist.

6.1.3 Highlights

This chapter presents the design of a hierarchy of interleaving coverage criteria. It will make the following two contributions:

(1) A wide-spectrum hierarchy of interleaving coverage criteria This hierarchy includes five layers and *seven* coverage criteria (*five out of the seven are newly proposed*). They are designed based on different concurrency fault models (these models are supported by previous research and the real-world bug characteristics study in Chapter 3) and represent different levels of software quality requirements. This hierarchy can provide a wide spectrum of choices and guidelines for effectively exploring the interleaving space and exposing concurrency bugs. The next chapter will further demonstrate how these criteria can guide the design of a framework for exposing atomicity violation bugs.

(2) Complexity analysis of the proposed coverage criteria The complexity analysis shows that the proposed coverage criteria range from exponential complexity to linear complexity. Some of our proposed criteria have only linear or quadratic coverage complexity and still have solid concurrency fault model basis. They have the potential to help practical concurrent program testing.

6.2 Background

6.2.1 Concepts on Coverage Criteria

A coverage criterion, in software testing, usually focuses on test case selection from a certain program testing space, e.g. the input space, the interleaving space, etc. Specifically, a criterion C includes two parts. One is a set Γ of program properties, which could be program statements, program branches, etc. The other one is a property-satisfaction function f, indicating what test cases can satisfy (exercise) a certain program property. Criterion C can measure the adequacy of a testing by checking how many program properties out of Γ are satisfied (exercised). If all the program properties are satisfied, the testing achieves *complete coverage* and is called a *complete testing* under C. Similarly, criterion C can guide the test case generation by pointing out which program properties are already satisfied and which are not yet.

Complexity, also called *cost*, and *bug-exposing capability* are the two most important metrics for a coverage criterion. The complexity of a coverage criterion can be measured by the number of test cases needed to exercise all the properties in Γ [Wey93]. The capabilities of exposing hidden program bugs could differ a lot among testings guided by different coverage criteria, because different coverage criteria focus on exercising different program properties.

It is usually difficult to reach a good balance between complexity and bug-exposing capability. That is why people used to compose hierarchical families [Bei90, FW88] of coverage criteria in order to gain a thorough understanding of the design trade-offs. In general, a good criterion should be based on a valid fault model. For example, structural coverage criteria are based on a fault model assuming most sequential bugs to be related to certain program structures and control flows.

6.2.2 A Model for Interleaving

Before presenting the interleaving coverage criteria, here we describe our model of concurrent program execution and interleavings. In the following design, we consider multi-threaded concurrent programs executed under an input upon a shared memory machine with sequential consistency memory model. Similar to previous work [Wei88], we model the concurrent execution by a sequence of shared variable access events. At any moment, only one thread is active and executes one event. When this event finishes, the same thread or some other thread will be chosen and execute its next event. Each interleaving test case is a total order among all accesses from all threads that follows the sequential consistency model.

6.3 Interleaving Coverage Criteria

This section presents five layers of coverage criteria with different sets of interleaving properties. These criteria are designed based on different concurrency bug models, starting from the most conservative bug assumption—most exhaustive criterion, and ending with the most aggressive (focused) assumption—most simplified criterion. These criteria together build a hierarchy that spans a wide spectrum. Among the seven criteria, two of them (criterion 1 and 4.A) have been proposed before, while the other five are newly proposed by us.

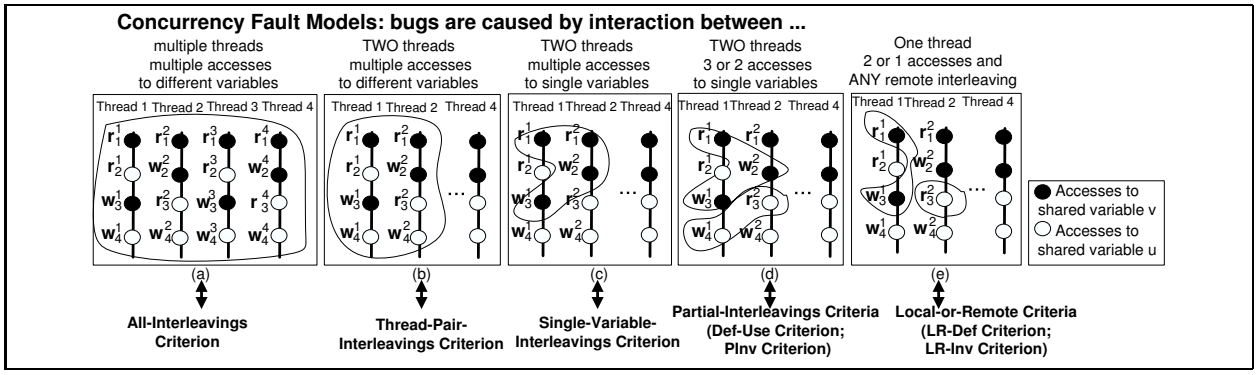


Figure 6.2: Different concurrency bug models and corresponding interleaving coverage criteria. (Note: (1) The bug models on the right are more focused and the corresponding testing complexities are smaller than those on the left. (2) The solid and hollow circles in the figure represent memory accesses to two different shared variables. (3) Different criteria focus on different interleaving properties. The curves in the figures illustrate the scopes of different properties.

• **Criterion 1: all-interleavings (ALL)** *The interleaving space gets a "complete coverage" based on ALL, iff all feasible interleavings of shared accesses from all threads are covered (Figure 6.2(a)).*

We start with a simple and exhaustive interleaving coverage criterion. It is clearly the most expensive yet also the most powerful in exposing concurrency bugs.

Property Set: *ALL* criterion treats every interleaving among all memory accesses as one property for testing. The size of the whole property set is factorial to the number of threads and exponential to the number of memory accesses within each thread. It can be calculated as following (N_i is number of access events from thread i; M is the total number of threads):

$$|\Gamma_{\text{ALL}}| = \prod_{i=1}^{M} \binom{\sum_{j=i}^{M} N_j}{N_i}$$

Example: In Figure 6.2(a), the property set of ALL includes all possible interleavings of the 16 accesses. The size of the property set is as large as 63063000 for such a simple program! Actually, the property set size for a four-thread, nine accesses per thread, program will be larger than 2^{64}!

• **Criterion 2: thread-pair-interleavings (TPAIR)** *Interleaving space gets 'complete coverage' under TPAIR, if all feasible interleavings of all shared memory accesses from any pair of threads are covered (Figure 6.2(b)).*

Starting from this model, we begin to use some concurrency bug models to gradually generalize our property set and decrease the coverage complexity.

Concurrency Bug Model: This criterion is based on the bug model that assumes most concurrency bugs to be caused by the interaction between two threads, instead of all threads. This is a widely believed concurrency bug model [SBN+97] and is also supported in our real-world characteristics study (Chapter 3).

Property Set: Each property in TPAIR is a total order of accesses from a pair of threads. To reach full coverage under TPAIR, testing needs to enumerate all possible pairs of threads and all possible total orders upon each thread pair.

The size of TPAIR property set is polynomial to the number of threads, which is a simplification from the Criterion ALL. However, it is still exponential to the number of memory accesses of each thread. The exact property set size can be calculated as following:

$$|\Gamma_{\text{TPAIR}}| = \sum_{1 \le i < j \le M} \binom{N_i + N_j}{N_i}$$

Example: Take Figure 6.2(b) as an example, one TPAIR property for thread 1 and 2 is an order among $r_1^1, r_2^1, w_3^1, w_4^1, r_1^2, w_2^2, r_3^2, w_4^2$. Since this property has no constraint on thread 3 and 4, totally 900900 ($= \binom{16}{8} \cdot \binom{8}{4}$) program interleavings of all four threads can satisfy such a property. The whole property set has size: 420.

• **Criterion 3: single-variable-interleavings (SVAR)** *The interleaving space gets a "complete coverage" under criterion SVAR, iff all feasible interleavings of all shared accesses to any specific variable from any pair of threads are covered (Figure 6.2(c)).*

Concurrency Bug Model: This criterion is based on the observation that many concurrency bugs involve conflicting accesses to *one* shared variable, instead of multiple variables. This is a widely-adopted assumption in concurrency bug detection [SBN+97][1].

Property Set: Each property in SVAR is an order among all accesses from two threads to one shared variable. The size of the whole SVAR property set is linear to the number of variables and is exponential to the number of accesses from each thread to one variable. This is simpler than TPAIR, whose property set size is exponential to the number of variables.

The exact property set size can be calculated as the follows ($N_{i,v}$ is the number of accesses from thread i to variable v; V is the set of all variables).

$$|\Gamma_{\text{SVAR}}| = \sum_{1 \le i < j \le M} \sum_{v \in V} \begin{pmatrix} N_{i,v} + N_{j,v} \\ N_{i,v} \end{pmatrix}$$

Example: As shown in Figure 6.2(c), an SVAR property is an interleaving among accesses to v or u from any pair of threads. For example, one property could be $r_1^1 \rightarrow w_3^1 \rightarrow r_1^2 \rightarrow w_2^2$ [2]. Since this property has no constraint on thread 3, 4 and u access w_4^1, it can be satisfied by totally 4504500 interleavings. The size of the whole property set is 72, much smaller than that of TPAIR.

Criterion 4: partial-interleavings (PI) To further reduce the coverage complexity, we can consider partial interleavings, i.e., execution order among a small number of accesses, based on the observation that *many concurrency bugs such as data races are caused by wrong order or wrong interaction among two or three accesses, instead of a complete interleaving among all accesses*. (This observation is supported by the characteristics study of real-world concurrency bugs presented in Chapter 3.)

This bug model is more aggressive than the previous ones. Note that is is still reasonable and has been used in previous bug detection and model checking work [MQ07, QW04]. It can have different variants. Here we discuss two variant as following:

• **Criterion 4.A: pair-interleave (PInv)** *The interleaving space gets a "complete coverage" under PInv, if for each consecutive access-pair from any thread, all feasible interleaving accesses to it have been covered (Figure 6.2(d)).*

In this criterion and the following criterion 5.B, we use *consecutive pair* or *consecutive accesses* to denote a consecutive access pair from one thread accessing the same shared variable (e.g.

[1] Of course, this model does not include those multi-variable concurrency bugs that we discussed in Chapter 5.

[2] We use $A \rightarrow B$ to represent A being executed before B.

pair (r_1^1, w_3^1) in Figure 6.2(d)); we also use *interleaving access* to denote a remote access from another thread. This access reads or writes the same variable between a consecutive pair described above (e.g. w_2^2 could become an interleaving access to (r_1^1, w_3^1) in Figure 6.2).

Concurrency Bug Model: As discussed in Chapter 4 and recent work [VTD06], unexpected interleaving accesses to a consecutive pair can indicate many concurrency bugs such as atomicity violations (an example is shown in Figure 1.1). PInv criterion is exactly based on this bug model.

Property Set: Each property is a triplet composed of two consecutive accesses (`a1` and `a2`) and an interleaving access (`a3`) from another thread. A program execution covers a property if `a3` is executed in the middle of `a1` and `a2`. The total size of the property set is no more than polynomial to the number of memory accesses in the program.

We use $PN_{i,v}$ to denote the number of consecutive access pairs in thread i to variable v. Obviously, $PN_{i,v} = N_{i,v} - 1$, if $N_{i,v} > 0$; otherwise $PN_{i,v} = 0$. The size of the PInv property set can be calculated by:

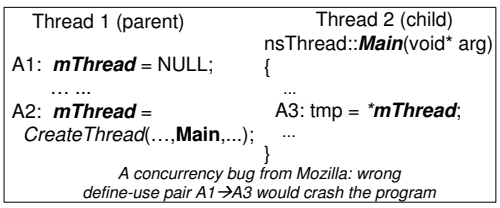
$$|\Gamma_{\text{PInv}}| = \sum_{1 \le i \ne j \le M} \sum_{v \in V} (PN_{i,v} \cdot N_{j,v}),$$

Example: In the example shown in Figure 6.2(d), a PInv property is a pair of consecutive accesses to v (or u) from one thread and an interleaving access from another thread also to v (or u). In this example, the property set has totally 48 properties, less than that of SVAR.

• **Criterion 4.B: define-use (Def-Use)** *The interleaving space gets a "complete coverage" under the Def-Use criterion, iff all possible define-use pairs are covered (Figure 6.2(d)).*

A similar criterion, a member of the data-flow coverage criteria family [FW88], is used for sequential programs for decades and was recently extended for concurrent programs [HM92, YSP98].

Concurrency Fault Model:
The underlying assumption is that, many bugs are caused by a read access using a variable defined by a wrong writer, i.e., a wrong define-use relation (an example is shown in the figure on the right).



