Finding and Tolerating Concurrency Bugs

by

Jie Yu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2013

To my family.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Professor Satish Narayanasamy, for his endless support during the past five years. It has been an honor to work with him and to be one of his first batch of PhD students. From him, I have learnt how to write papers, make presentations, and most importantly, be confident and strong. Satish has given me the freedom to pursue various projects that match my interests. He has also provided countless insightful discussions about my research during our weekly meetings. I appreciate all his contributions of time, ideas, and funding to make my PhD experience productive and stimulating.

I also would like to thank other members of my PhD dissertation committee, Professor Robert Dick, Professor Jason N. Flinn, Professor Scott Mahlke and Dr. Cristiano Pereira. It is no easy task, reviewing a thesis, and I am grateful for their invaluable comments and constructive criticisms that greatly improved my dissertation.

This dissertation is partially funded by National Science Foundation (NSF) and Intel Corporation, and I would like to thank both organizations for their generous support. I would like to thank Dr. Cristiano Pereira and Dr. Gilles Pokam from Intel for their valuable discussions and insights from industry perspectives.

I also would like to thank all my labmates. I would like to thank Dongyoon Lee for sharing ideas and brainstorming with me throughout my PhD studies, Abhayendra Singh for clarifying my uncertainties about memory consistency issues in many occasions, and Chun-Hung Hsiao for working with me on the smartphone project offering me unremitting assistance. Thanks also go to other members in the lab, including Shaizeen Aga and Gaurav Chadha, for making the lab such a comfortable home for me.

My time at Michigan was made enjoyable in large part due to the many friends and groups that became a part of my life. I would like to thank Lujun Fang and Yunjing Xu for always hanging out with me, sharing their visions and big ideas. Special thanks go to Yunjing for taking me to the hospital in midnight when I was bleeding badly. I would like to thank Lujun Fang and Yudong Gao for being my roommates for years, making a sweet home for me. Special thanks go to Yudong for playing basketball with me all the time. Thanks also go to my friends who have not been mentioned yet, including Junxian Huang, Yi Li, Feng Qian, Li Qian, Zhiyun Qian, Zhaoguang Wang, Qiang Xu and Xinyu Zhang, for providing support and friendship that I needed.

Lastly, I would like to thank my family for their unconditional love and support. My hard-working parents, Jianzu Yu and Shuyan Li, have sacrificed their lives for me and provided unconditional love and care. I love them so much, and I would not have made it this far without them. And most of all, I would like to thank my loving, supportive and encouraging wife Panmei Chen. For this PhD, we have been apart for almost three years. This dissertation would not be possible without the love and support from her.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Finding and Tolerating Concurrency Bugs

by
Jie Yu

Chair: Satish Narayanasamy

Shared-memory multi-threaded programming is inherently more difficult than single-threaded programming. The main source of complexity is that, the threads of an application can interleave in so many different ways. To ensure correctness, a programmer has to test all possible thread interleavings, which, however, is impractical. Many rare thread interleavings remain untested in production systems, and they are the major cause for a majority of concurrency bugs.

Given that untested interleavings are the major cause of a majority of the concurrency bugs, this dissertation explores two possible ways to tackle concurrency bugs in this dissertation. One is to expose untested interleavings during testing to find concurrency bugs. The other is to avoid untested interleavings during production runs to tolerate concurrency bugs. The key is an efficient and effective way to encode and remember tested interleavings.

This dissertation first discusses two hypotheses about concurrency bugs: the small scope hypothesis and the value independent hypothesis. Based on these two hypotheses, this dissertation defines a set of interleaving patterns, called interleaving idioms, which are used to encode tested interleavings. The empirical analysis shows that the idiom based interleaving encoding scheme is able to represent most of the concurrency bugs that are used in the study.

Then, this dissertation discusses an open source testing tool called Maple. It memoizes tested interleavings and actively seeks to expose untested interleavings. The results show that Maple is able to expose concurrency bugs and expose interleavings faster than other conventional testing techniques.

Finally, this dissertation discusses two parallel runtime system designs which seek to avoid untested interleavings during production runs to tolerate concurrency bugs. Avoiding untested interleavings significantly improve correctness because most of the concurrency bugs are caused by untested interleavings. Also, the performance overhead for disallowing untested interleavings is low as commonly occuring interleavings should have been tested in a well-tested program.

CHAPTER I

Introduction

Multi-threaded programming is inherently harder than von Neumann style single-threaded programming. The number of possible states at a program statement exponentially increases with the number of threads executed, as the memory operations in a thread could interleave with the memory operations in the other threads in many different orders. Understanding, testing and verifying the correctness of all possible thread interleavings is impractical. Programmers tend to test only a small fraction of all possible legal thread interleavings in an application before shipping it to customers. The remaining untested interleavings are the major cause of a majority of concurrency bugs.

Given that concurrency bugs are typically caused by untested interleavings, we explore two possible research directions to tackle them. First, we propose a tool, called Maple, that tries to expose as many untested interleavings as possible during testing. This increases the chance of exposing any latent concurrency bug before the program is shipped. Even so, the interleaving space for a given program is so huge that Maple cannot practically expose all untested interleavings. For the remaining untested interleavings, we propose another approach which seeks to avoid them during production runs. This approach has the potential to tolerate most of the concurrency bugs as untested interleavings are the major cause of a majority of concurrency bugs. The key to both techniques is a way to encode and remember tested interleavings, based on which we can either seek to expose untested interleavings during testing, or try to avoid untested interleavings during production runs.

A thread interleaving is typically defined to be the order in which the memory operations are executed by all the threads in an execution. One natural way to encode a tested interleaving is to record the partial order of the memory operations executed by all the threads in an execution. However, this encoding scheme is not suitable for our purpose. Under such an encoding scheme, an encoded tested interleaving is only meaningful for a particular test run and becomes hardly useful when a different program input is used. Therefore, for a given program, there might exist an infinite number of untested thread interleavings. If one randomly picks an untested interleaving to test, the probability of exposing a concurrency bug is low. Avoiding untested interleavings at runtime using such an encoding scheme is also not practical as it is likely that many of the interleavings the system encounters at runtime are not tested. Finally, checking if an interleaving has been tested or not requires checking every memory operations involved in the interleaving, which is very expensive.

Hence, the challenge is to derive a way to encode tested interleavings such that it is generic enough for different program inputs. At the same time, it should be able to capture the set of all tested interleavings so that by exposing or avoiding untested interleavings, we can find and tolerate most of the concurrency bugs. Finally, checking if an interleaving has been tested or not can be carried out efficiently.

To address the above challenges, our high level idea is to define a *finite* set of *interleaving fragments* that can possibly be exposed for a given program. The set of interleaving fragments should be able to represent an infinite number of legal thread interleavings of that program. At the same time, if one manages to expose the set of interleaving fragments, it is sufficient to expose a majority of concurrency bugs in this program. Tested interleavings are encoded using the set of interleaving fragments that are exposed during testing. We want the set of interleaving fragments to be as small as possible such that by exposing an untested interleaving fragment, the probability of finding a concurrency bug is much higher than randomly picking an untested thread interleaving from the infinite legal thread interleaving space.

1.1 Two Hypotheses and Interleaving Idioms

In order to define interleaving fragments, we first discuss two hypotheses we make about concurrency bugs. These two hypotheses are the foundation of this dissertation. The way we encode and remember tested interleavings as well as the tools for exposing and avoiding untested interleavings are all based on these two hypotheses.

The first hypothesis is called *small scope hypothesis*. It is originally proposed by Jackson and Damon [34]. A recent adoption of this hypothesis on concurrency bugs [49] states that most concurrency bugs can be exposed using a small number of preemptions. CHESS [52] exploits this observation to bound the number of preemptions to reduce the search space. Our adoption of this hypothesis is that we focus on those simple concurrency bugs that involve no more than two threads, two variables and two inter-thread dependencies.

Surely there exist concurrency bugs that are more complex, and our technique is not able to handle them. However, previous study has shown that complex concurrency bugs are less likely to be exposed than simple ones [12]. Given the fact that we cannot practically expose all concurrency bugs in a program within a constant time budget, we prefer to spending more time on those concurrency bugs that are more likely to occur in production environment, which are, in order words, the less complex ones.

We propose a second hypothesis about concurrency bugs called *value independent hypothesis*. It states that a majority of concurrency bugs gets triggered if the erroneous inter-thread memory dependencies are exposed, irrespective of the program input and the data values of the shared variables involved in the dependencies. This hypothesis is based on our observation of several concurrency bugs in real-world applications. In fact, this hypothesis is implicitly assumed by many of the previous work on detecting, exposing and tolerating concurrency bugs [43, 44]. Given the value independent hypothesis, it is safe to choose an interleaving encoding scheme that is input and value oblivious.

Based on these two hypotheses, we introduce a set of *interleaving idioms* which we use as a foundation to define interleaving fragments and encode tested interleavings. Each interleaving idiom is a pattern of inter-thread dependencies through shared-memory accesses, which defines a type of interleaving fragments that we are interested in. In this dissertation, we focus on a set of canonical idioms that involve no more than two threads, two variables and two inter-thread dependencies, according to the small scope hypothesis.

An instance of an interleaving idiom is called an *iRoot*, which is represented using a set of static memory instructions. We say an iRoot is exposed in an execution if any part of the thread interleaving in that execution satisfies the interleaving pattern specified by the interleaving idiom and the involving static memory instructions match that specified in the iRoot. We use iRoots to represent interleaving fragments.

The tested interleavings can therefore be encoded using a set of iRoots exposed during test runs. Our empirical analysis shows that such an encoding scheme is able to capture 97% (33 out of 34) of the concurrency bugs [1] we have analyzed.

1.2 Finding Concurrency Bugs by Exposing Untested Interleavings

There have been significant work on detecting various kinds of concurrency bugs such as data races [21, 19], atomicity violations [43, 22], order violations [71] and deadlocks [53]. They can be divided into two major categories: static tools and dynamic tools. Static tools detect concurrency bugs by analyzing program code statically. These tools usually produce a large volume of false positives, thus preventing them from being widely adopted by programmers. Dynamic tools, however, can be very precise. They analyze an execution of a program and report potential concurrency issues. However, these dynamic tools can find a concurrency bug only if an appropriate execution is exposed. Exposing thread interleavings that exhibit concurrency bugs is therefore another challenge that we face.

Exposing concurrency bugs is much more difficult than exposing sequential bugs because it demands exploring not only the input space but also the huge thread interleaving space. In fact, almost all the existing testing techniques for multi-threaded programs focus on exploring different thread interleavings and assume a fixed program input. One common practice for exposing concurrency bugs is stress testing [18, 12], where a multi-threaded program is subjected to extreme scenarios during a test run. This method is not adequate because executing a program again and again over an input tends to unnecessarily test similar thread interleavings and has less likelihood of exposing a rare buggy interleaving. An alternative to stress testing is systematic testing [26, 52], where the thread scheduler systematically explores all legal thread interleavings for a given test input. However, this approach does not scale well for large programs because of the huge interleaving space for those

[1] When we say a concurrency bug, we mean a *static* concurrency bug.

programs. Active testing [59, 73, 69] is another recent development. A typical active testing tool focuses on a specific type of concurrency bugs. It only tests a small set of representative interleavings that are more likely to expose that type of concurrency bugs. As a result, a programmer may not be able to determine whether she should continue testing or not after the active testing tool finishes.