Property Set: Each property under Define-Use criterion is simply a read-write access pair, where the read-access reads the variable defined by the write-access. These two accesses are from either the same thread or different threads. If we use N^r to denote the total number of read accesses in a program, and $N_{i,v}^r$ ($N_{i,v}^w$) for the total number of read (write) accesses in thread i to variable v, the size of the Def-Use property set can be calculated by:

$$|\Gamma_{\text{Def-Use}}| = N^r + \sum_{1 \le i \ne j \le M} \sum_{v \in V} (N_{i,v}^r \cdot N_{j,v}^w)$$

Note that, the N^r in above equation represents the cases when the read access reads a variable value defined by its own thread.

Example: As shown in Figure 6.2(d), a Def-Use property is a pair of write-read accesses to the same variable. For example, all properties related to access r_3^2 are: (w_4^1, r_3^2), $(\cdot, r_3^2)$, (w_4^3, r_3^2), (w_4^4, r_3^2), where the first element in each pair is the defining write, and $(\cdot, r_3^2)$ means that access r_3^2 reads the initial value in memory. The size of the whole property set $\Gamma_{\text{Def-Use}}$ is 32, smaller than that of Γ_{SVAR}.

Criterion 5: local-or-remote (LR)

To further reduce the coverage complexity, we can use more aggressive bug models. One possible direction is to extend criterion 4.A and 4.B to exercise *any one* remote interleaving access or *any one* remote definer, instead of *all* possible interleaving accesses or *all* possible remote definers. Following this direction, we only need to consider one property for each consecutive access pair or two properties for each reader (remote or local definer). Of course, this relaxation might lead to missing more concurrency bugs. In the following, we briefly discuss the criterion 5.A, extended from criterion 4.A, and 5.B, extended from criterion 4.B.

• **Criterion 5.A: local-or-remote interleave (LR-Inv)** *Interleaving space gets "complete coverage" under LR-Inv, if every consecutive access pair (e, e'), from any thread accessing any shared variable, has been unserializably interleaved (Figure 6.2(e)).*

Concurrency Bug Model: This criterion simplifies the criterion 4.A by relaxing the testing requirements to exercising *any one* unserializable interleaving access for every consecutive access pair.

Property Set: Each property under LR-Inv corresponds to one pair of consecutive memory accesses from the same thread to the same memory location. In order to cover the whole property set, the testing needs to unserializably interleave each consecutive access pair at least once.

The property set size is no larger than the number of all consecutive memory access pairs, which is linear to the total number of memory accesses.

• **Criterion 5.B: local-or-remote-define (LR-Def)** *Interleaving space gets "complete coverage" under LR-Def, if for each read-access r in the program, both of the following cases have been covered — r reads a variable defined by local thread (or the initial memory state) and r reads a variable defined by a different thread (Figure 6.2(e)).*

Concurrency Fault Model: The fault model is similar to the general LR criteria described above.

Property Set: The property set $\Gamma_{\text{LR-Def}}$ includes at most two properties for each read access r: r reading a value defined by a local thread's write access; and r reading value defined by a different

thread's write. The property set size ranges between N^r and $2N^r$, where N^r is the number of all read accesses.

Overall, criterion ALL subsumes criterion TPAIR, which subsumes SVAR, which subsumes PI and LR. All together, they build a hierarchy.

6.4 Cost Analysis

As discussed in section 6.2, the *testing cost* (also called complexity) of a criterion C measures how many test cases are needed to completely cover the property set defined by C. In our study, a test case is an interleaving among all memory accesses and the property-set size has been discussed in section 6.3. Based on the definition, the testing cost may not equal C's property set size because one test case may cover multiple properties. We should also note that the testing cost defined above does *not* include the effort to analyze the feasibility of each property or figure out the way to exercise each property during the testing.

For each testing coverage criterion, we will estimate both the upper-bound and the lower-bound of its testing cost, denoted by *MaxCost* and *MinCost*. Since it is both difficult and *unnecessary* to get an accurate and exact cost, we use the *Stirling Approximation* [3] and other approximations to get magnitude-level results for comparison purpose. Specifically, we assume that every thread executes the same number of memory accesses (N); each variable is accessed the same number of times in each thread ($N_V = \frac{N}{V}$); the number of read accesses from each thread equals the number of write accesses from each thread ($\frac{N}{2}$). We use V to denote the number of shared variables and M to denote the total number of threads. Finally, we consider N as a significantly large number.

ALL Criterion: Since we need exactly one test case to fulfill one property in ALL Criterion, ALL's testing cost is equal to its property set size. Stirling Approximation can give us following cost approximation:

$$\begin{aligned}\text{MaxCost(ALL)} = \text{MinCost(ALL)} = |\Gamma_{\text{ALL}}| &= \prod_{i=1}^{M} \begin{pmatrix} iN \\ N \end{pmatrix} \\ &= \frac{(MN)!}{(N!)^M} \approx (\frac{\sqrt{M}}{\sqrt{2\pi N}^{M-1}}) M^{MN}\end{aligned}$$

The cost of ALL criterion is exponential to the number of memory accesses (per thread) and factorial to the number of threads.

TPAIR Criterion: TPAIR's property set includes all interleavings among every pair of threads.

[3] $n! \approx \sqrt{2\pi} n^{n+1/2} e^{-n}$ [Mat]

One test case can cover at most $M(M-1)/2$ and at least $M-1$ unique properties in TPAIR. The rationale is as follows. If we look at each test case separately, a test case can always cover $M(M-1)/2$ TPAIR properties. However, TPAIR properties covered by different test cases inside the test case set may overlap. Fortunately, careful design can guarantee each test case to cover at least $M-1$ unique properties. This is based on the observation that, for *any* group of TPAIR properties $\{\text{interleaving}_{1,2}, \text{interleaving}_{2,3}, ..., \text{interleaving}_{M-1,M}\}$ ($\text{interleaving}_{i,j}$ denotes an execution order among thread i and j), we can always find an interleaving test case to cover all these $M-1$ pair-wise interleavings[4].

Based on the above analysis, we can use Stirling Approximation to get the following testing cost approximation:

$$\begin{aligned} \text{MinCost(TPAIR)} &\approx 2|\Gamma_{\text{TPAIR}}|/(M(M-1)) \\ &= \binom{2N}{N} \approx \frac{4^N}{\sqrt{\pi N}} \end{aligned}$$

$$\begin{aligned} \text{MaxCost(TPAIR)} &\approx |\Gamma_{\text{TPAIR}}|/(M-1) \\ &= \binom{2N}{N}\frac{M}{2} \approx \frac{4^N}{\sqrt{4\pi N}}M \end{aligned}$$

The cost of TPAIR criterion is exponential to the number of memory accesses (per thread) and no more than linear to the number of threads.

SVAR Criterion: Each SVAR property is an execution order among all accesses from two threads to one variable. When the SVAR-properties on different variables do not conflict with each other, the number of SVAR-properties an interleaving test case can cover reaches its upper limit: $VM(M-1)/2$. When the SVAR-properties on different variables conflict with each other, we conservatively estimate the lower limit to be $M-1$.

$$\begin{aligned} \text{MinCost(SVAR)} &\approx 2|\Gamma_{\text{SVAR}}|/(VM(M-1)) \\ &= \begin{pmatrix} 2N_V \\ N_V \end{pmatrix} \approx \frac{4^{N_V}}{\sqrt{\pi N_V}} \end{aligned}$$

$$\begin{aligned} \text{MaxCost(SVAR)} &\approx |\Gamma_{\text{SVAR}}|/(M-1) \\ &= \begin{pmatrix} 2N_V \\ N_V \end{pmatrix}\frac{VM}{2} \approx \frac{4^{N_V}}{\sqrt{4\pi N_V}}VM \end{aligned}$$

[4]Similar $M-1$ rationale will be used for several subsequent criteria to calculate their cost upper-bound.

The cost of SVAR criterion is exponential to the number of memory accesses (per thread per variable) and no more than linear to the number of threads.

PInv Criterion: Estimating how many test cases are needed to fully cover a PInv property set is not easy. Fortunately, the following observation can help us: each access r can never interleave more than one consecutive access pairs from one thread in one run. This observation indicates that we need at least $N_V - 1$ test cases to fully cover the PInv property set. To approximate the *MaxCost*, we simply leverage the fact that each test case can cover at least $M - 1$ unique PInv properties in a carefully designed test case set. The testing cost can be approximated as follows.

$$\text{MinCost(PInv)} \approx N_V - 1$$

$$\begin{aligned} \text{MaxCost(PInv)} &\approx |\Gamma_{\text{PInv}}|/(M-1) \\ &\approx \frac{MV{N_V}^2}{2} \end{aligned}$$

The cost of PInv criterion is at most linear to the number of threads and no more than polynomial to the number of memory accesses (per thread per variable).

Def-Use Criterion: The following observation can help approximate the testing cost of Def-Use: each read access can have only one definer in each run. Based on this, we consider each test case to cover at most $\frac{NM}{2}$, i.e., one for each memory read access, and at least $M - 1$ unique define-use pairs.

$$\begin{aligned} \text{MinCost(Def-Use)} &\approx 2|\Gamma_{\text{Def-Use}}|/(NM) \\ &= \frac{N_V(M-1)}{4} \end{aligned}$$

$$\begin{aligned} \text{MaxCost(Def-Use)} &\approx |\Gamma_{\text{Def-Use}}|/(M-1) \\ &= \frac{N}{2} + \frac{MV{N_V}^2}{8} \end{aligned}$$

The cost of Def-Use criterion is linear to the number of threads and no more than polynomial to the number of memory accesses per thread per variable.

LR Criterion: As we have seen, the property set size is linear to the total number of memory accesses (NM) for both LR-Def Criterion and LR-Inv Criterion. It is safe to approximate the upper-bounds of both criteria's testing costs as NM, the total number of memory accesses from all

threads. As for the lower-bounds, in ideal case, all LR-Def or LR-Inv properties could be covered by one interleaving test case. Therefore, the lower-bounds of both criteria's testing cost is constant.

All the testing cost analysis results are summarized in Table 6.1.

	ALL	TPAIR	SVAR
Upper-Bound	exponential to N; factorial to M	exponential to N; linear to M	exponential to N_V; linear to M
Lower-Bound	exponential to N; factorial to M	exponential to N; constant to M	exponential to N_V; constant to M

	PInv	DefUse	LR-Inv	LR-Def
Upper-Bound	polynomial to N_V; linear to M	polynomial to N_V; linear to M	linear to N; linear to M	linear to N; linear to M
Lower-Bound	linear to N_V; constant to M	linear to N_V; linear to M	constant	constant

Table 6.1: Testing costs of different coverage criteria. (This is based on the cost definition in Section 6.2. N is the number of memory accesses from each thread; N_V is the number of memory accesses from each thread to each variable; M is the number of threads.)

In summary, ALL, TPAIR and SVAR criteria all require exponential number of testing runs, too expensive to build complete test set in practice for real-world software. The other four criteria Def-Use, PInv, LR-Def and LR-Inv criteria are much better, requiring only quadratic or linear size test case set. If these four criteria can achieve reasonable bug-exposing capability, they are good choices to guide real-world concurrency testing.

6.5 Summary

This chapter has presented a hierarchy of seven interleaving coverage criteria for concurrent programs. These criteria are designed based on different concurrency fault models, which are supported by previous concurrency bug research and the characteristics study in this dissertation. Consequently, they have different complexities and bug exposing capabilities. All together, they provide a good guidance for systematic exploration of concurrent programs' interleaving space. The next chapter will use one of the proposed criteria as guideline to build a testing framework for exposing atomicity violation bugs.

Chapter 7

Exposing Concurrency Bugs II — CTrigger: A Framework to Expose Atomicity Violation Bugs

This chapter continues to work on exposing concurrency bugs. A framework called CTrigger is presented to effectively expose one important type of concurrency bugs: atomicity violation bugs.

7.1 Overview

7.1.1 Motivation

As discussed in Chapter 4, an atomicity violation bug is introduced when programmers assume the atomicity of a code region but their implementation does not use proper mechanisms such as locks or transactions to enforce it.

As also discussed in Chapter 4, atomicity violation bugs are one of the most common and important types of concurrency bugs [LTQZ06, LPSZ08, PS08, LDSC08, VTD06, FF04]. They widely exist because many programmers are used to sequential thinking and frequently assume code regions to be atomic without appropriate enforcement. Chapter 3 shows that about 70% of studied non-deadlock concurrency bugs in the examined server and desktop applications are caused by atomicity violations. In addition, atomicity violation bugs will remain even with advanced synchronization primitives such as transactional memories, because programmers might mistakenly separate a group of indivisible operations into different transactions [LTQZ06, LDSC08]. Therefore, techniques to help address atomicity violation bugs are highly desired.

Recently, much effort [XBH05, FF04, HDVT08, VTD06], including the AVIO work presented in Chapter 4, has been made to help detect atomicity violation bugs. Almost all of these works would significantly benefit from an effective way to *expose* atomicity violations during monitored runs (testing runs). For example, dynamic atomicity violation checkers like AVIO, SVD [XBH05], and other approaches [HDVT08] require atomicity violations to manifest during monitored runs in order to catch them. Although static approaches [FQ03, KRDV07] do not have such requirements, their power is limited due to the complexity of analyzing concurrency and pointer aliasing, especially for C/C++ programs. As a result, static tools can introduce many false positives. An

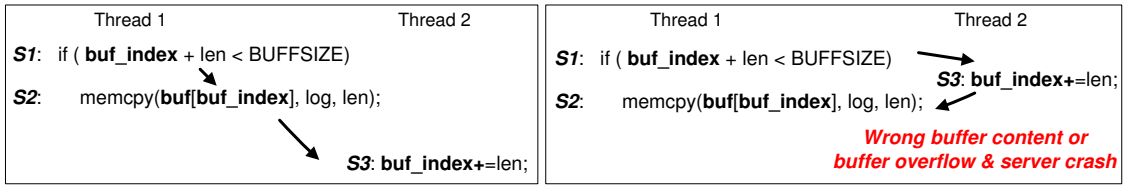
effective way to examine bug suspects via testing runs can also help static tools to separate false positives from true bugs [Sen08].

Bug-exposing techniques have been studied for a long time and many good techniques have been widely adopted to test *sequential* programs [Bei90]. In general, a good bug-exposing technique needs to have three properties:

- *Effectiveness*: how many hidden bugs can be exposed;
- *Efficiency*: how fast hidden bugs can be exposed. Although the performance issue of testing is not as critical as that of production runs, the bug-exposing process cannot take months or years because programmers are under constant pressure to release software. In particular, commercial software typically needs to run hundreds to thousands of test cases with different inputs and configurations prior to software release. Let us assume that there are 100 different input cases, 10 different OS/hardware/library configurations, and 20 machines for testing. If each test case takes 20 hours on a machine, it would require $(100 \times 10 \times 20)/\ 20 = 1,000$ hours (i.e., around 42 days) to finish these 100 test cases. This is certainly too slow for commercial companies who need to constantly roll out new changes.
- *Reproducibility*: if a hidden bug is exposed, how likely this bug can be reproduced for diagnosis. If the bug takes another 20 hours to reproduce, it might be very painstaking for programmers to examine the problem.

Unlike exposing sequential bugs, exposing a concurrency bug usually requires at least two conditions. The first condition, similar to that of sequential bugs, is a bug-triggering input. An appropriate input is needed to execute a faulty code segment with a bug-triggering state. Much work has been conducted in the past to generate comprehensive sets of inputs to cover code segments and specification space [Bei90]. The majority of these works are still applicable to concurrent programs, although some extensions specific to concurrent programs are needed to further increase the code coverage [SA06].

The second condition, unique to concurrency bugs, is a *bug-triggering interleaving*. Without this condition, the bug-triggering input alone may not expose the hidden concurrency bug. Figure 7.1 shows a real-world bug example from the Apache HTTPd Server, a widely used, open-source Web server. In this example, programmers forgot to protect the pair of accesses to `buf_index`, namely $\{S1, S2\}$, into the same atomic region using locks or transactions and introduce an atomicity violation bug. Unfortunately, this bug is hard to expose during testing because it manifests *only* when $S3$ is executed between $S1$ and $S2$. The probability for this particular interleaving to happen is very small. Actually, when we ran Apache with a *bug-triggering input* (an input that can potentially trigger the bug) on an 8-core machine, it took 22 hours for this bug to manifest.



(a) A non-bug triggering interleaving, which almost always occurs (b) A bug triggering interleaving, which rarely occurs

Figure 7.1: An example of the manifestation condition of concurrency bugs. (This example is simplified from a real Apache atomicity violation bug. It manifests only when S3 is executed between S1 and S2.)

In comparison to the first condition, the second condition is significantly understudied. Therefore, similar to recent concurrency testing efforts [MQ07, Sen08, EFN+02], this chapter relies on prior work to cover the first condition and focuses on the *bug-triggering interleaving issue*. Specifically, we will study how to systematically examine interleavings selected from the huge interleaving space and find out which ones hide atomicity violation bugs, if they exist.

7.1.2 State of the Art

The common practice to expose concurrency bugs is to run a program with each input test case for a long time (for servers) or for many times (for other types of applications). We refer to this as *stress testing*. Intuitively it makes some sense, since the non-deterministic nature of concurrent programs will help exercise different interleavings in different runs. Unfortunately, practice has shown that stress testing is neither efficient nor reproducible [MQ07]. The first part of this chapter will dig deeper into the reason behind the deficiency of stress testing, showing that different interleavings have different probabilities to be covered in a stress testing and many bug-hidden interleavings have very low probabilities to be covered.

Recently, several inspiring works [BFM+05, EFN+02, MQ07, Sen08] were proposed to improve stress testing. All these works aim to reduce the exponential size interleaving space into smaller sets of interleavings for practical testing to focus on.

ConTest [BFM+05] injects artificial delays at synchronization points (e.g., lock acquisition and lock release) in order to intensify the contention for synchronization resources. This would help expose deadlocks, but not data races or atomicity violation bugs, because the latter types of bugs are usually caused by programmers forgetting to use appropriate synchronizations to protect shared memory accesses.

CHESS [MQ07] cleverly limits the number of context switches during an execution to 1–4 and therefore significantly reduces the number of interleavings to explore. However, it has to make a difficult trade-off between coverage and testing time. Even with a couple of context switches,

there are still a huge number of possible choices, including which locations/threads to switch from and which locations/threads to switch to. The number of choices further increases polynomially if three or four switches are allowed. That is why CHESS allows context switches only at synchronization points when it was used by Microsoft developers on real-world programs in order to be practical. Such a constraint will make the method less effective for exposing atomicity violation and data race bugs, just like that in ConTest as discussed earlier. CTrigger, presented in this chapter, well complements CHESS by systematically picking out interleavings that have low occurrence probabilities and high association with atomicity violation bugs.

Based on the same motivation, RaceFuzzer [Sen08, PS08] focuses on potential data races reported by race detectors. It attempts to force all the reported race interleavings during testing in order to separate false positives from true race bugs. While this approach is definitely useful to help users automatically filter out false positives in race bug detection, its bug-exposing capability significantly relies on the underlying data race detectors: if the detector does not have a good coverage, RaceFuzzer would miss many bugs. Unfortunately, due to the inherent complexity of concurrent programs, there are still few race bug detectors that can achieve high coverage, especially for C/C++ programs and atomicity violation bugs.

In addition, both CHESS and RaceFuzzer only select one thread to execute at a time, which can significantly slow down each test run and cannot take full advantage of multi-core machines in testing. While it is possible to conduct multiple testing runs on the same machine, the contention for disk and network resources makes it impractical for I/O-intensive applications, such as server programs. In this chapter, CTrigger framework will be proposed to address this limitation and allow each test run to use multiple processors, just like that in stress testing.

7.1.3 Highlights

This chapter studies the interleaving characteristics of stress testing and proposes a practical method called CTrigger to efficiently expose atomicity violation bugs in large programs.

First, to select representative interleavings to focus on, we follow the guidance of LR-Inv interleaving coverage criteria (presented in Chapter 6). That is, based on the type of concurrency bugs we target, our bug-exposing framework focuses on a special type of interleaving (the unserializable interleaving) that is inherently correlated to atomicity violation bugs [XBH05, LTQZ06]. An *unserializable interleaving* is an interleaving that is not equivalent to any sequential execution of the involved operations (more details are presented in Section 7.2). As discussed in Chapter 6, focusing on unserializable interleavings can help expose most atomicity violation bugs and also allows a substantial reduction of the interleaving space that needs to be explored.

Second, using three large server programs, three SPLASH2 programs, and one utility program,

we examine why stress testing is insufficient in exposing atomicity violation bugs. Our evaluation shows that different unserializable interleavings have different probabilities of occurrence; different runs in stress testing usually cover similar interleavings, high-probability ones; and low-probability ones, which usually hide atomicity violation bugs, have little chance to be covered without external control, and are also hard to reproduce for bug diagnosis. The primary factors that affect interleaving probabilities are synchronizations, memory access distances, and so on.

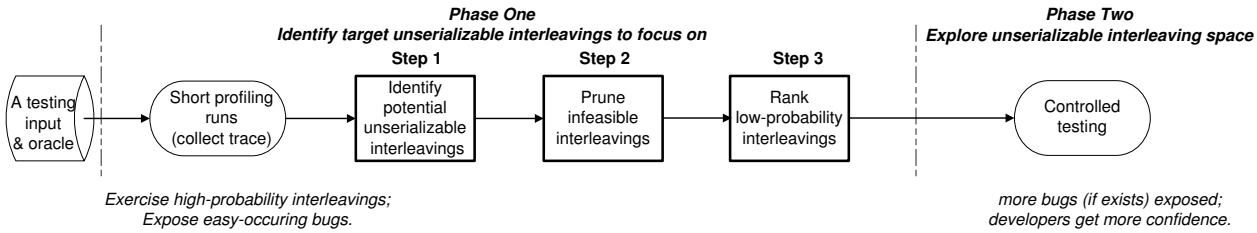


Figure 7.2: CTrigger testing framework

Third, based on these observations, a testing framework called CTrigger is designed to effectively, efficiently, and reproducibly expose atomicity violation bugs in concurrent programs. CTrigger achieves these goals by incorporating the following new ideas step by step, as shown in Figure 7.2.

- Focusing on unserializable interleavings. From a few profiling runs, CTrigger identifies a large set of potential unserializable interleavings.
- Pruning infeasible interleavings. Not every potential unserializable interleaving can happen during execution due to synchronizations. For example, two accesses protected by a lock cannot be unserializably interleaved by a remote access (access from another thread) protected by the same lock. To prune such infeasible interleavings, we design a pruning algorithm that considers two types of synchronization operations: order synchronization and mutual exclusion. By pruning out infeasible unserializable interleavings, it can significantly reduce the number of vain attempts to force those interleavings. Our experimental results show that 37%-96% of potential unserializable interleavings in the seven tested applications can be pruned.
- Ranking and identifying low-probability interleavings. As different interleavings have different probabilities to be exposed, we propose a simple metric to estimate interleaving probability and rank all unpruned unserializable interleavings. This ranking mechanism allows us to focus on low-probability interleavings during controlled testing, leaving high-probability ones to be covered by the simple stress-testing mechanism. Our experimental results show that our ranking mechanism is effective. It ranks bug-triggering interleavings high, mostly

within the top 10%, and accelerates bug exposing time by up to 457 times. Besides our work, the ranking metric may also be useful to other concurrency testing frameworks to improve testing efficiency.

- Minimum external control to force low-probability interleavings during testing on multi-cores. Unlike CHESS and RaceFuzzer, both of which control execution by scheduling one thread at a time, CTrigger inserts artificial synchronizations in only a small set of execution points corresponding to the target interleavings of interests. This allows the tested program to leverage multi-cores, and avoids slowing down execution periods that are unrelated to the target interleavings.

CTrigger is evaluated on 8-core machines with real-world buggy applications, including four server/desktop open-source programs (MySQL, Apache, Mozilla, and PBZIP2) and three SPLASH2 benchmarks. Among these applications, MySQL, Apache, and Mozilla are widely used, large open-source programs with up to 3.4 million lines of code. CTrigger exposes the tested atomicity violation bugs 10–1000 times faster than stress testing and previous methods (both synchronization-based or race-based techniques described in Section 7.1.2). For example, CTrigger takes 63 seconds and 235 seconds, respectively, to expose the two real-world Apache server bugs, whereas the stress testing requires more than 20 hours to expose them, and one of the bugs never manifests, even after *one week* of stress testing!

As explained before, testing efficiency is very important due to the time pressure to release the software as well as the substantial number of test cases. Therefore, an acceleration factor of 10–1,000 would be very beneficial. For example, if we use CTrigger to test the program for 1 hour, we would need to run stress testing or other testing mechanisms for 10–1,000 hours to achieve equivalent results. To perform the same calculation exercise as we did earlier with 100 different input cases and 10 different configurations on 20 machines, CTrigger would take 1,000/20 hours=50 hours$\approx$2.1 days to finish all test cases, whereas the stress testing would take 21–2,100 days to have similar exposing capability for atomicity violation bugs, which is definitely too long to be acceptable.

In addition, since CTrigger records the execution control that exposes a bug, it can perform the same control to reliably re-expose the same bug for diagnosis without any deterministic replay support. For the tested bugs, CTrigger re-exposes them mostly within 5 seconds, 300 to more than 60,000 times faster than stress testing.