To address the above issues, we propose a coverage driven testing tool, called Maple, for exposing concurrency bugs. Maple memoizes past tested interleavings using the idiom based encoding scheme discussed above, and guides each future test run to explore new untested interleavings. It leverages the value independent hypothesis to test for an interleaving only once, and avoids testing the same thread interleaving again and again across different test inputs. Thus, the number of thread interleavings to test would progressively reduce as we test for more inputs. Our experiences in using Maple to test real-world applications shows that Maple is able to expose interleavings and exposing concurrency bugs faster than other existing testing techniques for multi-threaded programs. We also find 3 previously unknown bugs, which further demonstrates the bug exposing capability of Maple.

1.3 Tolerating Concurrency Bugs by Avoiding Untested Interleavings

Even with a state-of-the-art testing tool like Maple, we cannot practically test all thread interleavings of a given program because of its huge interleaving space. For the remaining untested interleavings, one promising approach is to avoid them during production runs. This improves correctness because untested interleavings are the major cause of a majority of concurrency bugs. At the same time, avoiding untested interleavings at runtime should have little performance impact as the commonly occuring interleavings should already be tested for a well-tested program.

Concurrency bug avoidance technique [64, 62, 45] has gained increasing attentions recently. It serves as the last safety net for the production system to prevent the concurrency bugs from manifesting in production runs. Some techniques integrate dynamic bug detection tools with checkpoint and re-execution systems [62, 64]. In these tools, once a bug is detected, the program will be rolled back to a previous checkpoint and re-execute so as to bypass the buggy interleavings. Other techniques [45] limit the freedom that threads can interleave in production systems such that some types of concurrency bugs will less likely to be exposed in production runs. For example, Atom-Aid [45] dynamically groups instructions into chunks and enforces a serial order between them. However, none of the previous technique takes tested interleavings into account and actively seeks to avoid untested interleavings. As a result, they are very sensitive to false positives, thus limiting their capabilities on avoiding variety of concurrency bugs.

In this dissertation, we propose two runtime system designs for tolerating concurrency bugs. Both systems remember tested interleavings and actively seek to avoid untested interleavings in production runs. Though possible, a software solution would incur significant performance overhead to the runtime system. Therefore, a hardware solution is a must. We first discuss a customized shared memory multiprocessor design for tolerating concurrency bugs. Like Maple, it encodes tested interleavings based on interleaving idioms. However, to ensure a complexity-effective hardware design, we only consider the simplest interleaving idiom which involves only one inter-thread dependency. Even though, we are still able to avoid at least 60% of the concurrency bugs according to our empirical analysis. We encode the tested interleavings in a program's binary executable using *Predecessor Set* (PSet) constraints. These constraints are efficiently enforced at runtime using processor support, which ensures that the runtime follows a tested interleaving. We analyze 17 bugs in open source applications and show that, by enforcing PSet constraints, we can avoid variety of concurrency bugs.

Though the PSet based runtime system is effective in tolerating many types of concurrency bugs, it has two major drawbacks. First, since we only consider the simplest interleaving idiom that involves only one inter-thread dependency due to hardware complexity concern, the system cannot avoid those concurrency bugs that involve multiple variables and inter-thread dependencies. For example, it cannot avoid multi-variable atomicity violations [41]. Second, the technique requires non-trivial changes to the commodity hardware, preventing it from being widely adopted.

As Hardware Transactional Memory (HTM) becoming a reality [2], we propose another runtime system design, called LifeTx, to tolerate concurrency bugs. It is based on a new type of interleaving constraints called *lifeguard transaction* (LifeTx), which is designed to be enforcible by HTM. Lifeguard transactions are similar to the programmer specified transactions [30, 32] in that it instructs the runtime to execute them in a serializable order. The difference is that LifeTxes are automatically derived based on interleavings observed during testing. LifeTx constraints can be efficiently enforced by a new hardware design similar to the eager conflict detection capability that exist in a conventional hardware transactional memory (TM) systems [32, 48], but without the need for versioning, rollback and unbounded TM support [8]. This greatly simplify the hardware design. Our experiments show that LifeTx is able to avoid 11 out of 12 atomicity violation bugs, including multi-variable atomicity violations, with only 0.6% performance overhead on average.

It is not unusual that the production environments (e.g. program inputs, hardware platforms, OS workloads) are substantially different from the testing environments and the frequencies with which particular interleavings occur might be different from what we observed during testing. To understand the degree to which such heterogeneity between testing and production environments can affect the effectiveness and efficiency of our interleaving constrained systems, we make sure that the program inputs used in production runs are different from those used in testing in all of our experiments. Moreover, we try to use different hardware platforms and operating systems for testing and production runs. For instance, in LifeTx, we simulate the production environment in a simulator which is configured to be very different from the real machine in which testing is conducted. In spite of that, our interleaving constrained systems are still effective in avoiding concurrency bugs with little performance overhead as indicated by our results, which provides some evidences about the ability of our technique in tolerating the heterogeneity between testing and production environments. Nevertheless, if other factors which we have not studied cause performance problems in productions runs, we always have a choice to turn off the protection to recover performance if we find the performance overhead is too high and outweighs the extra reliability we get.

1.4 Contributions

In summary, this dissertation makes the following contributions:

- We discuss two hypotheses about concurrency bugs: the small scope hypothesis that we adopt and the value independent hypothesis that we propose. Based on these two hypotheses, we define a set of interleaving idioms which we use to encode tested interleavings. Being able to encode tested interleavings enables us to build better tools for finding and tolerating concurrency bugs as we can concentrate on exposing and avoiding untested interleavings.

- We build a coverage driven testing tool called Maple. It memoizes tested interleavings based on interleaving idioms and actively seeks to expose untested thread interleavings as much as possible. We discuss our experiences in using the tool to expose 13 known and unknown bugs in real-world applications such as Apache and MySQL. Our results show that Maple is able to expose interleavings and expose concurrency bugs faster than other conventional testing techniques.

- We propose a customized shared memory multi-processor design for tolerating concurrency bugs. It is optimized for the simplest interleaving idiom which only involves one inter-thread dependency. We encode the tested interleavings in a program's binary executable using Predecessor Set (PSet) constraints. These constraints are efficiently enforced at runtime using processor support, which ensures that the runtime follows a tested interleaving. We analyze 17 concurrency bugs in open source applications such as MySQL, Apache, Mozilla, etc., and show that, by enforcing PSet constraints, we can avoid not only data races and atomicity violations, but also other forms of concurrency bugs.

- Following the PSet work, we discuss another hardware design for tolerating atomicity violation bugs. It is based on a new interleaving constraints called lifeguard transaction (LifeTx). LifeTx constraints can be efficiently enforced by a new hardware design similar to the eager conflict detection capability that exist in a conventional Hardware Transactional Memory (HTM) systems, but without the need for versioning, rollback and unbounded TM support. We show that 11 out of 12 atomicity violation bugs in programs like Apache, MySQL and Mozilla could be avoided using the proposed approach for only 0.6% performance overhead on average.

- We have made several contributions to open source community. We released a set of concurrency bugs that we collected from a few open source applications [1]. This online bug database has already been used and cited by dozens of research papers. We also made Maple, along with a dynamic analysis framework for concurrency programs, available to the public under the Apache 2.0 license [4]. It has already been adopted by a few researchers.

1.5 Structure

This dissertation is organized as follows. Chapter II discusses the background and the related work on finding and tolerating concurrency bugs. Chapter III defines the interleavings idioms and discuss two hypotheses about concurrency bugs which we use as a foundation of this dissertation. Chapter IV presents Maple, a coverage driven testing tool for exposing concurrency bugs. Chapter V and Chapter VI discuss two approaches for tolerating concurrency bugs. Chapter V describes the PSet based interleaving constrained shared memory multi-processor. Chapter VI discusses LifeTx, a transaction based approach for tolerating atomicity violation bugs. We discuss our future work in Chapter VII and conclude the dissertation in Chapter VIII.

CHAPTER II

Background and Related Work

In this chapter, we discuss previous work on finding and tolerating concurrency bugs. We first discuss existing approaches on detecting concurrency bugs (Section 2.1). Then, we discuss current testing techniques for exposing concurrency bugs (Section 2.2). Finally, we discuss existing runtime system solutions for tolerating concurrency bugs (Section 2.3).

2.1 Detecting Concurrency Bugs

Concurrency bug detection is among one of the most widely studied techniques for finding concurrency bugs. Numerous tools have been made to find data races [19, 77, 21], atomicity violations [24, 43, 22], deadlocks [53] and other forms of concurrency bugs [71]. These tools report potential concurrency errors in a program either by analyzing the program statically (static tools), or by analyzing its dynamic executions (dynamic tools).

Static bug detection tools [19, 77, 53, 24, 41] predict concurrency bugs in a program by analyzing the program statically. They usually do not produce false negatives. However, one of the common problems with many static bug detection tools is that they produce a large volume of false positives. The reason is, without knowing the actual program states at runtime, these tools usually make conservative assumptions about program executions, many of which are not feasible. Because of the high false positive rate, static bug detection tools are not widely adopted by programmers.

Dynamic bug detection tools [21, 43, 22, 85, 80], however, can be very precise and produce no false positive [21, 22]. They monitor dynamic program executions and report concurrency errors that manifest in those executions, or predict errors that will potentially manifest in alternate executions. Since dynamic bug detection tools rely on dynamic program executions, they might produce false negatives because some bugs can only be detected under specific interleavings. Typically, they do not actively seek to perturb program executions to reduce false negatives. Another disadvantage of dynamic bug detection tools is that the runtime overhead is usually high because of the program monitoring cost.

2.1.1 Data Race Detection

A data race can be defined as a pair of memory accesses to the same memory location, where at least one of the accesses is a write, and neither one happens-before the other. Dynamic data race detectors can be classified into two major categories: happens-before based and lockset based. A happens-before data race detector [55] finds only the data races that manifest in a given program execution. Lockset based techniques [68] can predict data races that have not manifested in a given program execution, but can report false positives.

Not all data races are harmful data races [54]. Many of the data races in production systems are benign, as programmers intentionally allow data races to optimize performance. Programmer constructed synchronizations could also result in benign data races. Benign data races also tends to be frequent, as opposed to harmful data races [54]. Therefore, a data race is really a heuristic, which is used to detect a particular interleaving pattern (concurrent memory accesses with no happens-before relation between them) that strongly correlates with the concurrency bugs that programmers tend to make.

2.1.2 Atomicity Violation Detection

Atomicity violations is another major source of concurrency errors [82, 43, 58, 20, 7, 22]. An atomicity violation occurs when a programmer assume a code region to be atomic, but fails to enforce its atomicity. Most of the atomicity violation detectors rely on programmers to specify the atomic regions through annotations. Some tools [82, 43, 41] use heuristics to automatically infer atomic regions. Detecting atomicity violations becomes straightforward once the atomic regions are determined.

2.2 Exposing Concurrency Bugs

Concurrency bug detection is one way to find concurrency bugs. Another way to find concurrency bug is to actually expose them in real executions. In this section, we discuss previous work on concurrent software testing.

2.2.1 Coverage Driven Testing

Coverage metrics for single-threaded programs are well studied in the literature. However, defining an effective coverage metric for multi-threaded programs is much more difficult and remains an open problem. There have been a few studies on coverage metrics for concurrent programs [74, 83, 11, 40, 70, 38]. Taylor et al. [74] presented a family of coverage criteria for concurrent Ada programs. All-du-path [83] is a coverage metric for concurrent programs which is based on definition-use pairs. Sherman et al. [70] discussed a few coverage metrics based on synchronizations and inter-thread dependencies. However, none of these work discusses a synergistic set of testing tools that can help programmers achieve high coverage for the proposed coverage metric and analyzes its effectiveness in exposing concurrency bugs.

2.2.2 Stress Testing and Random Testing

Stress testing is still widely used in software industry today. A parallel program is subjected to extreme scenarios during testing with the hope of increasing the likelihood of exposing buggy interleavings. This method is clearly inadequate since naively executing a program again and again over an input tends to unnecessarily test similar thread interleavings. A few techniques have been proposed to improve stress testing. The main idea is to randomize the thread interleavings so that different thread interleavings will be exercised in different test runs. These techniques mainly differ in the way in which they randomize thread interleavings. For example, ConTest [18] injects random delays at synchronization points. PCT [12] assigns a random priority to each thread and changes priorities at random points during an execution. However, all of these random testing techniques still suffer a common problem: the probability of exposing a rare interleaving that can trigger a concurrency bug is very low given that the interleaving space is so huge.

2.2.3 Systematic Testing

An alternative to stress testing is systematic testing [31, 26, 52, 78], which tries to explore all possible thread interleavings for each test input. Even with partial order reduction techniques [25, 23], the number of thread interleavings to test for a given input is still huge. Therefore, a few heuristics have been proposed to further reduce the testing time at the cost of missing potential concurrency errors. CHESS [52] bounds the number of preemptions in each test run. HaPSet [78] records PSet (discussed in Chapter V) during testing and guides the systematic search towards those interleavings that can produce new PSet dependencies. Even though, these tools still suffer from scalability problem, especially for long-running programs. Furthermore, these tools do not have a way to remember tested interleavings across different inputs. Finally, systematic testing tools usually require a closed unit testing environment which is not easy to setup in practice. However, systematic testing tools do have one distinct advantage in that they can provide certain guarantees to find a concurrency bug in a program for a given input.

2.2.4 Active Testing

Active testing has recently emerged as a new way to test concurrent software [69, 59, 58, 73, 39, 33, 36]. A typical active testing tool has two phases: a prediction phase and a validation phase. In the prediction phase, active testing tools use approximate bug detectors to predict potentially buggy thread interleavings in a program. Then, in the validation phase, an active scheduler will try to exercise each of the suspicious buggy interleavings in a real execution to verify whether it is really a bug or just a false positive.

In the prediction phase, active testing tools use either static or dynamic analysis techniques to predict certain types of concurrency bugs in a program such as data races [69], atomicity violations [59, 58, 73], atomic-set serializability violations [39, 33], and deadlocks [36]. The interleaving patterns of these tools represent erroneous interleaving patterns and target certain types of concurrency bugs. Therefore, they are not generic. For a given test input, after actively testing for all the predicted buggy thread interleavings, a programmer may not be able to determine whether she should continue testing other thread interleavings for the same input or proceed to test a different input.

There are two common ways in which validation can be performed. One way is to precisely compute an alternate schedule from the observed schedule such that the computed alternate schedule is guaranteed to expose a buggy thread interleaving.

The computed alternate schedule will then be enforced during an execution [73, 33, 37]. However, computing an alternate schedule precisely is usually very expensive. It typically involves changing the relative order of independent events in an observed trace and making sure that the transformed trace is feasible. Therefore, this solution not suitable if the goal is to achieve high interleaving coverage.

The other approach is to use heuristics, usually best effort, to expose predicted interleavings [69, 59, 58, 39, 36]. For example, CTrigger [59] injects time delays at certain points to increase the chance of exposing buggy interleavings.

2.2.5 Test Input Generation

Test input generation is a testing technique that can be used to achieve high code coverage [27, 13, 28, 57]. For a given program, their goal is to generate test inputs so that testing the program with the generated test inputs can cover most of the code in the program. This technique is typically for single-threaded programs and is orthogonal to what we are doing in this dissertation.

2.3 Tolerating Concurrency Bugs

Bug avoidance and tolerating techniques have gained increasing interests recently. They serve as the last safety net to prevent concurrency bugs from manifesting in production systems. The main idea is to limit the freedom that threads can interleave so that some concurrency bugs can be avoided. Some techniques integrate dynamic bug detection tools with checkpoint and re-execution systems [62, 64]. In these tools, once a bug is detected, the program will be rolled back to a previous checkpoint and re-execute so as to bypass the buggy interleavings. Such systems have two major drawbacks. First, the bug detection tools used are optimized for reducing false positives, which could limit their capabilities on avoiding variety of concurrency bugs. Second, supporting checkpoint and re-execution is usually heavyweight and complex.

Atom-Aid [45] is a seminal work on tolerating concurrency bugs. It leverages the chunk-based execution model described in [14] in which the processor dynamically constructs chunks and enforces a serial order between them. The authors show that many concurrency bugs can be probabilistically avoided using such an execution model. However, it is optimized for avoiding single-variable atomicity violations. For other concurrency bugs, though it can still avoid them sometimes, it is purely by chance.

ISOLATOR [65] and ToleRace [67] detect and avoid one specific type of concurrency errors: asymmetric data races. An asymmetric data race occurs when one thread obeys the locking discipline correctly while some other threads do not. Both systems can efficiently detect and avoid asymmetric data races by maintaining a shadow memory copy for each critical section and updating it speculatively.

Deterministic execution [17, 56] constrains the thread interleavings to provide a guarantee that any execution of a multi-threaded program would yield the same output as long as the input remains the same. In other words, they guarantee a deterministic order of all memory accesses for a given program input. This technique also reduces the freedom that threads can interleave at runtime, thereby can tolerate concurrency bugs at some degree. It could also help programmers in reproducing bugs. However, for a given input, the system chooses the deterministic order based on arbitrary program events (for example, number of retired stores). As a result, the chosen deterministic order is not going to be more correct than a random order chosen by the current systems. Therefore, a programmer would still have to test all legal interleavings to ensure concurrency bug free. This is a fundamental problem with the shared-memory multi-threaded programming model, which we seek to address in this dissertation.

CHAPTER III

Encoding Tested Interleavings Using Interleaving Idioms

As we mentioned in Chapter I, the key of this dissertation is an efficient and effective way to encode tested interleavings. Based on that, we can either expose untested interleavings during testing (Chapter IV), or avoid untested interleavings at runtime (Chapter V and Chapter VI).

A thread interleaving is typically defined to be the order in which the memory operations are executed by all the threads in an execution. One natural way to encode a tested interleaving is to record the partial order of the memory operations executed by all the threads in an execution. However, this encoding scheme is not suitable for our purpose. Under such an encoding scheme, an encoded tested interleaving is only meaningful for a particular test run and becomes hardly useful when a different program input is used. Therefore, for a given program, there might exist an infinite number of untested thread interleavings. If one randomly picks an untested interleaving to test, the probability of exposing a concurrency bug is low. Avoiding untested interleavings at runtime using such an encoding scheme is also not practical as it is likely that many of the interleavings the system encounters at runtime are not tested. Finally, checking if an interleaving has been tested or not requires checking every memory operations involved in the interleaving, which is very expensive.

In this chapter, we discuss our approach for encoding tested interleavings. Our idea is to define a *finite* set of *interleaving fragments* that can possibly be exposed for a given program. The set of interleaving fragments should be able to represent an infinite number of legal thread interleavings of that program. At the same time, if one manages to expose the set of interleaving fragments, it is sufficient to expose a majority of concurrency bugs in this program. Tested interleavings are encoded using the set of interleaving fragments that are exposed during testing. We want the set of interleaving fragments to be as small as possible such that by exposing an untested interleaving fragment, the probability of finding a concurrency bug is much higher than randomly picking an untested thread interleaving from the infinite legal thread interleaving space. Furthermore, checking if an interleaving fragment is tested or not can be carried out quite efficiently.

In order to define interleaving fragments, we first discuss two hypotheses we make about concurrency bugs (Section 3.1). Based on these two hypotheses, we introduce interleaving idioms and iRoots which we use to represent interleaving fragments (Section 3.2). Then, we introduce a set of canonical interleaving idioms that we focus on (Section 3.3), and discuss the relation between interleaving idioms and concurrency bugs (Section 3.4). Finally, we present our empirical analysis on 34 documented bugs to show the effectiveness of our idiom based interleaving encoding scheme (Section 3.5).

3.1 Two Hypotheses about Concurrency Bugs

We first discuss two hypotheses we make about concurrency bugs. As we mentioned in Chapter I, these two hypotheses are the foundation of this dissertation.

The first hypothesis is called *small scope hypothesis*. It is originally proposed by Jackson and Damon [34]. A recent adoption of this hypothesis on concurrency bugs [49] states that most concurrency bugs can be exposed using a small number of preemptions. CHESS [52] exploits this observation to bound the number of preemptions to reduce the search space. Our adoption of this hypothesis is that we focus on those simple concurrency bugs that involve no more than two threads, two variables and two inter-thread dependencies.

Surely there exist concurrency bugs that are more complex, and our technique is not able to handle them. However, previous study has shown that complex concurrency bugs are less likely to be exposed than simple ones [12]. Given the fact that we cannot practically expose all concurrency bugs in a program within a constant time budget, we prefer to spending more time on those concurrency bugs that are more likely to occur in production environment, which are, in order words, the less complex ones.

The second hypothesis is called *value independent hypothesis.* This hypothesis is proposed by us. It states that a majority of concurrency bugs gets triggered if the erroneous inter-thread memory dependencies are exposed, irrespective of the data values of the shared variables involved in the dependencies. This hypothesis is based on our observation of several concurrency bugs in real-world applications, which is shown in Section 3.5. In fact, this hypothesis is implicitly assumed by many of the previous work on detecting, exposing and tolerating concurrency bugs [43, 44].

3.2 Interleaving Idiom

Based on these two hypotheses, we introduce interleaving idioms and iRoots which we use to represent interleaving fragments. An *interleaving idiom* is a pattern of inter-thread dependencies and the associated memory operations. An inter-thread memory dependency (denoted using $\Rightarrow$) is an immediate (read-write or write-write) dependency between two memory accesses in two threads. A memory access could be either to a data or a synchronization variable. For example, the simplest interleaving idiom is shown in the dotted box in Figure 3.2. The arrow in the idiom represents an inter-thread memory dependency. A_X and B_X represent two conflicting memory accesses where A and B are static instruction addresses.

A dynamic instance of an idiom in a program's execution is called as an *interleaving root* (iRoot). A memory access in an iRoot is represented using the static address of the memory instruction. We say that an iRoot is exposed in an execution if any part of the thread interleaving in that execution satisfies the interleaving pattern specified by the interleaving idiom and the involving static memory instructions match that specified in the iRoot. For example, consider the execution shown in Figure 3.2. For the given idiom shown in the dotted box, three iRoots of that idiom are exposed in this execution: $I_1 \Rightarrow I_4$, $I_2 \Rightarrow I_5$ and $I_5 \Rightarrow I_3$. We propose to use iRoots to represent interleaving fragments. Tested interleavings can therefore be encoded using the set of iRoots that are exposed during test runs.