7.2 Background: Unserializable Interleavings

Facing the huge interleaving space, the first step of concurrency testing is to decide what *type* of interleavings to focus on. In this section, we first give a brief review of atomicity violation bugs (more details are in Chapter 4) and then discuss why we focus on unserializable interleavings to expose atomicity violation bugs (you can also refer to Criteria 5.A, LR-Inv, in Chapter 6 for a deeper understanding).

Atomicity, also called as **serializability**, is a property for the concurrent execution of several operations when their data manipulation effect is equivalent to that of a serial execution of them. Programmers often assume some code regions to be atomic. Unfortunately, their implementation may not guarantee the atomicity. Consequently, the assumed atomicity can be broken when the code region is unserializably interleaved by accesses from another thread, which leads to an atomicity violation bug.

As discussed in details in Chapter 4 and some recent work [VTD06], the basic type of unserializable interleavings is composed of three memory accesses (shown in Figure 7.3). Two of them, referred to as p(receding)-access and c(urrent)-access, consecutively access a shared location from the same thread. The third one, referred to as r(emote)-access, accesses the same memory location in the middle of the previous two from a different thread. For example, the key part of the bug shown in Figure 7.1 is such a basic type of unserializable interleaving. The bug manifests when r-access S3 unserializably interleaves the p-access S1 and c-access S2.

Due to the inherent connection between atomicity violation bugs and unserializable interleavings, it is natural to focus on unserializable interleavings in order to expose atomicity violation bugs. Furthermore, for simplicity and efficiency, we can start with the basic type of unserializable interleavings described above. Specifically, for every shared memory access instruction C, we can try to exercise at least one unserializable interleaving associated with C, i.e., *interleaving-C*, short for an unserializable interleaving with instruction C as the current access. We accordingly define the exploration space as $\{\text{interleaving-}C | C \text{ is a shared-memory access instruction}\}$. Within this space, some unserializable interleavings may never happen due to synchronization. We will discuss how to prune out these *infeasible interleavings* in later sections.

The size of the unserializable interleaving space defined above is linear [1] to the static size of the program. It is much smaller than the entire interleaving space and is therefore practical to thoroughly explore. In the meantime, unserializable interleaving space gives a good coverage for all potential atomicity violation bugs. Covering this space during testing would give developers at least some level of confidence on their software quality against atomicity violations.

[1] This linear-sized space provides a good foundation for testing. Of course, figuring out and exercising all feasible unserializable interleavings inside this space would require more than linear complexity.

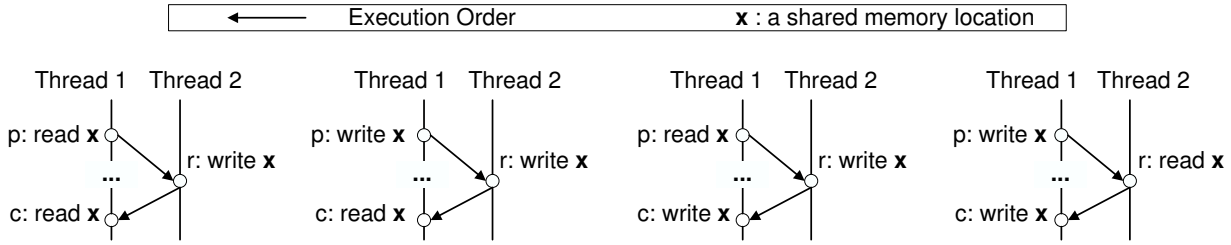


Figure 7.3: Unserializable interleavings. (A static instruction C's unserializable interleaving is exercised iff at least one of its dynamic instances follows above pattern during execution.)

7.3 Why Stress Testing is Not Good — An Interleaving Characteristic Study

Stress testing (defined in Section 7.1.2) is the current dominant practice. To understand why it is ineffective at exposing atomicity violation bugs, we quantitatively study its characteristics from the perspective of unserializable interleaving space. The understanding will guide our design of CTrigger and help find out how to enhance stress testing in order to disclose hidden atomicity violation bugs.

7.3.1 Methodology

We use four widely-used open-source server/desktop applications, Apache HTTPd, MySQL, Mozilla and PBZIP2, and three applications from the SPLASH2 [WOT+95] benchmark-suite. These applications cover different types of functionalities and synchronization models, as shown in table 7.1. The experiments use a dual quad-core (totally eight processors) Intel Xeon machine, and each application is configured to have eight worker threads.

App.	LOC	Description	Synchronization Model	Workload
Apache	302K	Web server	lock	SURGE [BC98]
MySQL	1.9M	Database server	lock	MySQL-test*
Mozilla	3.4M	Web browser suite	lock	JavaScript test suite*
PBZIP2	2.0K	Parallel BZIP2 file compressor	lock & queue	a random file
FFT	1.0K	FFT transformation	barrier	default setting with 8 processors
LU	1.0K	Matrix factorization	barrier	
Barnes	3.0K	N-body problem	lock & queue	

Table 7.1: Applications and workloads used in the interleaving characteristics study. (*:MySQL-test and JavaScript test suite are designed by MySQL and Mozilla developers.)

In order to collect the interleaving information, we use PIN binary instrumentation tool [LCM+05] to monitor the execution. To make sure that our study can reflect the real non-perturbed execution environment, we carefully design our instrumentation to give minimum perturbation in a thread-balanced way.

7.3.2 Observations

Our experimental results reveal the following observations:

(1) *Is stress testing non-deterministic in a random way?*

From the perspective of covering unserializable interleavings, the answer is *no*. As shown in Figure 7.4, the majority of unserializable interleavings exercised by different runs (or different iterations for server programs) are the same.

(2) *Can we rely on stress testing to cover the whole unserializable interleaving space?*

The answer is *no*. As shown in Figure 7.5 [2], stress testing hardly exercises any new unserializable interleaving after the first few runs and leaves some feasible unserializable interleavings uncovered in every application. Actually, some feasible interleavings are never exercised in days of stress testing. Unfortunately, these interleavings are exactly the most obnoxious ones that usually hide difficult-to-detect and tough-to-diagnose atomicity violation bugs.

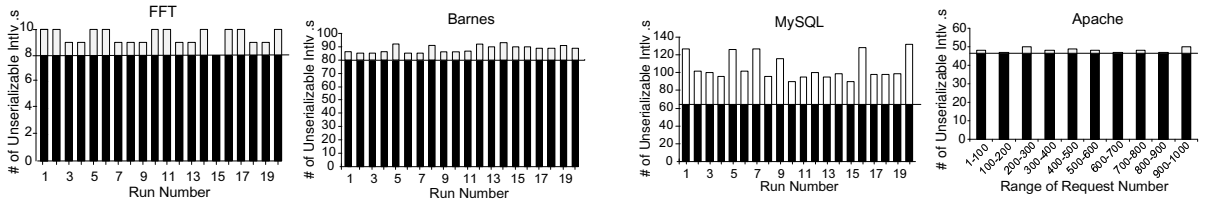
(3) *Why do some unserializable interleavings have low probability to be exercised?*

Different interleavings have completely different occurrence probabilities. For example, Figure 7.4 shows that some interleavings are exercised in all stress testing runs, i.e., about 100% occurrence probability. On the contrary, as we discussed above, some interleavings are never exercised during days of experiment, i.e., almost 0% probability. Further examination reveals the following major factors to determine the probability: (i) program synchronizations, such as lock, barrier, flag-synchronization, etc, that make some interleavings always happen and some never happen; (ii) distances between related instructions: when two memory accesses from a thread are close to each other, the chance is very small for them to be unserializably interleaved by a remote conflicting access; (iii) the number of dynamic instances of a static instruction: the more dynamic instances a static instruction has, the more likely that one of them will be unserializably interleaved.

7.3.3 Implications to Exposing Atomicity Violation Bugs

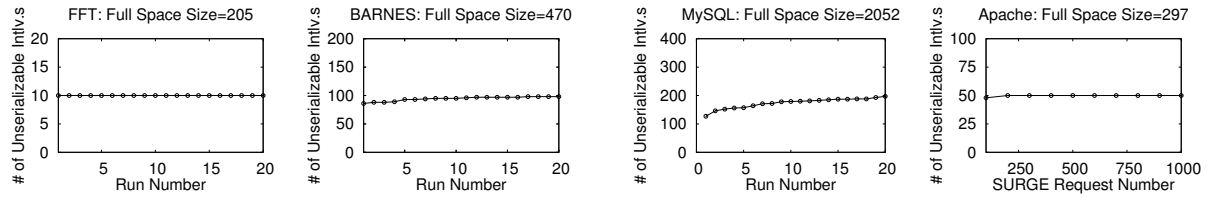
In summary, we can see that stress testing is not good at exposing atomicity violation bugs because it cannot effectively exercise the unserializable interleaving space. Without perturbation

[2] Since similar trends are demonstrated in all these applications and workloads, we only show the results for FFT, Barnes, MySQL-test (union) and Apache-HTTPd.



(a) Scientific applications (b) Server applications

Figure 7.4: Interleaving Similarity across runs. (Each bar shows the number of unserializable interleavings covered in each run. The dark part includes the interleavings exercised by *all* runs, i.e., having 100% occurrence frequency. The interleavings with less than 100% frequencies are included in the white part.)



(a) SPLASH2 applications (b) Server applications

Figure 7.5: The accumulative set of exercised unserializable interleavings grows slowly after the first few runs. (The *full space* includes all potential unserializable interleavings. How to calculate the full space is discussed in Section 7.4.1).

to the execution, stress testing repeatedly tests those high-probability unserializable interleavings. *Atomicity violation bugs can easily hide in those low-probability unserializable interleavings and escape into production runs.* Such bugs are usually the most obnoxious, difficult-to-catch and tough-to-diagnose concurrency bugs due to their rare occurrences (without external control) [LJZ07, LPSZ08, LTQZ06, XBH05].

Based on this observation, CTrigger focuses on exploring low-probability unserializable interleavings. In the following sections, we will discuss (i) how CTrigger identifies potential unserializable interleavings (Section 7.4.1); (ii) how CTrigger prunes out infeasible unserializable interleavings to avoid wasting testing efforts (Section 7.4.2); (iii) how to identify low probability interleavings to prioritize the testing runs (Section 7.4.3); (iv) how to systematically control the execution and make above low-probability interleavings more likely to occur (Section 7.5).

7.4 CTrigger Phase One: Identify Which Unserializable Interleavings to Focus On

Based on the observations described in above two sections, we design a framework called CTrigger to expose hidden atomicity violation bugs in concurrent programs. It is composed of two phases

(shown in Figure 7.2 in Introduction): at the first phase, for a given concurrent program and a given test input, CTrigger conducts trace analysis to obtain a list of unserializable interleavings for exploration. At the second phase, CTrigger explores these unserializable interleavings through controlled testing and exposes hidden atomicity violation bugs.

In this section, we discuss how CTrigger obtains the target unserializable interleaving list through three steps (marked as Step 1, 2 and 3 in Figure 7.2). We will discuss the second phase in the next section.

Please note that we take similar assumptions with recent work on concurrency testing [MQ07, Sen08, EFN+02]. We assume that programmers have a test case suite and they go through CTrigger's phase 1 and 2 for each test input. We also assume that, for one input, the code statements executed at different runs are mostly, maybe not completely, the same.

7.4.1 Step 1: Profiling and Identifying Potential Unserializable Interleavings

In CTrigger, we use a few profiling runs with a given test input to collect memory access information and conduct trace analysis to build the initial list of unserializable interleavings, which consists of potential (may not be all feasible) unserializable interleavings.

In a program, not every memory access instruction has its corresponding unserializable interleaving. Since an unserializable interleaving is composed of three accesses, a p(receding)-access, a c(urrent)-access and an r(emote)-access (refer to Section 7.2), at the first step, CTrigger goes through every memory access instruction C and checks whether C has a p-access and an r-access. If so, we identify interleaving-C as a *potential* unserializable interleaving.

Specifically, for an instruction C, a p-access is its preceding access from the same thread to the same memory location and an r-access is an access from a different thread to the same memory location. In addition, the access types of p-access, r-access, and C should match one of the four unserializable interleaving patterns shown in Figure 7.3. Both p- and r-accesses can be easily identified, if they exist, by checking the memory access trace of the program execution.

In CTrigger, this step is based on profiling. Certainly, it can also be done via static analysis, but static analysis will be limited by the pointer aliasing problem and the lack of concurrency information, especially for C/C++ programs.