Notice that the definitions of interleaving idiom and iRoot are oblivious to the data values of the shared variables involved in the inter-thread dependencies. In order words, if an iRoot has been already exposed during an earlier execution for some test input, we say this iRoot is exposed and we will not seek to expose the same iRoot again even for a different test input. This is driven by the value independence hypothesis we just discussed. Moreover, for a given set of interleaving idioms, the set of possible iRoots of those idioms in a given program is finite, which satisfies our needs for interleaving fragments.

3.3 Canonical Idioms

As we discussed above, interleaving idioms should be generic enough such that, by exposing their iRoots, most concurrency bugs could be triggered. At the same time, the set of iRoots that needs to be tested, called coverage domain, should be small enough that, the probability of exposing an unknown concurrency bug is high when an untested iRoot is exposed. To meet these competing demands, we make an assumption that most concurrency bugs can be exposed using simple thread interleaving patterns. This assumption is inspired by the small scope hypothesis just discussed.

We study a set of canonical idioms that can be constructed for one or two inter-thread dependencies (which implies there can be only one or two shared-variables) involving no more than two threads. Figure 3.1 enumerates the canonical set of idioms for two inter-thread dependencies and two threads. There are six idioms in total. We refer to idiom1 as a simple idiom, and the rest as compound idioms. For compound idioms, to reduce the coverage domain without significantly compromising the ability to expose a concurrency bug, we include two additional constraints. First, the number of instructions executed between two events in the same thread should be less than a threshold. We refer to this threshold as the vulnerability window (vw). Second, in an idiom, if atomicity of two memory accesses in a thread T to a variable V is violated by accesses in another thread, we disallow accesses to V between those two accesses in the thread T. For example, in idiom3 we do not allow any access to the variable X between the two memory accesses A_X and D_X, but there could be accesses to X between B_X and C_X.

Six idioms in Figure 3.1 can represent interleavings required to expose a majority

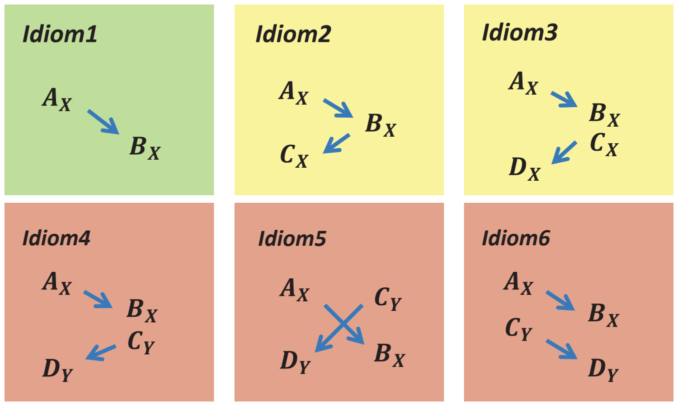


Figure 3.1: The canonical idioms for two inter-thread dependencies and two threads.

of concurrency bugs: atomicity violations, including both single variable (idiom1, idiom2, idiom3) and multi-variable (idiom4, idiom5); typical deadlock bugs (idiom5), and generic order related concurrency bugs (idiom1, idiom6). These interleaving patterns are more general than the anomalous patterns used in prior studies to find specific classes of concurrency bugs [69, 59, 73].

3.4 Relation with Concurrency Bugs

The iRoot of a concurrency bug specifies the minimum set of inter-thread dependencies and the associated memory or synchronization accesses, which if satisfied, are sufficient to trigger that bug in an execution. Therefore, for a given concurrency bug, we can either seek to expose its iRoot during testing to find it, or try to avoid its iRoot at runtime to tolerate it. Of course, higher order iRoots may also expose the same concurrency bug, but for the purpose of classifying concurrency bugs, we consider the iRoot that provides the minimum set of sufficient interleaving conditions.

Figure 3.2 shows an example of a concurrency bug. The idiom of the bug is shown in the dotted box. A and B represent static instructions in the program and X represents a memory location. The arrows denote inter-thread dependencies. The

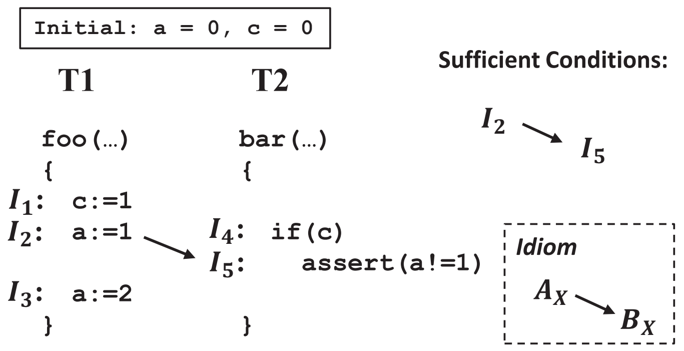


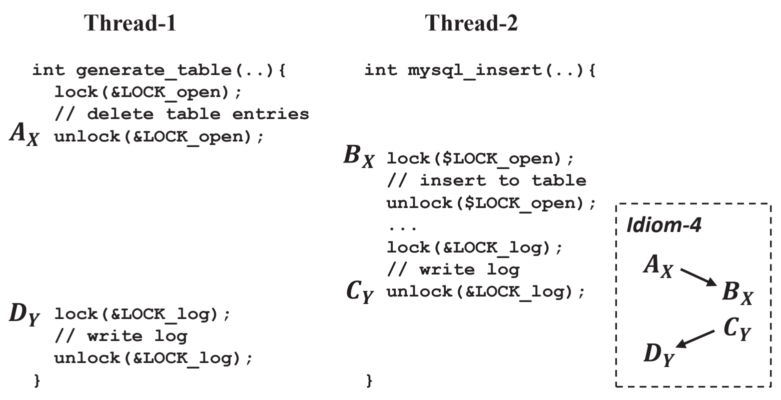
Figure 3.2: An idiom1 concurrency bug.

bug is triggered whenever the inter-thread dependency $I_2 \Rightarrow I_5$ is satisfied in an execution. Therefore, this is an idiom1 bug and its iRoot is $I_2 \Rightarrow I_5$ which we refer to as *idiom-conditions*.

Note that there exists an inter-thread dependency $I_1 \Rightarrow I_4$ that must also be satisfied before the iRoot $I_2 \Rightarrow I_5$ can be exposed. This dependency affects the control flow of the thread $T2$ and determines whether $I5$ is executed or not. We refer to such necessary conditions which must be satisfied in order to satisfy the idiom-conditions as *pre-conditions*. We choose not to include pre-conditions into the iRoot of a concurrency bug because these conditions might be irrelevant to the root cause of the bug and can typically be derived automatically based on the idiom-conditions [37].

Also notice that $I_5 \Rightarrow I_3$ will be exposed if the bug is triggered. However, this condition need not be part of the bug's iRoot ($I_2 \Rightarrow I_5$), because it is always implied by the bug's iRoot interleaving conditions.

Figure 3.3 shows a real concurrency bug in MySQL and its idiom. In this example, two critical sections in Thread-1 are expected to execute atomically, but the programmer did not enforce that constraint explicitly. The bug will be exposed when the critical sections in Thread-1 are intercepted with the critical section in Thread-2.



```
int generate_table(..){
  lock(&LOCK_open);
  // delete table entries
  unlock(&LOCK_open);

  lock(&LOCK_log);
  // write log
  unlock(&LOCK_log);
}
```

```
int mysql_insert(..){
  lock($LOCK_open);
  // insert to table
  unlock($LOCK_open);
  ...
  lock(&LOCK_log);
  // write log
  unlock(&LOCK_log);
}
```

Figure 3.3: A real idiom4 concurrency bug from MySQL.

The iRoot for this bug is of type idiom4 consisting of the two inter-thread dependencies between the lock and unlock operations. This example conveys an important observation that even if a concurrency bug is fairly complex involving many different variables and inter-thread dependencies, the iRoot of that bug (minimum set of interleaving conditions that need to be satisfied to trigger that bug) could be quite simple. Thus, by testing iRoots for a small set of idioms, we can hope to expose a significant fraction of concurrency bugs.

3.5 Empirical Analysis

To verify our hypothesis that exposing untested iRoots for our simple set of interleaving idioms could expose a significant fraction of concurrency bugs, we conducted an empirical study using 34 documented concurrency bugs from various programs including Apache, MySQL, and Memcached [1]. Here, when we say a concurrency bug, we mean a *static* concurrency bug.

Table 3.5 lists all the bugs we have analyzed. Each bug in the table has a unique name (Column-1) which is referred to throughout this dissertation. Column-2 provides a reference for each bug, which either specifies the issue number in the corresponding bug database if exists, or the paper in which the bug is studied. Among