7.4.2 Step 2: Pruning Infeasible Unserializable Interleavings

Among the potential unserializable interleavings, some can never happen due to synchronizations. It is important to prune them to avoid the vain attempt to force them. In this subsection, we discuss

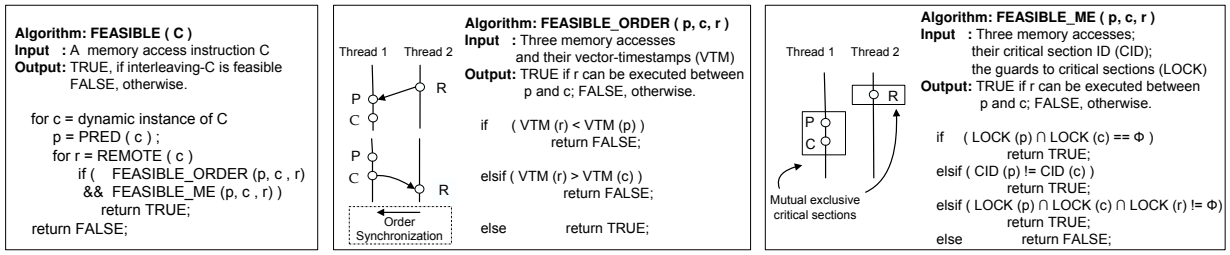
how CTrigger analyzes traces to prune out infeasible unserializable interleavings. Specifically, we want to identify tuples such as (p, c, r) where the r-access can never be executed between the p-access and the c-access.

Algorithms and Implementations

Different synchronization operations affect concurrent execution in different ways. Without losing generality, CTrigger categorizes synchronization operations into two types, *order* synchronization and *mutual exclusion*, and designs pruning algorithms accordingly (shown in Figure 7.6). Additionally, CTrigger also prunes other types of infeasible interleavings such as those caused by memory recycling.

Infeasible interleavings caused by order synchronization An order synchronization operation, such as a barrier and a thread create/join, forces certain order between events from different threads. In our scenario, order synchronization could force an r-access to be executed before p or after c. When either case occurs, r can never be executed between p and c. By checking this condition, CTrigger can prune out infeasible interleavings caused by order synchronizations. The process is shown on Figure 7.6(b).

In our implementation, CTrigger records all barrier and thread-create/join operations into the trace. In trace analysis, CTrigger uses vector timestamps to maintain and compare the order relationship between accesses. Note that vector timestamps used in CTrigger are similar but different from those used in conventional happens-before race detection algorithms [NM91]: CTrigger does **not** increase vector timestamps at lock/unlock operations, because lock/unlock does not force absolute orders.



(a) algorithm to distinguish feasible unserializable interleaving

(b) infeasible interleavings caused by order synchronization and algorithm to prune them

(c) infeasible interleavings caused by mutual exclusion and algorithm to prune them

Figure 7.6: CTrigger feasible interleaving analysis algorithm. (PRED and REMOTE denote preceding access(es) and remote accesses(es). They are collected in step 1.)

Infeasible interleavings caused by mutual exclusion Synchronization operations, such as a lock or a transaction, enforce mutual exclusions instead of orders among protected code regions.

When we consider mutual exclusions in the program, an r-access cannot interleave a p-access

and a c-access iff there exist two mutual exclusive critical sections so that one holds the r and the other holds both the p and c. Following this, we can prune infeasible interleavings caused by mutual exclusions (Figure 7.6(c)).

Specifically, CTrigger records all lock/unlock operations into the trace. During trace analysis, CTrigger maintains a lock set for each shared memory access and uses that to determine which critical section(s) the access belongs to. Different from the lock-set race detection algorithm [SBN+97], the lock-sets maintained by CTrigger record *dynamic*, rather than static, lock instances that protect each access. In this way, CTrigger can tell whether two accesses are inside the same critical section.

Memory recycling issue CTrigger also considers infeasible interleavings caused by memory address recycling. Specifically, two instructions may access the same memory address during the course of execution, but actually can never conflict with each other. It occurs when they access different program variables that happen to have the same address due to memory recycling. CTrigger prunes this type of infeasible interleavings by intercepting memory allocation and deallocation operations and differentiating memory locations allocated at different time.

Discussions

Above pruning analysis works well for real-world server programs written in C, as we will see in the experiments (Section 7.7). Most infeasible interleavings can be correctly identified. However, a small number of infeasible interleavings may be missed due to un-identified customized synchronization operations. This issue is handled at CTrigger's second phase: when trying to force an interleaving, CTrigger sets an expiration time for each artificial delay. Once the time expires, CTrigger gives up and continues exploring other interleavings. Since most infeasible interleavings are pruned, the wasted effort in phase two is very small.

Our current prototype can be extended to consider other synchronization operations. For example, if programs use transactional memory for synchronization, CTrigger can trace transaction-begin and transaction-end, and analyze the interleaving feasibility in a similar way to what we do in Figure 7.6 (c).

7.4.3 Step 3: Ranking Low-Probability Interleavings

As discussed in Section 7.3.2, different interleavings have different probabilities in occurrence during stress testing. Some interleavings rarely occur but have high likelihood to hide atomicity violation bugs, especially those bugs that are hard to reproduce for diagnosis. Therefore, it is desirable to identify and prioritize low-probability interleavings during testing in order to effectively expose bugs.

This section first discusses the major factors that affect the probability of interleavings. It then introduces our probability ranking metrics and explain the detailed ranking algorithms. Note that accurately calculating the interleaving probability is difficult and also unnecessary. CTrigger aims at using simple and yet effective metrics to select low-probability interleavings.

Two major Factors that Affect Interleaving Probability

The occurrence probability of an unserializable interleaving is affected by many factors in real applications. Among them, two factors are most important: how close the two local accesses (p-access and c-access) are, and how far away a remote access (r-access) is from the local accesses. Intuitively, when a p-access and a c-access are close to each other, the time window can be too small for a remote access (to the same memory location) to interleave in between. Similarly, when a remote access is far away from the local accesses, the chance of an interleaving is small. Above intuition is demonstrated in Figure 7.7 through a toy program. Section 7.7.5 will measure how CTrigger ranking mechanism based on the above intuitions has helped the bug exposing.

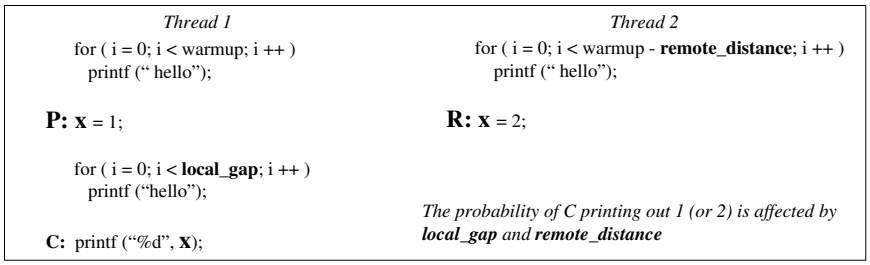


Figure 7.7: A toy example showing how local-gap and remote-distance affect the interleaving probability (assume thread 1 and 2 start execution at the same time).

Based on the intuition above, we define the following two simple metrics to estimate the probabilities and to rank the unserializable interleavings (Figure 7.8).

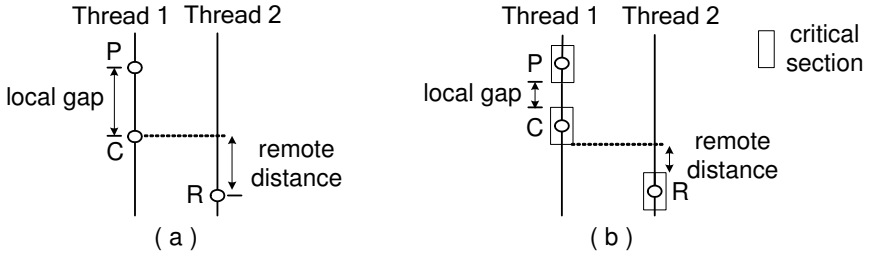


Figure 7.8: Local gap and remote distance.

- **Local gap** is the execution time distance between a p-access and a c-access for an unserializable interleaving (p, c, r) as defined in Section 7.2. This metric represents the size of an

interleavable window, i.e., the period where an r-access can interleave between the p- and c-accesses.

- **Remote distance** is the time difference between an interleavable window and an r-access. As remote distance increases, the r-access gets farther from the interleavable window and is less likely to interleave the p and c.

There is actually a big difference between the local gap and the remote distance. Local gap is the distance between two accesses from one thread. Therefore, it is stable across runs and is always a good indicator of the interleaving probability. On the contrary, the property of remote distance highly depends on the nature of applications. In some cases, such as that in Figure 7.7, remote distance has big impact on the interleaving probability. In some other cases, when the execution time of R is totally independent with the execution of P and C, the measured value of remote distance is completely random and should *not* be used to estimate the interleaving probability. Because of this, our current prototype of CTrigger uses the local gap as the primary ranking metric, and refers to the remote distance only when multiple interleavings have similar local gaps.

How to Compute the Metrics?

The main idea of CTrigger ranking mechanism is straightforward. CTrigger first analyzes the profiling run traces and gets the local gap for every unserializable interleaving. It then generates the ranking based on the local gaps: the smaller the local gap is, the higher an interleaving is ranked — as it is less likely to happen without external control.

Although the above basic idea is simple, several issues need to be addressed:

(1) How to measure the distance? We use CPU performance counter (accessible through `RDTSC` x86 assembly instruction) to measure local gaps. This scheme can include the different latencies of different operations, such as disk I/O, into gap information. Currently we do not consider the effect of context switches in local gap measurement. Fortunately, the time slice for preemptive context switches is very large, so only few instructions will be affected.

(2) How to deal with synchronizations between local accesses? Synchronization operations would affect the effective interleavable windows and thereby should be considered when calculating local gaps. For example, when each of p, c, and r accesses is protected by a same lock *separately* (Figure 7.8(b)), the local gap should measure the execution period starting from the end of p's critical section to the beginning of c's critical section. The rational is that r cannot be concurrently executed with either critical section that contains p or c.

(3) How to deal with multiple instances of the same static instruction? Intuitively, the more dynamic instances a static instruction has, the more likely an interleaving would occur. Consequently,

CTrigger takes *the summation* of all local gaps from all the dynamic instances of an unserializable interleaving.

At the end, CTrigger gets a list of likely feasible unserializable interleavings ranked based on estimated occurrence probability. CTrigger further excludes the interleavings that are already exercised during the profiling runs, and delivers the remaining list to its second phase: interleaving exploration and bug exposing.

7.5 CTrigger Phase Two: Explore Unserializable Interleaving Space

After CTrigger identifies and ranks unserializable interleavings in the first phase, it tries to force these interleavings via controlled execution. In this phase, CTrigger systematically controls the concurrent execution to exercise unserializable interleavings that are not yet exercised by profiling runs, starting from the ones with the lowest (estimated) occurrence-probabilities.

Execution control for one interleaving Unlike previous work such as CHESS [MQ07] and RaceFuzzer [Sen08, PS08] that control thread schedule and execute only one thread at a time, CTrigger controls execution by suspending a thread's execution at appropriate places to increase the occurrence probability of the targeting unserializable interleaving. The period of suspension is carefully controlled to avoid significant performance degradation.

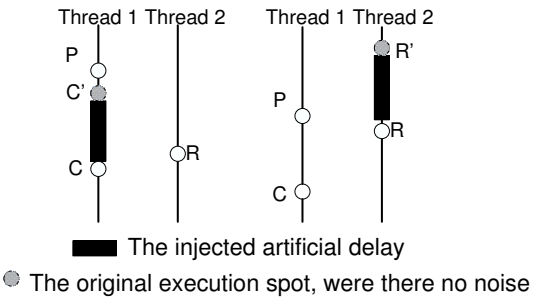


Figure 7.9: CTrigger's execution control

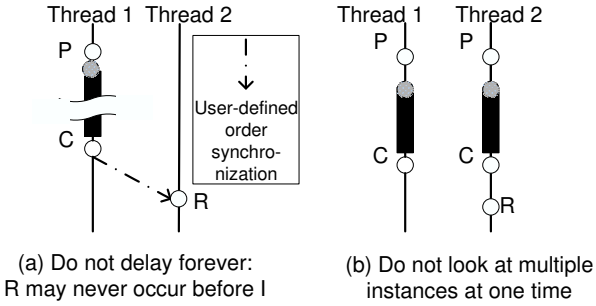


Figure 7.10: Issues in execution control

Specifically, for an unserializable interleaving, CTrigger suspends corresponding threads before its c-access C or r-access R whenever necessary during the execution (Figure 7.9). This can help increase the *local gap* and decrease the *remote distance* of the target unserializable interleaving, and therefore increase its occurrence probability.