Bug Name	Ref.	Category	Idiom	Description
Apache-1	25520	Real	Idiom-1	A data race in `ap_buffered_log_writer`. The function can be invoked by multiple threads and no synchronization is used to protect the shared memory accesses. This will cause garbage data in the access log.
Apache-2	21285	Real	Idiom-3	An atomicity violation in `mod_mem_cache.c`. A temporary object is added to a cache and will later be removed from the cache. The object is expected, but not guaranteed, to be in the cache when the removal happens, which can lead to a possible crash.
Apache-3	21287	Real	Idiom-1	An atomicity violation in `mod_mem_cache.c`. `apr_atomic_dec` is not atomic when it is complied on some platforms, which might cause an object to be cleared twice.
Apache-4	45605	Real	Idiom-4	This bug is due to the lack of atomicity between the update of variable `queue_info->idlers` and the notification of conditional variable `wait_for_idler` in `ap_queue_info_set_idle`.
MySQL-1	169	Real	Idiom-4	An multi-variable atomicity violation in `sql_delete.cc`. When deleting a table, two critical sections in `generate_table()` are interleaved with two critical sections in `mysql_insert()`, resulting in an unexpected order in the log file.
MySQL-2	644	Real	Idiom-4	A bug involves `join_init_cache` in `sql_select.cc`. The shared accesses in `join_init_cache` are expected to be atomic, but not enforced. If these accesses are interleaved with accesses in `Item_field::fix_fields`, the server will crash.
MySQL-3	791	Real	Idiom-1	The bug is triggered when a user simultaneously rotates the log and performs a table insertion. A log entry will be lost if `mysql_insert` performs the check on the variable `log_type` while the log file is temporarily closed.
MySQL-4	2011	Real	Idiom-4	An multi-variable atomicity violation in `slave.cc`. The bug is triggered when a thread rotates the log while another thread is reading the log by invoking `next_event`.
MySQL-5	3596	Real	Idiom-1	A data race involves accesses in `innobase_mysql_print_thd` and `do_command`. The variable `thd->proc_info` is unexpectedly nullified by a remote thread after it is checked and before it is used.
MySQL-6	12228	Real	Idiom-3	An atomicity violation in the procedure handling logic. A thread invalidates the procedure cache by calling `sp_drop_procedure` while another thread is executing a stored procedure, leading to a crash.
MySQL-7	12848	Real	Idiom-1	A data race in `sql_cache.cc`. A temporary value of `query_cache_size` in `Query_cache::resize()` is read by a remote thread, leading to a crash.
Pbzip2	[1]	Real	Idiom-1	The main thread waits for the output thread to finish and then releases the mutexes. If there exist multiple consumer threads, they can still be running while the main thread is releasing the mutexes and so get a SIGSEGV.
Memcached	127	Real	N/A	The bug happens when two clients concurrently increment or decrement cached data. The in-place increment/decrement is not atomic, which might cause some updates being lost.
Transmission	[42]	Real	Idiom-1	A race condition on variable `bandwidth`. The bug is triggered when a remote thread reads the variable before it is initialized in function `tr_sessionInitFull`, leading to an assertion failure.
Cherokee	[1]	Real	Idiom-1	A race condition in function `cheroke_buffer_add`. Two threads can simultaneously call this function and no synchronization is used to protect the shared accesses. This could cause a potential corrupted log.
Aget-1	[1]	Real	Idiom-1	A lock is still hold by the signal handling thread when it is canceled by the main thread. The main thread will try to acquire the same lock later, causing the execution to freeze.
Aget-2	[1]	Real	Idiom-4	When a user types ctrl-c on the console, `save_log` will be called in a separate thread to save the downloaded data. The atomicity of this function is not enforced, which could cause a potential loss of downloaded data.
MySQL-8	15530	Extract	Idiom-1	A bug usually occurs when MySQL starts. Due to an unintentional interleaving, an uninitialized value is read by a thread, resulting in all data nodes becoming a master node.
Mozilla-1	[43]	Extract	Idiom-2	An atomicity violation bug in Mozilla [43]. While one thread is loading a script and compiling it, the other thread nullifies the script. This leads to a program crash.
Mozilla-2	342577	Extract	Idiom-1	An atomicity violation bug in `nsZipArchive.cpp`. Two different threads call `SeekToItem()` simultaneously, leading to one piece of code, which is supposed to be executed only once, get executed twice. One thread will then read garbage data.
Mozilla-3	200119	Extract	Idiom-1	A bug in `nsNSSComponent.cpp`. While closing Mozilla, two threads check and free a timer object simultaneously. If two threads interleave incorrectly, the object will get freed twice, leading to a crash.
Mozilla-4	52111	Extract	Idiom-1	A race between accesses in `nsFileTransport.cpp` and `nsAsyncStreamListener.cpp`. There is one temporary state which should be invisible to other threads. Under a bad interleaving, this temporary state can be read by another thread, leading to a deadlock.
Mozilla-5	[42]	Extract	Idiom-1	An order violation in `nsthread.cpp`. It is possible that a thread reads a location that is not initialized yet, causing Mozilla to crash.
Mozilla-6	[42]	Extract	Idiom-1	A race between `macio.c` and `macthr.c`. A callback function is expected to be invoked after a thread writes to a variable, but it is not synchronized properly. This bug will cause a deadlock.
Mozilla-7	190106	Extract	Idiom-1	An order violation in `nsImapProtocol.cpp`. An event will be ignored when certain interleaving occurs. This will cause Mozilla to hang.
Mozilla-8	90196	Extract	Idiom-1	An order violation in `nsHttpdConnection.cpp`. `OnHeadersAvailable()` is expected to be called after `AsyncWrite()` returns, but it is not enforced. It will crash Mozilla.
Mozilla-9	106009	Extract	Idiom-1	A bug in `TimerThread.cpp`. It happens when `Shutdown()` is executed prior to `Run()`. In that case, the system will lose the exit event and enter into a freezing state.
Mozilla-10	[42]	Extract	Idiom-4	An atomicity violation that involves multiple variables. An inconsistent state is observed by a remote thread, causing corrupted memory.
OpenLDAP	[79]	Extract	Idiom-5	A deadlock bug in `back-bdb/cache.c`. Two threads try to acquire two locks, `lru_mutex` and `c_rwlock`, in opposite orders, leading to a deadlock.
StringBuffer	[24]	Extract	Idiom-4	The java.lang.StringBuffer overow bug. On append, not all required locks are held. Another thread may change buffer during append. State becomes inconsistent.
Pfscan	[84]	Injected	Idiom-1	A counter, which specifies the number of remaining threads, is used by the main thread to wait until all the child threads finish execution. The bug occurs any child thread finishes execution before the counter is initialized in the main thread.
BankAccount	[45]	Synthetic	Idiom-2	Shared bank account data structure bug. Simultaneous withdrawal and deposit with incorrectly synchronized program may lead to inconsistent final balance.
CircularList	[45]	Synthetic	Idiom-3	Shared work list data ordering bug. Removing, processing, and adding work units to list non-atomically may reorder work units in list.
LogProcSweep	[45]	Synthetic	Idiom-1	Shared data structure NULL dereference bug. Threads inconsistently manipulate shared log. One thread sets log pointer to NULL, another reads it and crashes.

Table 3.1: Brief descriptions about concurrency bugs that we used in our studies.

Idiom1	Idiom2	Idiom3	Idiom4	Idiom5	Idiom6	Other
20	2	3	7	1	0	1

Table 3.2: Empirical study on 34 documented bugs.

all the 34 concurrency bugs, 17 of them are real bugs for which we use the original programs and study their real executions, 13 of them are extracted bugs for which we analyze the buggy code snippets extracted from the real buggy programs, 3 of them are synthetic bugs and 1 of them is an injected bug. For each bug, Column-4 shows its idiom classification, which is to the best of our understanding and interpretation of the bug. Column-4 presents a short summary for each bug.

Table 3.2 summarizes the results. Except one, the remaining 33 concurrency bugs can be characterized using one of our interleaving idioms. We could not represent one bug using any of our idioms (Memcached) because it was value dependent. We did not find any concurrency bug that can be classified as idiom6.

CHAPTER IV

Exposing Untested Interleavings: Maple

Chapter III presents a way to encode tested interleavings using interleaving idioms and iRoots. Based on the new interleaving encoding scheme, in this chapter, we propose a new testing tool, called Maple, that actively seeks to expose untested interleavings to expose concurrency bugs. We first provide an overview about Maple in Section 4.1. Then, we discuss the three major components in Maple: the online profiler (Section 4.2), the active scheduler (Section 4.3) and the memoization database (Section 4.4). Finally, in Section 4.5, we show our evaluation results and discuss our experience in using Maple to expose known and unknown bugs in open source applications.

4.1 Overview

Testing a shared memory multi-thread program and exposing concurrency bugs is a hard problem. For most concurrency bugs, the thread interleavings that can expose them manifest only rarely during an unperturbed execution. Even if a programmer manages to construct a test input that can trigger a concurrency bug, it is often difficult to expose the infrequently occuring buggy thread interleaving, because there can be many correct interleavings for that input.

As discussed in Chapter I, existing testing techniques for exposing concurrency bugs still have their problems. Stress testing tends to explore similar thread interleavings and has less likelihood of exposing a rare buggy interleaving. Systematic testing, on the other hand, is guaranteed to expose a buggy interleaving but suffers from the scalability problem. Some recently emerged active testing tools use approximate bug detectors such as static data race detectors [19, 77] to predict buggy thread interleavings. Using a test input, an active scheduler would try to exercise a suspected buggy thread interleaving in a real execution and produce a failed test run to validate that the suspected bug is a true positive. Active testing tools target specific bug types such as data races [69] or atomicity violations [59, 58, 73, 39, 33], and therefore are not generic. For a given test input, after actively testing for all the predicted buggy thread interleavings, a programmer may not be able to determine whether she should continue testing other thread interleavings for the same input or proceed to test a different input.

In this chapter, we discuss a tool called Maple that employs a coverage driven approach for testing multi-threaded programs. Maple encodes tested interelavings using the idiom based approach discussed in Chapter III. The coverage domain is then defined to be the set of iRoots that needs to be tested. The goal of Maple is to achieve higher coverage by exposing as many untested iRoots as possible. To this end, we built the Maple testing infrastructure comprised of an online profiler and an active scheduler shown in Figure 4.1.

Maple's online profiler examines an execution for a test input, and predicts the set of candidate iRoots that are feasible for that input but have not yet been exposed in any prior test runs. Predicted untested iRoots are given as input to Maple's active scheduler. The active scheduler takes the test input and orchestrates the thread interleaving to realize the predicted iRoot in an actual execution using a set of novel

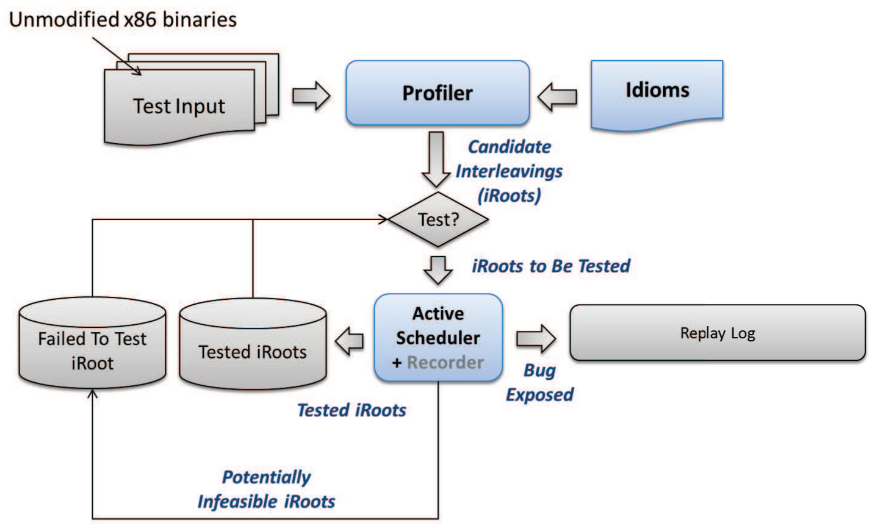


Figure 4.1: Overview of the framework.

heuristics. If the iRoot gets successfully exposed, then it is memoized by storing it in a database of iRoots tested for the program. We also consider the possibility that certain iRoots may never be feasible for any input. We progressively learn these iRoots and store them in a separate database. These iRoots are given a lower priority when there is only limited time available for testing.

When the active scheduler for an iRoot triggers a concurrency bug causing the program produces an incorrect result, Maple generates a bug report that contains the iRoot. Our active scheduler orchestrates thread schedules on a uniprocessor, and therefore recording the order of thread schedule along with other non-deterministic system input, if any, could allow a programmer to reproduce the failed execution exposed by Maple.

Maple seeks to achieve higher coverage by exposing as many different iRoots as possible during testing. Unlike coverage metrics such as basic block coverage, it is hard to estimate the total number of iRoots for a given program. However, number of exposed iRoots can be used as coverage metric for a saturation-based test adequacy [70, 38]. That is, a programmer can decide to stop testing at a point when additional tests are unlikely to expose new iRoots. We believe saturation-based testing approach is a practical solution for problems such as concurrent testing where estimating the coverage domain is intractable.

We envision two usage models for Maple. One usage scenario is when a programmer has a test input and wants to test her program with it. In this scenario, Maple will help the programmer actively expose thread interleavings that were not tested in the past. Also, a programmer can determine how long to test for an input, because Maple's predictor would produce a finite number of iRoots for testing.

Another usage scenario is when a programmer accidentally exposed a bug for some input, but is unable to reproduce the failed execution. A programmer could use Maple with the bug triggering input to quickly expose the buggy interleaving. We helped a developer at Intel in a similar situation to expose an unknown bug using Maple.

We built a dynamic analysis framework using PIN [46] for analyzing concurrent programs. Using this framework, we built several concurrency testing tools including Maple, a systematic testing tool called CHESS [52] and tools such as PCT [12] that rely on randomized thread schedulers, which we compare in our experiments.