Although above ideas are intuitive, there are several efficiency and effectiveness issues we need to address:

(1) How long should the suspension be? An intuitive answer is to suspend the execution until the interleaving occurs, i.e., suspend c's thread until r executes or suspend r's thread until c is ready to execute. Unfortunately, the unserializable interleaving may never occur, as shown in Figure 7.10 (a). To avoid such endless suspension (deadlock), CTrigger sets a time-out threshold for each suspension point.

(2) When should a thread be suspended? An intuitive answer is to suspend a thread whenever it is about to perform the c or r access as long as the interleaving has not occurred. However, this intuitive solution has several problems. First of all, when more than one thread (e.g., two) execute the c instruction (Figure 7.10 (b)), suspending both threads may actually decrease the interleaving probability. Therefore, CTrigger only suspends one thread at a time. Secondly, a static instruction might have many dynamic instances. Suspending before every instances can result in huge slow-downs. For efficiency concerns, CTrigger sets a threshold for the number of times that threads are suspended for each unserializable interleaving.

(3) The danger of waiting inside a critical section Suspending a thread inside critical sections might also block other threads that are waiting to enter critical sections. Although it will not lead to a deadlock, as CTrigger has an expiration time for each suspension, it may prevent the targeting interleavings from happening. CTrigger can address this issue by suspending the execution right before the outermost critical section that holds the targeting instruction.

(4) Context sensitivity The occurrence of some unserializable interleavings depends on the program context or thread context. That is, they only happen when the involved instructions are executed upon certain stack frame or by certain threads. CTrigger provides the option to collect call-stack and thread information from trace analysis and use such information in execution control.

Execution control for a list of interleavings Controlled testing for a ranked list of unserializable interleavings is a complex planning problem, because exploring one interleaving might interfere with the exploration of another interleaving. Facing this problem, CTrigger follows a simple greedy principle — one interleaving at a time. After the targeting interleaving occurs or the time expires, it moves on to the next interleaving. Note that it does not mean one interleaving per run. Each run can still explore multiple target interleavings.

As regards to which one to explore first, CTrigger provides two options. The first option is to simply go down the ranked list and explore unserializable interleavings one by one. While simple, it may be inefficient if a high-ranking interleaving appears late during the execution. The second option is to consider a set of interleavings with similar ranks at a time. CTrigger suspends execution for whichever interleavings whose involving instructions appear first. In our experiments, we use the first option for a short list of unserializable interleavings (such as those in SPLASH2 applications) and the second option for a long list of unserializable interleavings (such as those in sever applications).

Implementation CTrigger controls execution via binary instrumentation using Intel's PIN tool [LCM+05]. CTrigger takes the list of unserializable interleavings provided by previous CTrigger analysis, and instruments every instruction that involves in at least one unserializable interleaving. At run time, CTrigger intercepts every dynamic instances of these instructions and injects delay according to the above strategies.

Outcome Interpretation If a target unserializable interleaving is successfully forced by CTrigger's controlled execution and the software misbehaves, a bug is then exposed. Certainly, various bug detection tools can be used during the execution to detect subtle errors that do not lead to visible program misbehaviors (e.g., crashes). In this case, CTrigger records the specific execution control it added during the bug-triggering run and can retry the same control to reproduce the bug reliably for diagnosis.

If a target unserializable interleaving is successfully forced by CTrigger's controlled execution and the software does not misbehave or the underlying bug detection tool does not detect any error, programmers gain more confidence about the software quality in this case. In the meantime, benign atomicity violations can also be identified.

If the targeting interleaving does not happen even after the controlled execution, most likely the targeting interleaving is actually infeasible and it probably skips our pruning process due to customized synchronization mechanisms. Such information can still be useful as it can help identify customized synchronization operations and help bug detection and concurrent program analysis.

7.6 Methodology

To evaluate our ideas and CTrigger framework, we apply CTrigger on seven applications and evaluate how well it can expose the tested atomicity violation bugs inside these applications. These applications include three large open-source server/client applications, i.e., Apache, MySQL and Mozilla, one utility application, PBZIP2, and three SPLASH2 [WOT+95] benchmarks. We evaluate one or two *real-world* atomicity violation bugs in each application, including five in Apache, MySQL, Mozilla, and PBZIP2 and three in SPLASH2 introduced by external macro writers. The details of the evaluated applications and bugs are described in Table 7.1 (in Section 7.3.1) and Table 7.2. We chose to use real-world bugs instead of manually injected ones because the former is much more representative.

The platform setting is the same as that in Section 7.3.1. The selection of testing inputs for the server/client applications are based on the original bug reports on corresponding forums (since CTrigger focuses on testing the interleaving space, not the inputs, figuring out the bug-triggering inputs is out of our scope).

Application	Bug Id	Bug description
Apache	Apache#1	Server crash during cache management
	Apache#2	Log-file corruption
MySQL	MySQL	DB log missing database actions
Mozilla	Mozilla*	Wrong results of java-script execution
PBZIP2	PBZIP2	Crash during file decompression
FFT	FFT	A problem in platform-dependent macro (introduced by external macro providers) leading to atomicity violation bugs that generate wrong outputs
LU	LU	
Barnes	Barnes	

Table 7.2: Applications and atomicity violation bugs evaluated in CTrigger. (*: Mozilla code is slightly modified to help compare the execution result with the oracle.)

Note that, for all bugs, CTrigger does not assume any prior-knowledge about the bug-triggering interleavings. It strictly follows the process described in previous sections to systematically identify and exercise low-probability unserializable interleavings. For instance, we do **not** know the existence of the SPLASH2 macro bugs in advance. CTrigger exposes those bugs when it tests SPLASH2 using the default inputs.

We evaluate the effectiveness, efficiency and reproducibility of CTrigger: whether the bugs can be exposed, how quickly the bugs can be exposed, and how reliably the bugs can be reproduced after their first manifestation. We compare CTrigger with four other bug exposing mechanisms on the same platform as shown in Table 7.3.

Stress	Stress testing
Pure-Pin	Stress testing running upon the PIN binary instrumentation framework This is the baseline for the next three schemes, which are all implemented by us upon PIN.
Sync-based	A bug exposing mechanism that injects delay at synchronization operations just like ConTest [BFM+05]. The released version of CHESS [MQ07] is similar, also sync-based.
Race-based	A bug exposing mechanism that forces suspect data races reported by a race detector. This is similar to RaceFuzzer [Sen08]. Our implementation is based on PIN and the state-of-the-art open-source Valgrind-lockset race detection tool [NS07]. It is extended by our execution control to run multi-threads concurrently instead of one thread at a time like in the original RaceFuzzer.
CTrigger	Our method presented in this chapter

Table 7.3: Evaluated concurrency testing methods

7.7 Experimental Results

7.7.1 Efficiency and Effectiveness

Bug exposing time Overall, as shown in Table 7.4, CTrigger can expose all the tested atomicity violation bugs efficiently, within 1–235 seconds. It is about 10 to over 1000 times faster than all

BugId.	Stress	Pure-Pin	Synch-based	Race-based	**CTrigger**	CTrigger Speedup*
Apache#1	> **1 week**	NO	NO	NO	235.0	> 2573.6 X
Apache#2	80604.0	NO	14976.0	126.0	63.6	1267.4 X
MySQL	287.0	5431.0	3796.0	3.5	2.0	143.5 X
Mozilla	NO	NO	NO	65759.6	66.2	> 1305.1 X
PBZIP2	NO	NO	32.0	2.6	2.6	> 9391.3 X
FFT	673.0	2284	NO	NO	0.94	716.0 X
LU	188.6	3459	NO	NO	4.2	44.9 X
Barnes	248.7	NO	NO	NO	17.6	14.1 X

Table 7.4: Time (unit: second) spent to expose every tested atomicity violation bugs. (**NO**: the bug was not exposed in our maximum testing time, which is one day for Apache, MySQL, Mozilla, and half day for other small applications. *: the speedup is compared with stress testing.)

alternative testing methods for all tested bugs, except for Apache#2, MySQL and PBZIP2 bugs where its efficiency is comparable with Race-based testing. CTrigger is especially effective for large server/client applications. For example, CTrigger needs just 4 minutes to expose Apache bug#1, which can **not** be exposed by any alternative testing schemes within one full day. Actually, even after one week, the bug was still not exposed with stress testing (Note that this bug did appear during production runs and bothered the Apache server users. That is why it was reported in Apache's bugzilla database and was later fixed by developers). All these results indicate that CTrigger can greatly reduce the testing time and make atomicity violation bug detection and diagnosis more efficient.

It is not hard to understand that Pure-Pin testing is as ineffective as stress testing. Actually, since PIN framework (even without any instrumentation) slows down each testing run, it takes longer than stress testing to expose the tested atomicity violation bugs.

Synch-based testing perturbs the execution at synchronization points. It can help expose the PBZIP2 bug within half a minute, because this bug is caused by an unserializable access to a lock variable and the program crashes at lock acquisition time. However, Synch-based testing cannot help the other seven tested bugs because these bugs, like most real-world atomicity violation bugs, were introduced because programmers forgot to use synchronization operations. As a result, Synch-based testing slows down each testing run without improving the chances of exposing the tested bugs. Its bug exposing time is similar to that of Pure-Pin.

As regards to Race-based testing, the eight tested bugs can be divided into three categories. The first category includes Apache#2, MySQL, and PBZIP2. These bugs are successfully caught by Valgrind as race suspects. By leveraging the race detection results, Race-based testing can expose these bugs in similar amount of time with CTrigger. It is still slower than CTrigger in case of Apache#2, because the rareness-based ranking mechanism enables CTrigger to focus on buggy-interleavings earlier than Race-based testing. The second category includes Apache#1 and

BugId.	Profiling Runs	CTrigger Analysis	Controlled Testing
Apache#1	61.4	1.1	172.5
Apache#2	61.4	1.1	1.1
MySQL	0.90	0.10	0.90
Mozilla	8.0	1.0	57.2
PBZIP2	0.56	0.0006	2.01
FFT	0.52	0.23	0.19
LU	1.40	2.58	0.18
Barnes	4.88	7.81	4.94

Table 7.5: The breakdown of CTrigger bug exposing time (unit: second). (CTrigger analysis includes the three steps of setting up unserializable interleaving space.)

the three SPLASH2 bugs. Valgrind fails to detect these bugs and affects Race-based testing to be as inefficient as Pure-Pin and Synch-based testing. This indicates that the bug exposing capability of Race-based testing greatly relies on the underlying race detector's coverage. The last category is the Mozilla bug. Interestingly, the buggy code is reported by Valgrind as race suspects. However, the race between the reported racing instructions does not always lead to atomicity violation. Enforcing the race is insufficient to expose this atomicity violation bug.

CTrigger bug exposing time breakdown Table 7.5 shows the time spent in every step of CTrigger for exposing above bugs. CTrigger trace collection and analysis take about 1 to 60 seconds. The tracing time mainly depends on how fast the set of unserializable interleavings exercised by stress testing becomes stable, and the analysis time is affected by the execution's memory footprint size.

CTrigger needs less than 5 seconds of controlled testing to expose most of the tested atomicity violation bugs. Such efficiency is the combined effects of CTrigger infeasible interleaving pruning, ranking and execution control strategies. In almost all cases, the bug-triggering interleavings are ranked very high in the low-probability interleaving list (refer to Section 7.7.5 for detailed ranking results). As a result, the bugs are exposed very quickly after few seconds of controlled testing. However, in the cases of Apache#1 and Mozilla, the bug-triggering interleavings are ranked relatively low and thus take longer testing time. In both cases, multiple benign atomicity violations are exercised and validated to be benign before the bugs get exposed.

7.7.2 Unserializable Interleaving Coverage

CTrigger can effectively explore low-probability unserializable interleavings, and improve the coverage within the unserializable interleaving space.

Figure 7.11 shows the unserializable interleavings *additionally* explored by CTrigger for Apache compared with the stress testing (profiling runs). These additionally covered interleavings

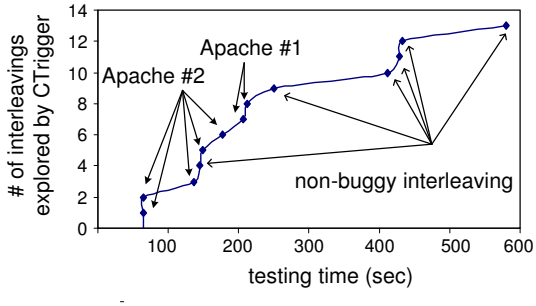


Figure 7.11: Unserializable interleavings additionally explored by CTrigger. (The base line is the unserializable interleavings covered in profiling runs. The first 60 seconds are devoted to profiling and have no additional coverage.)

include both bug-triggering ones and non-bug-related ones, as denoted by Figure 7.11. Covering bug-triggering ones helps CTrigger to expose the two Apache bugs; covering non-bug-related ones validates the correctness of these low-probability interleavings. In contrast, the number of interleavings explored in stress testing is saturated after around 60–70 seconds.