We perform several experiments using open-source applications (Apache, MySQL, Memcached, etc.). Though Maple does not provide hard guarantees similar to CHESS [49] and PCT [12], it is effective in achieving higher coverage faster than those tools in practice. We evaluate 13 documented bugs and show that Maple is able to expose 11 of them faster than these prior methods, which provides evidence that achieving higher coverage for our metric based on interleaving idioms is effective in exposing concurrency bugs. We also discuss our experiences in using Maple to find 3 unknown bugs in `aget`, `glibc`, and `CNC`.

Our dynamic analysis framework for concurrent programs and all the testing tools we developed are made available to the public under the Apache 2.0 license. They can be downloaded from (`https://github.com/jieyu/maple`).

4.2 Online Profiling For Predicting iRoots

In this section, we discuss the design and implementation of Maple's online profiler. Given a program and a test input, the profiler predicts a set of candidate iRoots that can be tested for the given test input.

4.2.1 Notations and Terminology

As we discussed in Section 3.2, an iRoot for an idiom comprises of a set of inter-thread dependencies between memory accesses. A memory access could be either a data access or a synchronization access. For synchronization accesses, we only consider lock and unlock operations. A lock or an unlock operation is treated as a single access when we construct an iRoot, and all memory accesses executed within lock and unlock functions are ignored.

T_i (where i = 1,2,3,...) uniquely identifies a thread. A_X^i represents a dynamic memory access. The super script i uniquely identifies a dynamic access, but we usually omit it in our discussion to improve readability. If A_X^i is a data access, A stands for the static address of the memory instruction and X refers to the memory location accessed. Two data accesses are said to conflict if both access the same memory location and at least one of them is a write. If A_X^i is a lock/unlock synchronization access, A stands for the address of the instruction that invoked the lock/unlock operation and X refers to the lock variable. Two synchronizations accesses conflict if one of them is a lock and the other is an unlock operation.

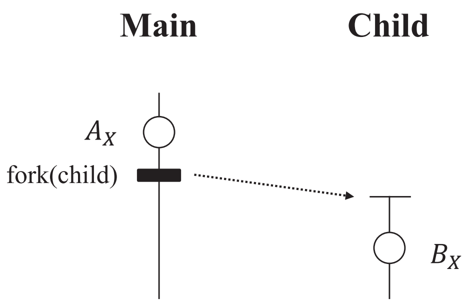


Figure 4.2: Infeasible iRoots due to non-mutex happens-before relations.

4.2.2 Naive Approach

We start with the simpler problem, which is predicting idiom1 iRoots for a given test input. One naive way to predict idiom1 iRoots is as follows. First, run the program once and obtain the set of accesses executed by each thread. Then, for any access A_X in thread T_c, if there exists a conflicting access B_X in another thread T_r, we predict two idiom1 iRoots: $A \Rightarrow B$ and $B \Rightarrow A$. For synchronization accesses, we predict $A \Rightarrow B$ only if A is an unlock operation and B is a lock operation. Obviously, this approach could produce many infeasible iRoots. An iRoot is said to be infeasible for a program if it can never manifest in any legal execution of that program. In the following sections, we discuss two major sources of inaccuracy in this naive algorithm and present our solutions.

4.2.3 Non-Mutex Happens-Before Analysis

The profiler predicts iRoots based on a few profiled executions for a given test input. The predicted iRoots may not appear in any of the profiled executions, but they are predicted to be realizable in some other legal executions. We should avoid predicting iRoots that can never manifest in any of the legal executions.

We observe that some of the happens-before relations tend to remain the same in all of the legal executions. Therefore, these happens-before relations can be used to filter out infeasible iRoots predicted in the naive approach. For example, in Figure 4.2, an access A_X is executed before the main thread forks the child thread. B_X is executed in the child thread. Assuming that A_X and B_X are conflicting, the naive approach will predict two idiom1 iRoots: $A \Rightarrow B$ and $B \Rightarrow A$. However, it is trivial to observe that in any legal execution, A_X executes before B_X because of the fork call. As a result, we should not predict the iRoot $B \Rightarrow A$ as a candidate to test.

We improve the accuracy of our profiler by exploiting the observation that non-mutex happens-before relations mostly remain the same across different executions for a given input. A non-mutex happens-before relation is due to any synchronization operation other than a lock/unlock. Happens-before relations due to locks tend to change across executions, because the order in which locks are acquired could easily change. On the contrary, we find that non-mutex happens-before relations (e.g. fork-join, barrier and signal-wait) are more likely to remain constant across different executions. Therefore, the profiler predicts an iRoot only if it does not violate the non-mutex happens-before relations in at least one of the profiled executions. For the program in Figure 4.2, B_X cannot happen before A_X any of the executions according to the non-mutex happens-before relation due to fork. As a result, the profiler will not predict $B \Rightarrow A$ as a candidate iRoot to test. Though effective in pruning infeasible iRoots, this analysis is not sound because some non-mutex happens-before relations are not guaranteed to remain constant across different executions for an input.

4.2.4 Mutual Exclusion Analysis

Mutual exclusion constraints imposed by locks could also prevent naively predicted iRoots from manifesting in any of the alternate executions. For example, in Figure 4.3, all the accesses (A_X, B_X and C_X) are protected by the same lock m. Assume that these accesses conflict with each other. The naive approach would predict

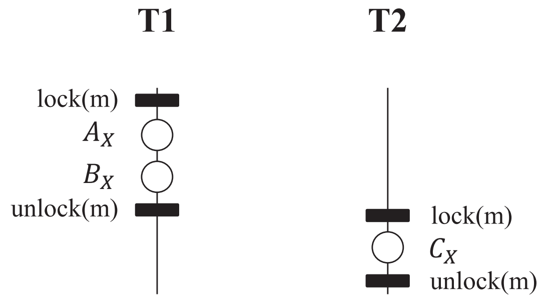


Figure 4.3: Infeasible iRoots due to mutual exclusion.

$A \Rightarrow C$ (and $C \Rightarrow B$) to be a candidate iRoot to test. Clearly, $A \Rightarrow C$ (and $C \Rightarrow B$) is not feasible because of the mutual exclusion constraint imposed by the lock m.

To further improve its accuracy, the profiler is augmented with a mutual exclusion analysis phase to filter those infeasible iRoots that are caused by the mutual exclusion constraints. To achieve this, the profiler needs to collect two types of information for each access A_X. One is the lockset information which contains the set of locks that are held by $Thd(A_X)$ when executing A_X. The other is the critical section information which specifies whether A_X is the first or the last access to X in the critical section that contain A_X.

We now use an example to illustrate how these two types of information can be used to filter infeasible iRoots caused by the mutual exclusion constraints. Consider the example in Figure 4.3. The profiler needs to decide whether iRoot $A \Rightarrow C$ is feasible. It first checks the locksets of both accesses: A_X and C_X. If the locksets are disjoint, the profiler will immediately predict the iRoot to be feasible. If not, the profiler will go to the next step. In this example, A_X and C_X have the same lockset $\{m\}$. Therefore, the profiler proceeds to the next step. In the second step, for each common lock (in our example its m), the profiler checks whether the mutual exclusion constraint imposed by the common lock will prevent the iRoot from manifesting. It checks whether A_X is the last access to X in the critical section that is guarded by the common lock m, and whether C_X is the first access to X in the critical section that is guarded by the common lock m. If either of them is not true, the profiler will predict that the iRoot is infeasible. In our example, since B_X is the last access to X in the critical section that is guarded by the common lock m, the iRoot $A \Rightarrow C$ is predicted to be infeasible. This analysis is also not sound since control flow differences between executions could affect our analysis, but it works well in practice.

4.2.5 Online Profiling Algorithm

Our profiler predicts candidate iRoots to test for a particular idiom using an online mechanism that we describe in detail in this section. An online algorithm avoids the need to collect large traces.

Baseline Algorithm

The profiler monitors every memory access. For each object, the profiler maintains an access history for each thread. We use $AH_X(T_i)$ to denote the access history for object X and thread T_i. Each access A_X in the access history $AH_X(T_i)$ is ordered by the execution order of T_i, and is associated with a vector clock and an annotated lockset. The vector clock, denoted as $VC(A_X)$, is used to perform the non-mutex happens-before analysis. It is the same as that used in many of the dynamic data race detectors, except that here we consider non-mutex happens-before relations. The annotated lockset, denoted as $AnnoLS(A_X)$, is used to perform the mutual exclusion analysis. It consists of a set of locks, each of which is annotated with a sequence number and two bits. The sequence number is used to uniquely identify each critical section guarded by the lock, and the two bits indicate whether the access is the first or the last access in the corresponding critical section. We say that two annotated locksets are disjoint if no common lock is found between the two sets. Both the vector clock and the annotated lockset are recorded when $Thd(A_X)$ is executing A_X.

When an access A_X is being executed by T_c, the profiler checks the access histories from all other threads on object X (i.e. $AH_X(T_r), T_r \neq T_c$). If there exists a conflicting access B_X in $AH_X(T_r)$, the profiler will predict the iRoot $B \Rightarrow A$ if the following conditions are true: (1) B_X does not happen after A_X by checking $VC(B_X)$ and $VC(A_X)$ (the non-mutex happens-before check). (2) Either $AnnoLS(A_X)$ and $AnnoLS(B_X)$ are disjoint, or for each common lock m held by A_X and B_X, A_X is the first access to X in the corresponding critical section guarded by m and B_X is the last access to X in the corresponding critical section guarded by m (the mutual exclusion check). Similarly, the profiler will also predict the iRoot $A \Rightarrow B$ according to the above rules.

To make the profiling algorithm online, we need to deal with several issues. One issue is that when A_X executes, some access, say C_X, has not been performed yet. As a result, C_X will not be in any access history. However, the profiler will still correctly predict iRoot $A \Rightarrow C$ and iRoot $C \Rightarrow A$ at the time C_X is executed if they are feasible. Another issue with the online algorithm is that when executes A_X, the profiler cannot precisely compute the annotated lockset $AnnoLS(A_X)$ required by the mutual exclusion analysis. The reason is because it does not know whether the access A_X will be the last access in the current critical section or not. We solve this issue by delaying predicting iRoots for A_X until either of the following events happens: (1) another access to X is reached by $Thd(A_X)$. (2) X is about to be deallocated (e.g. free()). (3) $Thd(A_X)$ is about to exit. The insight here is that the profiler can precisely compute the annotated lockset for A_X if any of the above events happens.

Optimizations

We have designed a few optimizations to make the online algorithm practical for large applications.

- *Condensing access histories.* An access history can be very large because it stores information about all the dynamic accesses to an object from a given thread. This can cause problems both in terms of analysis time and space. To solve that, we propose to condense each access history by removing duplicate entries in it. We say two accesses from the same thread are *identical* if they have the same static instruction address, vector clock and annotated lockset. We find that storing two identical accesses in an access history is not necessary because these two accesses will always lead to the same iRoot during prediction. Therefore, every time a new access is about to be added into an access history, we first check if an identical entry already exists in the access history. If yes, we choose not not update the access history. This check can be carried out efficiently using a hash table.
- *Caching prediction results.* As we discussed above, when an access A_X is being executed, the profiler needs to scan the access history of each remote thread on object X. We call this operation a *full scan*. A full scan is a time consuming operations. However, we observe that it is not always necessary to perform a full scan. If an identical copy of A_X is found in the access history as we just described, the profiler can safely skip the full scan as no new iRoot will be predicted even if a full scan is performed.
- *Removing useless access history entries.* We observe that it is not necessary to keep all access histories from the beginning of the program execution. If an

access in the access history can no longer be part of any potentially predicted iRoot, we can safely remove it from the access history.

- *Monitoring only shared instructions.* Maintaining access histories for each memory location is very expensive. Clearly, the profiler does not need to maintain access histories for thread private locations. We perform an online analysis that runs concurrently with the profiler to detect those instructions that can access shared locations. Such instructions are called *shared instructions*. The profiler uses this information to create access histories only for those locations that are accessed by shared instructions. To discover shared instructions, we maintain meta data for each memory location in which we store information about those instructions and threads that have accessed that location. Whenever we discover a shared location (accessed by multiple threads), we mark all instructions that access that location as shared instructions.

4.2.6 Predicting iRoots for Compound Idioms

To predict iRoots for compound idioms, we designed an algorithm that leverages the idiom1 prediction results. The approach is generic to all compound idioms defined in Section 3.2. The algorithm is divided into two parts: identifying local pairs, and correlating with idiom1 prediction results. In this section, we discuss these two parts in detail.

Identifying Local Pairs

A local pair, as suggested by its name, is a pair of accesses from the same thread. During a profiling execution, if the profiler finds two accesses A_X and B_Y (X may or may equal to Y) such that A_X and B_Y are from the same thread, A_X is executed before B_Y, and the number of dynamic instructions executed between A_X and B_Y is

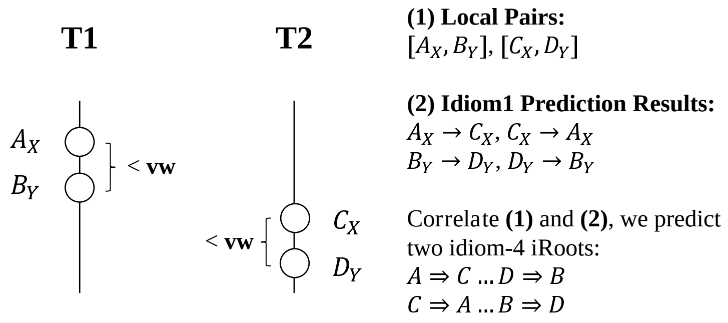

T1
T2
A_X
B_Y
< vw
< vw
C_X
D_Y
(1) Local Pairs:
$[A_X, B_Y]$, $[C_X, D_Y]$
(2) Idiom1 Prediction Results:
$A_X \rightarrow C_X$, $C_X \rightarrow A_X$
$B_Y \rightarrow D_Y$, $D_Y \rightarrow B_Y$
Correlate **(1)** and **(2)**, we predict two idiom-4 iRoots:
$A \Rightarrow C \ldots D \Rightarrow B$
$C \Rightarrow A \ldots B \Rightarrow D$

Figure 4.4: Predicting iRoots for compound idioms.

less than a pre-set threshold vw (vw stands for vulnerability window and is specified in the idiom definition), it will record a local pair $[A_X, B_Y]$. For example, Figure 4.4 shows a profiling execution. Accesses A_X and B_Y in T_1 are executed first, followed by C_X and D_Y in T_2. The profiler records two local pairs from this profiling execution: $[A_X, B_Y]$ and $[C_X, D_Y]$. To collect local pairs, the profiler uses a rolling window for each thread to keep track of the recent accesses.

Correlating with Idiom1 Prediction Results

To predict iRoots for compound idioms, we propose to leverage the idiom1 prediction results. We use an example to illustrate how to correlate local pairs with idiom1 prediction results to predict compound idiom iRoots. Consider the example shown in Figure 4.4. As mentioned, the profiler identifies two local pairs: $[A_X, B_Y]$ and $[C_X, D_Y]$. Meanwhile, the profiler also records the idiom1 prediction results. For instance, A_X and C_X can produce two idiom1 iRoots $A \Rightarrow C$ and $C \Rightarrow A$ according to the idiom1 prediction algorithm, therefore the profiler records both $A_X \rightarrow C_X$ and $C_X \rightarrow A_X$ in the idiom1 prediction results [1]. Similarly, the profiler records $B_Y \rightarrow D_Y$ and $D_Y \rightarrow B_Y$. Now, consider the first local pair $[A_X, B_Y]$. According to the predicted idiom1 results, C_X can potentially depend on A_X, and B_Y can potentially

[1]Notice that the idiom1 prediction results are only useful for the current profiling execution, and will be discarded once the execution finishes. They are different from the predicted idiom1 iRoots which last across executions. They contain more information than idiom1 iRoots do.

depends on D_Y. As a result, the profiler predicts an idiom4 iRoot $A \Rightarrow C...D \Rightarrow B$ (assume $X \neq Y$). Similarly, for another local pair $[C_X, D_Y]$, the profiler predicts another idiom4 iRoot $C \Rightarrow A...B \Rightarrow D$. Currently, the profiler performs the correlation part at the end of each profiling execution. Similar optimization technique is used to condense local pairs, that is if two local pairs from the same thread have both their accesses identical, the profiler just records one of them.

4.3 Actively Testing Predicted iRoots

In this section, we discuss the design and implementation of Maple's active scheduler. Maple's profiler predicts a set of iRoots that can be realized in an execution using a test input. The goal of Maple's active scheduler is to validate the prediction by orchestrating the thread schedule to realize the predicted iRoots in an actual execution for the test input.

4.3.1 A Naive Approach

Suppose that we want to expose an idiom1 candidate iRoot $A \Rightarrow B$. The static instructions A and B are called candidate instructions. In a test run, there might be multiple dynamic accesses associated with a single candidate instruction. We still use A_X to denote a dynamic accesses to object X by the candidate instruction A. The naive approach works as follows. Whenever a candidate instruction (say A_X) is reached by a thread (say T_1), the active scheduler delays the execution of T_1. During the delay, if another thread (say T_2) reaches the other candidate instruction (say B_X), then the iRoot $A \Rightarrow B$ is exposed by executing A_X first and then executing B_X (as shown in Figure 4.5).

This approach is used in several prior studies (e.g. [59]). While it is simple, it can lead to several issues, including deadlocks (also referred as thrashing in [36]).

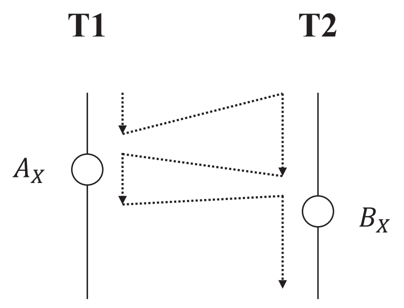


Figure 4.5: The ideal situation for exposing an idiom1 iRoot $A \Rightarrow B$.

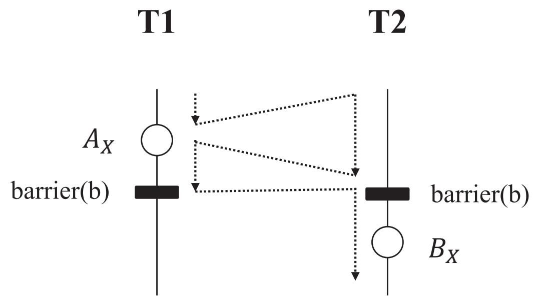


Figure 4.6: The naive approach could deadlock when exposing an idiom1 iRoot $A \Rightarrow B$.

Consider the example in Figure 4.6. Suppose that T_1 reaches A_X first. The active scheduler, in this case, delays the execution of T_1, waiting for the other candidate instruction to be reached in T_2. T_2 is blocked when calling the barrier function, leading to a deadlock because no thread can make forward progress at that state. One way to mitigate this issue is to make use of timeout. In the example, if a timeout is introduced for each delay, T_1 will eventually be woken up when the timeout has expired. However, as discussed in the following sections, this is not enough to address most of the issues.

4.3.2 Non-preemptive and Strict Priority Scheduler

There are two problems with a timeout-based approach. First, it is sensitive to the underlying environment, hence fragile [35]. For instance, the timeout should be set to a larger value when running a program on a slower machine. Second, determining how long the timeout should be is not straightforward. A large timeout is detrimental to performance due to the longer delays, while a shorter timeout could cause unnecessary give ups.

An alternative to timeout is to monitor the status of each thread (blocked or not) by instrumenting every synchronization operations and blocking system calls (e.g. [69, 58, 36]). For example, in Figure 4.6, if the active scheduler keeps track of the status of each thread, it should know that T_2 is blocked after it calls the barrier function. Thus, T_1 will be woken up immediately since no other thread in the system can make forward progress at that state.

Our approach eliminates the need for monitoring thread status. The main idea is to leverage a thread scheduler provided by the underlying OS that supports non-preemptive and strict priority. All threads are forced to run on a single processor. Each thread is assigned a non-preemptive strict priority. Under this scenario, a lower priority thread never gets executed if there exists a non-blocked higher priority thread.

By using a non-preemptive strict priority scheduler, the deadlocks will be automatically detected and resolved by the underlying OS since it knows the status of each thread. Let us consider the example in Figure 4.6. Initially, T_1 has a priority $P_{init}(T_1)$ and T_2 has a priority $P_{init}(T_2)$. Suppose that $P_{init}(T_1) > P_{init}(T_2)$. Therefore, T_1 executes first. When T_1 reaches A_X, the active scheduler changes the priority of T_1 to P_{low} such that $P_{low} < P_{init}(T_2)$. According to the semantics of a non-preemptive strict priority scheduler, T_1 is preempted by T_2 immediately after the priority change. When T_2 calls the barrier function, it is blocked. At this moment, T_1 becomes the only non-blocked thread, and resumes execution immediately after T_2 is blocked. The deadlock situation is naturally resolved. Note that the active scheduler only needs to monitor the instructions involved in the iRoot being exposed, thus limiting the runtime overhead.

Any thread scheduler that implements a non-preemptive strict priority semantics can be used for our purpose. For example, we can use the scheduler discussed in Frost [75]. However, it requires modifications to the underlying Linux kernel, making it not easily portable. Alternatively, we may choose to use the real-time scheduler (schedule policy SCHED_FIFO) provided by the Linux kernel, which is an *approximation* of a non-preemptive strict priority scheduler. In most of the time, the real-time scheduler will follow the semantics of a non-preemptive strict priority scheduler. However, in some cases where a higher priority thread is executing some I/O related operations (e.g. page swaps), the OS may choose to suspend the higher priority thread and run a lower priority thread in the meantime even if the higher priority thread is not actually blocked.

In Maple, we choose to use the real-time scheduler provided by the Linux kernel mainly for portability concern. The slight inaccuracy introduced by the real-time scheduler may cause a few false negatives. However, our results shown in Section 4.5 indicate that this effect is small. Maple is still very effective in exposing both known and unknown bugs in spite of this inaccuracy.