7.7.3 Reproducing a Previously-Exposed Bug

BugId.	Stress	Pure-Pin	Sync-based	Race-based	**CTrigger**	Speedup*
Apache#1	–	–	–	–	76.2	–
Apache#2	NO	–	11664.0	0.70	1.3	> 66461.5 X
MySQL	348.0	5239.7	10054.0	0.90	0.90	386.7 X
Mozilla	–	–	–	5.44	4.39	–
PBZIP2	–	–	0.43	0.52	0.44	–
FFT	1658	5633	–	–	0.18	9211 X
LU	562.3	NO	–	–	0.18	3124 X
Barnes	165.4	–	–	–	0.45	367.6 X

Table 7.6: Time (unit: second) spent to reproduce a exposed bug. (**NO**: the bug was not reproduced within one day. *: speedup is calculated based on stress testing. –: we do not measure reproducing time when the bug cannot be exposed even once as shown in Table 7.4.)

As shown in Table 7.6, CTrigger can efficiently reproduce all tested atomicity violation bugs, mostly within 5 seconds. This high bug reproducibility provided by CTrigger can greatly help programmers' bug diagnosis. CTrigger achieves the high reproducibility by recording and replaying its execution control. After an atomicity violation bug is exposed, CTrigger immediately knows which unserializable interleaving causes the manifestation of this bug. By repeating the same execution control and enforcing the same unserializable interleaving, the atomicity violation bug can be easily repeated.

Race- and Synch-based testing also record and repeat the perturbation they inject during the bug exposing runs. However, the perturbation record-and-replay scheme only helps the bug reproducing when the original bug exposing is directly *caused* by the perturbation, instead of by random effects. For example, Race-based testing can quickly repeat Apache#2, MySQL, and PBZIP2 bugs because the recorded perturbation and the enforced data races are the cause of the bug manifestation at the first places. In other cases, including almost all the cases of Synch-based testing, the perturbation is not the root cause of the bug exposing. Repeating it cannot help bug reproducing. For example, it still takes hours for Sync-based testing to reproduce Apache#2 and MySQL bugs.

Finally, as we can see in the table, for stress testing and Pure-Pin, reproducing a bug is always as difficult as exposing it at the first time, because neither mechanism records any interleaving information when a bug is exposed.

7.7.4 CTrigger Infeasible Interleaving Pruning

BugId.	# of Mem-Acc Instructions	# of Potential Unserializable Interleavings*	# of Feasible Unserializable Interleavings	Pruning Percentage (compared w/ potential ones)
Apache	2551	297	157	47.1%
MySQL	2257	113	25	77.9%
Mozilla	2376	76	48	36.8%
PBZIP2	149	93	25	73.1%
FFT	311	205	21	89.8%
LU	377	177	7	96.0%
Barnes	716	470	143	69.6%

Table 7.7: The effectiveness of the infeasible interleaving pruning. (* : The two Apache bugs can be triggered using the same input. Therefore, only one result is put here. MySQL uses a different input with that in Section 7.3.)

Identifying infeasible interleavings is critical for CTrigger to set a reachable testing goal. Table 7.7 shows that CTrigger feasibility analysis is very effective: 37–96% of the potential unserializable interleavings are successfully identified as infeasible. In order to examine the stability of the feasibility analysis results, we execute each SPLASH2 application for 20 times. The sets of feasible interleavings generated from each of these 20 runs are exactly the same.

7.7.5 CTrigger Low-Probability Interleaving Ranking

In this subsection, we evaluate how CTrigger ranking mechanism helps improve the efficiency of exposing hidden atomicity violation bugs.

For comparison, we applied an alternative scheme to decide the order of controlled testing: first come first serve. Specifically, we rank the unserializable interleavings based on their occurrence

order, rather than estimated occurrence probability, during the execution. Using this ranking, we similarly apply the controlled testing and measure how long it takes to expose the tested atomicity violation bugs.

As shown in Figure 7.12, CTrigger speeds up the alternative ranking method by up to 457.2 times in terms of total time to expose the tested bugs. This validates our observation that atomicity violation bugs usually hide inside low-probability interleavings. The results also show that CTrigger's ranking method is effective: CTrigger ranks the bug-triggering interleavings high, as shown in Figure 7.13, using its local-gap based probability estimation.

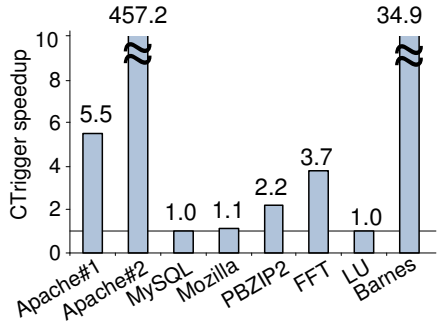


Figure 7.12: Speedups of CTrigger over the alternative ranking mechanism

Apache#1	7th out of 157
Apache#2	1st out of 157
MySQL	1st out of 25
Mozilla	14th out of 48
PBZIP2	3rd out of 25
FFT	2nd out of 21
LU	1st out of 7
Barnes	4th out of 143

Figure 7.13: CTrigger low-probability ranking: the rank of a bug-triggering interleaving among all feasible interleavings.

7.8 Summary

This chapter has presented a study of the interleaving characteristics in stress testing and proposed a new method, called CTrigger, to expose difficult-to-detect and tough-to-diagnose atomicity violation bugs that are often hidden in low-probability unserializable interleavings. CTrigger achieves this by selecting representative interleavings, pruning infeasible ones, identifying low-probability ones, and controlling program execution to improve their occurrence probabilities.

The experiments with seven real-world server/desktop and scientific applications on 8-processor machines show that CTrigger is effective at exposing atomicity violation bugs. It achieves 2 – 4 orders of magnitude speedup on bug exposing time over stress testing on the tested real-world atomicity violation bugs. For some server bugs that need several days of stress testing to manifest, CTrigger can expose them within 4 minutes. CTrigger can also reliably reproduce the atomicity violation bugs for diagnosis after their first manifestation, 2 – 5 orders of magnitudes faster than stress testing. With the significantly improved efficiency and reproducibility of bug ex-

posing, CTrigger well complements the existing techniques on improving the quality of concurrent programs: bug detectors can detect bugs more quickly and accurately; and developers can save a lot of efforts in bug diagnosis.

Our work is only the beginning on addressing the important problem of exposing atomicity violation bugs. It can be improved by more accurate infeasible interleaving pruning, better selection of rare interleavings and execution control. Future work can also combine it with test input generation and other interleaving testing mechanisms.

Chapter 8

Conclusions and Future Work

This dissertation makes contributions along three directions to address the concurrency bug problem and improve the dependability of the increasingly pervasive concurrent programs. These three directions are understanding concurrency bugs, detecting concurrency bugs, and exposing concurrency bugs.

Along the direction of understanding concurrency bugs, this dissertation presents one of the first comprehensive characteristic studies of a large set (105) of real-world concurrency bugs collected from four widely used C/C++ open-source applications. This study reveals many interesting properties regarding the bug patterns, manifestation conditions, and fix strategies of concurrency bugs. Specifically, the characteristics presented here have motivated and guided the following work of detecting atomicity violation bugs, multi-variable concurrency bugs, and designing interleaving coverage criteria in this dissertation. Future research on bug detection, testing, and concurrent programming language design can also benefit from this study.

Along the direction of detecting concurrency bugs, this dissertation makes the following contributions: proposes the idea of automatically inferring programmers' synchronization intention; identifies two important types of synchronization intentions (Access-Interleaving invariant and multi-variable access correlation); and builds two tools (AVIO and MUVI) to effectively detect two types of important concurrency bugs.

- The AVIO detection tool focuses on one of the most common concurrency bugs: atomicity violation bugs. It looks at atomicity violation bugs from the perspective of a new program invariant, the access-interleaving (AI) invariant, that is simple yet essential to synchronization correctness. AVIO automatically infers AI-invariants through training and detects atomicity violation bugs when AI-invariants are violated. In our evaluation, AVIO can effectively detect more atomicity violation bugs than existing tools with fewer false positives. In addition, the two designs of AVIO (pure software-based and hardware-supported) make it suitable for both online detection and offline diagnosis usage scenarios.

- The MUVI detection tool looks at an important type of concurrency bug that has rarely been studied before: the multi-variable concurrency bug. MUVI identifies a semantic property,

variable access correlation, as the essence behind this type of bug. It automatically infers variable access correlation through data-mining source code. Using the inferred access correlation, MUVI detects not only multi-variable concurrency bugs, but also inconsistent-update semantic bugs. Experiments show that MUVI can infer many access correlations with good accuracy. MUVI also detected many new bugs from widely used open-source applications, including Linux, Mozilla, and MySQL. Furthermore, the MUVI work demonstrates that variable access correlation is a common and important program semantic property. The access correlation inferred by MUVI can be used by other tools such as AutoLocker [MZGB06] and Colorama [CMvPT07] to improve the dependability of concurrent programs.

Finally, this dissertation contributes to the direction of exposing concurrency bugs by designing a hierarchy of interleaving coverage criteria and building a framework to expose atomicity violation bugs.

- The proposed coverage criteria hierarchy is composed of seven interleaving coverage criteria designs based on different concurrency fault models. These criteria span a wide spectrum of testing complexities and bug-exposing capabilities. Together, they provide good guidance for systematic exploration of the interleaving space.

- CTrigger is a testing framework that can effectively, efficiently, and reliably expose atomicity violation bugs. Guided by above coverage criteria study, CTrigger focuses on unserializable interleavings to expose atomicity violation bugs. CTrigger trace analysis algorithms can identify from the profiling traces what are feasible interleavings and what are low-probability interleavings. With this analysis setting the focus, CTrigger carefully controls the testing runs to exercise feasible and rare unserializable interleavings, and effectively expose atomicity violation bugs. Experiments show that CTrigger can expose the tested real-world atomicity violation bugs with 2–4 orders of magnitude speedup over stress testing, the current common practice. CTrigger can also reliably repeat the bug manifestation process, 2–5 orders of magnitudes faster than stress testing.

This dissertation has made contributions to understanding, detecting and exposing concurrency bugs. Of course, many open problems are still left as future work to finally address the concurrency bug problem.

Along the lines of bug detection, false negatives, false positives, and performance are three haunting problems. This dissertation presents AVIO and MUVI to catch two big categories of false negatives from previous bug detection tools. The proposed CTrigger framework can also provide some dynamic bug detection tools with a better testing coverage and therefore fewer false

negatives. It remains for future work to look at other types of concurrency bugs that cannot be detected yet. In terms of the false positive, in this dissertation, AVIO leverages the training process to effectively reduce the false positives. However, training does not work for many other bug detection approaches, including static analysis-based approaches and most existing data race detectors. How to make the training idea more general or find other techniques to reduce false positives is an interesting and challenging problem. High overhead is another big problem of concurrency bug detection. Bug detection overhead not only slows down the diagnosis process but also perturbs the program execution and affects the manifestation of concurrency bugs. In this dissertation, new hardware extension is designed to speed up atomicity violation bug detection. Whether we can leverage existing or simpler hardware extensions to support concurrency bug detection remains for future work.

Along the line of bug exposing, the work in this dissertation is only a starting point. First of all, how to accurately identify infeasible interleavings and how to effectively enforce specified unserializable interleavings still need further improvement and will likely benefit from more program analysis. Second, this dissertation, just like all previous work on this problem, assumes that input generation is taken care of by previous sequential testing techniques and is beyond the scope of interleaving testing. This assumption is fair and useful at the current stage. However, after interleaving testing techniques are improved, eventually, we need to consider inputs and interleavings together to support effective and practical concurrent program testing.

This dissertation does not look at how to avoid concurrency bugs and how to fix them after they are detected. Both are important research directions. Although avoiding concurrency bugs is very difficult, many recent proposals (e.g., transactional memory model) have been made in this direction. It is an interesting research topic to see how synchronization invariants would change under new programming language constructs. Currently, fixing bugs mostly depends on the programmers' manual effort and judgment. The characteristics study presented in this dissertation can provide guidance for future research to provide more support for the bug-fixing process.

References

[AAK+05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA*, 2005.

[AGEB08] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. Sharc: checking data sharing strategies for multithreaded c. In *PLDI*, 2008.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Bierre. High-level data races. The First International Workshop on Verification and Validation of Enterprise Information Systems, 2003.

[ATKS07] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Transactional programming in a multi-core environment. In *PPOPP*, 2007.

[BA82] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Inf.*, 18:31–45, 1982.

[BC98] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS*, June 1998.

[Bei90] Boris Beizer. *Software Testing Techniques, 2nd edition*. New York: Van Nostrand Reinhold, 1990.

[BFM+05] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *PPoPP*, 2005.

[BGM+00] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA*, 2000.

[BL02] Michael Burrows and K. Rustan M. Leino. Finding stale-value errors in concurrent programs. In *Compaq SRC Technical Note 2002-04*, 2002.

[BLR02] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.

[C+02] Jong-Deok Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[CC98] Subhachandra Chandra and Peter M. Chen. How fail-stop are faulty programs? In *FTCS*, 1998.

[CC00] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN*, 2000.

[CER] CERT. Cert advisories. http://www.cert.org/advisories/.

[CMC+06] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *PLDI '06*, 2006.

[CMvPT07] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *HPCA*, 2007.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Software Eng.*, 15(11):1318–1332, 1989.

[CTTC06] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, 2006.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *SOSP*, pages 73–88, 2001.

[DLFC08] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2008.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP*, 1990.

[DS91] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In ACM/ONR Workshop on Parallel and Distributed Debugging (AOWPDD), 1991.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[ECGN00] Michael Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[ECH+01] 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*, pages 57–72, 2001.

[EFN+02] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multi-threaded java program test generation. *IBM Systems Journal*, 2002.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *PLDI*, 2000.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[FFLM96] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka. Testing concurrent programs: a formal evaluation of coverage criteria. In *ICCSSE*, 1996.

[FM06] Azadeh Farzan and P. Madhusudan. Causal atomicity. In *CAV*, 2006.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS*, 2003.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 1988.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, 1975.

[GKIY03] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of linux kernel behavior under errors. In *DSN*, 2003.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[Gro] Edison Design Group. EDG C/C++ front end. http://www.edg.com.

[GZ03] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceeding of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Nov 2003.

[HDVT08] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, 2008.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[HH08] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, 2008.

[HJM04] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2004. ACM.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[HM92] Mary Jean Harrold and Brian A. Malloy. Data flow testing of parallelized code. In *Proceedings of the International Conference on Software Maintenance*, 1992.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05*, 2005.

[HSW+04] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4), 2004.

[HWC+04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[KRDV07] Nicholas A. Kidd, Thomas W. Reps, Julian Dolby, and Mandana Vaziri. Static detection of atomic-set serializability violations. Tech. Report TR-1623 , Computer Sciences Dept., Univ. of Wisconsin, 2007.

[KT96] Pramod V. Koppol and Kuo-Chung Tai. An incremental approach to structural testing of concurrent software. In *ISSTA*, 1996.

[KTB+06] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, Nov 2006.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[LDSC08] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[LELS05] Tong Li, Carla Ellis, Alvin Lebeck, and Dan Sorin. On-demand and semantic-free dynamic deadlock detection with speculative execution. In *USENIX Annual Technical Conference*, 2005.

[LJZ07] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (short paper)(ESEC/FSE07)*, September 2007.

[LLMZ04] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI*, 2004.

[LLQ+05] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: A benchmark for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[LPH+07] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *The 13th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS08)*, March 2008.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[LTQZ06] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *The 12th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS06)*, 2006.

[LTQZ07] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. *The IEEE Micro Special Issue on Top Picks from Computer Architecture Conferences*, October 2007.

[LTW+06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, 2006.

[LZ05] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, Sept 2005.

[Mat] Mathworld. Stirling's approximation. http://mathworld.wolfram.com/StirlingsApproximation.html.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[MC91] Sang L. Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *ASPLOS*, 1991.

[MCT08] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, 2008.

[MDG+98] Peter S. Magnusson, Fredrik Dahlgren, Hkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenstrom, and Bengt Werner. Simics/sun4m: A virtual workstation. In *Usenix Annual Technical Conference*, 1998.

[MH06] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 2006.

[Moi97] Mark Moir. Transparent support for wait-free transactions. In *11th International Workshop on Distributed Algorithms*, 1997.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 446–455, New York, NY, USA, 2007. ACM.

[MS00] E. Marcus and H. Stern. Blueprints for high availability. John Willey and Sons, 2000.

[MZGB06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.

[Nat02] National Institute of Standards and Technlogy (NIST), Department of Commerce. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002-10, June 2002.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, 2006.

[NM91] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[NPC06] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ASPLOS*, 2006.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[NWT+07] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[OWB05] Thomas Ostrand, Elaine Weyuker, and Robert Bell. Predicting the location and number of faults in large software systems. *TSE*, 2005.

[PBB+02] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, Technical Report UCB//CSD-02-1175, U.C.Berkeley, Mar 2002.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM.

[PK96] Dejan Perkovic and Peter J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, 1996.

[PLZ09] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their finding places. In *The 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS09) (to appear)*, March 2009.

[Prv06] Milos Prvulovic. Cord:cost-effective (and nearly overhead-free) order-reordering and data race detection. In *HPCA*, 2006.

[PS03] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP*, 2003.

[PS08] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.

[PT03] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, 2003.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies c a safe method to survive software failures. In *SOSP*, 2005.

[QW04] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI*, 2004.

[RHP+07] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP*, 2007.

[RL98] Brad Richards and James R. Larus. Protocol-based data-race detection. In *SPDT*, 1998.

[SA06] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *FSE*, 2006.

[SAWS05] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP*, 2005.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[SC92] Mark Sullivan and Ram Chillarege. A comparison of software defects in database management systems and operating systems. In *FTCS*, 1992.

[Sco98] Donna Scott. Assessing the costs of application downtime. Gartner Group, May 1998.

[SDT] SDTimes. Testers spend too much time testing. http://www.sdtimes.com/SearchResult/31134.

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

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[TLK92] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 1992.

[vPG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *OOPSLA*, 2001.

[VTD06] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[Wei88] Stewart N. Weiss. A formal framework for the study of concurrent program testing. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, 1988.

[Wey93] E. J. Weyuker. More experience with data flow testing. *IEEE Trans. Softw. Eng.*, 19(9):912–919, 1993.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[WS05] Liqiang Wang and Scott D. Stoller. Static analysis for programs with non-blocking synchronization. In *PPoPP*, 2005.

[XBH03] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[XBH05] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[XHB06] Min Xu, Mark D. Hill, and Rastislav Bodík. A regulated transitive reduction (rtr) for longer memory race recording. In *ASPLOS*, 2006.

[YP03] Cheer-Sun D. Yang and Lori L. Pollock. All-uses testing of shared memory parallel programs. *Software Testing, Verification, and Reliability Journal*, 2003.

[YRC05] Yuan Yu, Thomas Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[YSP98] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *ISSTA*, 1998.

[Z. 06] Z. Li et. al. Have things changed now? – an empirical study of bug characteristics in modern open source software. In *ASID workshop in ASPLOS*, 2006.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 1997.

[ZLL+04] Pin Zhou, Wei Liu, Fei Long, Shan Lu, Feng Qin, Yuanyuan Zhou, Sam Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *MICRO*, 2004.

[ZTZ07] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.

Vita

Research Interests

Software systems, computer architecture, software reliability (with focus on concurrent programs)

Education

- University of Illinois at Urbana-Champaign

  2008 Ph.D. Computer Science, Advisor: Professor Yuanyuan Zhou
- University of Science and Technology of China, Hefei, China

  2003 B.S. Computer Science

Research Experience

- Research Assistant, 2004–2008, University of Illinois, Urbana-Champaign, Illinois, USA

  Worked on system support for improving software reliability, with a focus on concurrent systems.

- Summer Intern, Summer 2005, Microsoft Research, Redmond, Washington, USA

  Worked on OS configuration vulnerability problems.

- Visiting Student, Sep. 2002 – May 2003, Microsoft Research Asia, Beijing, China

  Worked on video encryption algorithms.

MAIN TEACHING EXPERIENCE

- Teaching Assistant. Course: Computer Architecture I, University of Illinois at Urbana-Champaign, Spring 2004, Fall 2004
- Teaching Assistant. Course: Computer Architecture II, University of Illinois at Urbana-Champaign, Fall 2003

SELECTED AWARDS AND HONORS

- 2007 W. J. Poppelbaum Memorial Award, University of Illinois
- 2006 paper selected for the IEEE Micro Top Picks in Computer Architecture
- 2003 Guo Moruo Presidential Award, University of Science & Technology of China

PUBLICATIONS

Journal Articles

- **Shan Lu**, Joe Tucek, Feng Qin, and Yuanyuan Zhou, "AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants", *IEEE Micro Special Issue: **Top Picks** from Computer Architecture Conferences*, January-February 2007 Issue.
- Zhenmin Li, **Shan Lu**, Suvda Myagmar and Yuanyuan Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code", *IEEE Transactions on Software Engineering (**IEEE-TSE**)*, April 2006.

Conference Papers

- Soyeon Park, **Shan Lu**, Yuanyuan Zhou, " CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places", accepted by *14th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS-XIV**)*, March 2009.

- **Shan Lu**, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou, "Learning from mistakes — a comprehensive study of real world concurrency bug characteristics", *13th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS-XIII**)*, March 2008.
- **Shan Lu**, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca Popa, Yuanyuan Zhou, "MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs", *21st ACM Symposium on Operating Systems Principles (**SOSP'07**)*, October 2007.
- Joseph Tucek, **Shan Lu**, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou, "Triage: Diagnosing Production Run Failures at the User's Site", *21st ACM Symposium on Operating Systems Principles (**SOSP'07**)*, October 2007.
- **Shan Lu**, Weihang Jiang and Yuanyuan Zhou, "A Study of Interleaving Coverage Criteria", *15th ACM SIGSOFT Symposium on the Foundations of Software Engineering (**FSE'07**)* (short paper), September 2007.
- Joseph Tucek, James Newsome, **Shan Lu**, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou and Dawn Song, "Sweeper: A Lightweight End-to-end System for Defending Against Fast Worms", *2nd ACM SIGOPS EuroSys (**EuroSys'07**)*, March 2007.
- **Shan Lu**, Pin Zhou, Wei Liu, Yuanyuan Zhou, Josep Torrellas, "PathExpander: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection", *39th Annual IEEE/ACM International Symposium on Microarchitecture (**MICRO-39**)*, December 2006.
- **Shan Lu**, Joe Tucek, Feng Qin, and Yuanyuan Zhou, "AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants", *12th International Conference on Architecture Support for Programming Languages and Operating Systems (**ASPLOS-XII**)*, October 2006 (IEEE Micro Top Picks Award).
- Chad Verbowski, Emre Kiciman, Arunvijay Kumar, and Brad Daniels, **Shan Lu**, Juhan Lee, Yi-Min Wang, Roussi Roussev. "Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management", *7th Symposium on Operating System Design and Implementation (**OSDI'06**)*, November 2006.

- Chad Verbowski, Brad Daniels, Emre Kiciman, **Shan Lu**, Roussi Roussev, Yi-Min Wang and Juhan Lee. "Analyzing Persistent State Interactions to Improve State Management", *Joint International Conference on Measurement and Modeling of Computer Systems (**SIGMETRICS/Performance'06**)* (short paper), June 2006.
- Feng Qin, **Shan Lu** and Yuanyuan Zhou, "SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs", *10th International Symposium on High-Performance Computer Architecture (**HPCA'05**)*, February 2005.
- Zhenmin Li, **Shan Lu**, Suvda Myagmar and Yuanyuan Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code", *6th Symposium on Operating System Design and Implementation (**OSDI'04**)*, December 2004.
- Pin Zhou, Wei Liu, Long Fei, **Shan Lu**, Feng Qin, Yuanyuan Zhou, Samuel Midkiff and Josep Torrellas, "AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants", *37th Annual IEEE/ACM International Symposium on Microarchitecture (**MICRO-37**)*, December 2004.
- Keman Yu, **Shan Lu**, Jiang Li and Shipeng Li, "Half-pixel Motion Estimation Bypass Based on a Linear Model", *24th Picture Coding Symposium (**PCS'04**)*, December 2004.
- **Shan Lu**, Keman Yu, Jiang Li and Shipeng Li, "A Low Complexity 2-Power Transform for Video Compression", *4th International Conference on Information, Communications & Signal Processing (**ICICS'03**)*, December 2003.

Selected Refereed Workshop Papers

- Zhenmin Li, Lin Tan, Xuanhui Wang, **Shan Lu**, Yuanyuan Zhou and Chengxiang Zhai, "Have Things Changed Now? – An Empirical Study of Bug Characteristics in Modern Open Source Software", *1st Workshop on Architectural and System Support for Improving Software Dependability (**ASID**)* held together with *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.
- **Shan Lu**, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou and Yuanyuan Zhou, "BugBench: A Benchmark for Evaluating Bug Detection Tools", *Workshop on the Evaluation of Software*

*Defect Detection Tools (**Bug 2005**)* held together with *Programming Language Design and Implementation (PLDI)*, June 2005.