4.3.3 Complementary Schedules

Another problem with the approach discussed in Section 4.3.1 is that it does not have a mechanism to control the order in which threads get executed. Assume that we want to expose the idiom1 iRoot $A \Rightarrow B$ in the example of Figure 4.7, where A_X is in T_1 and B_X in T_2, respectively. Because both A_X and B_X are protected by the same lock m, if B_X is reached by T_2 first, the iRoot will not be exposed. The delay introduced before B_X will not help because T_1 will never be able to reach A_X due to the fact that T_2 is still holding the lock m. In order to expose this iRoot, A_X must be reached by T_1 first. However, the naive approach (Section 4.3.1) cannot guarantee this as it does not have a mechanism to control the order in which the threads get executed.

We address this issue using a technique called complementary schedules. The idea is to use two test runs on each candidate iRoot. Each newly created thread T_i, including the main thread, is assigned with an initial priority $P_{init}(T_i)$. In the first test run, the initial priorities are assigned from high to low, following the order of thread creation. To be more precise, we have $P_{init}(T_i) > P_{init}(T_j)$ (T_i has a higher priority than T_j) if thread T_i is created earlier than T_j. In the second test run, initial priorities are assigned from low to high, following the order of threads creation. In order words, $P_{init}(T_i) < P_{init}(T_j)$ if thread T_i is created earlier than T_j. Using this technique, we increase the likelihood that, in one of the two test runs, A_X will be reached first in the example shown in Figure 4.7.

Complementary schedules is a heuristic. If more than two threads are involved, we may encounter problems like priority inversion [12] and our current solution may not work as we expected. One possible solution is to use more test runs for each candidate iRoot and shuffle the initial priorities differently in each test run. We leave that as a future work.

4.3.4 Watch Mode Optimization

A problem with the naive approach is that it can unnecessarily give up exposing some iRoot in certain cases. Consider the example in Figure 4.7. We are still interested in exposing the idiom1 iRoot $A \Rightarrow B$. If T_1 reaches A_X first, the naive approach gives up exposing the iRoot for A_X right after the timeout. However, giving up is not necessary here because it is still possible that B_X could execute later without any access to X in between A_X and B_X.

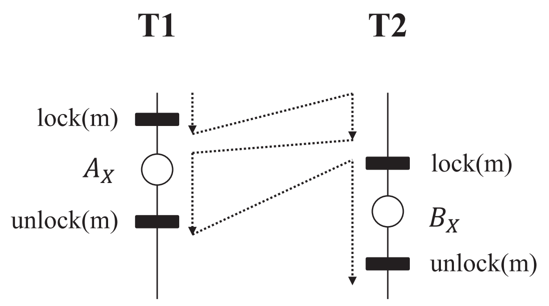


Figure 4.7: The situation in which the watch mode is turned on for exposing an idiom1 iRoot $A \Rightarrow B$.

We use a mechanism called watch mode to exposes an iRoot in such case. In watch mode, every memory access is monitored. Consider again the example in Figure 4.7. When T_1 reaches A_X first and sets its priority to P_{low} (P_{low} is lower than any initially assigned priorities), T_2 gets control and executes, but is blocked when trying to acquire the lock m. As mentioned above, T_1 resumes immediately after T_2 is blocked and executes A_X. At this moment, instead of giving up exposing iRoot for A_X, the active scheduler enters the watch mode and monitor every memory access. The active scheduler still keeps the priority of T_1 to P_{low}. Once the lock m is released, T_1 is preempted by T_2 because T_2 has a higher priority than T_1. Shortly after, T_2 reaches B_X and no access to X is found in between A_X and B_X. Therefore, the iRoot $A \Rightarrow B$ is exposed.

In the same example, during the watch mode, it is likely that T_1 reaches an instruction – no matter whether it is a candidate instruction or not – that accesses X as A_X does. In such case, the active scheduler is not able to expose iRoot for A_X because T_1 already has the lowest priority at that moment. It just ends the watch mode and gives up exposing iRoot for A_X. If the access to X (not instruction B) is from a thread other than T_1 (say T_3), the active scheduler sets the priority of T_3 to P_{low}. The intuition here is that some other threads may make progress and reach the other candidate instruction B. However, if the conflicting access is eventually executed by T_3 in spite of its lowest priority, the active scheduler ends the watch mode and gives up.

The watch mode can be implemented efficiently using debug registers or by leveraging the selective instrumentation mechanism in PIN [46]. We implement the second approach. For compound idioms, the execution under watch mode is usually short given that we have distance constraints in the idiom definitions. For idiom1, the selective instrumentation mechanism in PIN can affect performance depending on how long the active scheduler spends in watch mode. The overhead is discussed in Section 4.5.

4.3.5 Candidate Arbitration

There might exist multiple dynamic accesses that correspond to the same candidate instruction during the execution. In many cases, the active scheduler has to decide which of these accesses belongs to the candidate iRoot to expose. For example, while the active scheduler is exposing iRoot for A_X (seeking candidate instruction B in other threads), it is possible that another thread also reaches the candidate instruction A, or another thread reaches candidate instruction B, but it happens to access a location other than X (say B_Y). In either one of these situations, the active scheduler has two choices: either continue to expose the iRoot $A \Rightarrow B$ for A_X, or give up on that attempt and seek to expose the iRoot for latter access of A or for B_Y. We decided to make these choices random with equal probability. We choose not to use a fixed policy because it could cause some feasible iRoot to become impossible to expose. We save the random seed for reproduction purpose.

We aware that the random arbitration algorithm we use may cause a later access exponentially unlikely to be used as part of a candidate iRoot. This will become an issue when we do want to expose an iRoot for a later access (e.g. only the iRoot that uses this access will lead to a bug). Nevertheless, we are still able to expose all the bugs we have analyzed using this strategy (Section 4.5.2). This may be because many of the bugs we analyzed manifest early in their executions, or not many dynamic accesses exist for the candidate instructions in the iRoots that expose the bugs. Even if that, we believe this is an important problem and we plan to address it in our future work. Currently, we can think of two possible ways. First, we can associate more information with each candidate instruction such as its calling context so that some irrelevant accesses that use the same instruction but different contexts will be filtered out. Second, we can devise a more sophisticated arbitration algorithm using more test runs.

4.3.6 Dealing with Asynchronous External Events

Some applications depend on asynchronous external events such as network and asynchronous signals. These events are usually difficult to deal with in the active scheduler because it has no control on when these events are delivered. Consider the example in Figure 4.8 where T_2 has a higher priority initially. When T_2 reaches B_X, its priority is changed to P_{low}, at which point T_1 is scheduled. If T_1 is blocked when calling the function `sigwait` (e.g. because the signal might not have been delivered yet), since all threads except T_2 are blocked in the system at that time, T_2 has to execute B_X in spite of its lowest priority; thus giving up the exposition of iRoot $A \Rightarrow B$ for B_X. We observe that if the asynchronous signal in this example is delivered earlier, the active scheduler might be able to expose the iRoot.

To solve this problem, the active scheduler introduces extra time delay where it is about to give up, hoping that the potential external event will arrive during that period. For instance, in the example of Figure 4.8, when the active scheduler is about to give up by executing B_X in T_2 after T_1 has been blocked, a time delay is

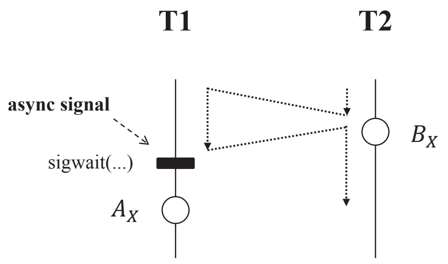


Figure 4.8: Problem with asynchronous external events.

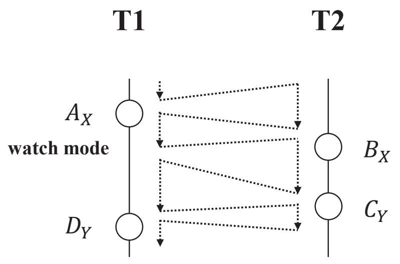


Figure 4.9: Expose a compound idiom iRoot $A \Rightarrow B ... C \Rightarrow D$.

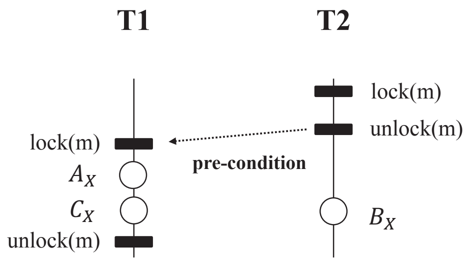


Figure 4.10: A pre-condition exists when trying to expose iRoot $A \Rightarrow B$.

injected right before B_X is executed. During this period, if the asynchronous signal is delivered, the active scheduler can successfully expose the iRoot $A \Rightarrow B$.

For applications that do not depend on asynchronous external events, there is no need for the active scheduler to inject extra time delay. We detect whether an application depends on asynchronous external events by monitoring system calls and signals during profiling. Even if an application does depend on asynchronous external events, this might not be true for all the iRoots. During profiling, we mark each candidate iRoot with a flag indicating whether this iRoot depends on asynchronous external events or not. The active scheduler uses this flag to decide whether to inject time delay or not. Finally, to ensure forward progress, we set a timeout for each delay. The timeout value can be optimized according to the application and the input.

4.3.7 Compound Idioms

A compound idiom iRoot is composed of multiple idiom1 iRoots. Our general policy for exposing compound idiom iRoots is to expose each of the idiom1 iRoot one at a time. Each of the idiom1 iRoot is exposed as described before, but the algorithm for exposing compound idiom iRoots needs to address two more issues that we describe next.

First, the active scheduler always enters the watch mode after the first candidate instruction in a compound iRoot is executed. To understand why, consider the example in Figure 4.9. The goal is to expose an idiom4 iRoot $A \Rightarrow B...C \Rightarrow D$. According to the idiom4 definition, we need to make sure that there is no other access to the same locations that A and D access in between them. Therefore, after A_X is executed in this example, we enter the watch mode. If there is any access to location X before D_Y is reached, the active scheduler stops trying to expose the iRoot for A_X because this violates the idiom definition. One complex aspect of this implementation is that, before D_Y is reached, we do not know which location it is going to access. We solve this problem by recording the set of locations that are accessed by T_1 after A_X is executed. This set is checked when D_Y is reached, to verify that none of the addresses touched conflicts with the address accessed by D_Y. In addition, the active scheduler exits the watch mode and gives up exposing iRoot for A_X if the number of dynamic instructions executed by T_1 after A_X exceeds the pre-defined threshold specified in the idiom definition.

Second, the arbitration is biased after the first part of the compound idiom iRoot is exposed. In the example of Figure 4.9, after the first part of the iRoot ($A \Rightarrow B$) is exposed, if T_2 reaches A_Z (an access to Z by candidate instruction A), we have two choices: (1) ignore A_Z and continue looking for the second part of the iRoot; (2) expose the iRoot for A_Z and discard the already exposed first part. In such a case, we select the first choice with a higher probability.

Finally, exposing iRoots for idiom5 is slightly different from other compound idioms. To expose an idiom5 iRoot $A \Rightarrow B...C \Rightarrow D$, our strategy is to let two different threads reach A and C, respectively; then execute them, and seek B and D in the corresponding threads.

4.3.8 Exposing Pre-conditions

One limitation of the current active scheduler is that it cannot handle pre-conditions. The pre-conditions for an iRoot is the necessary conditions that need to be satisfied to expose the iRoot. For example, in Figure 4.10, there exists a pre-condition (from `unlock` to `lock`) that needs to be satisfied so that the iRoot $A \Rightarrow B$ can be exposed. Currently, our active scheduler has no knowledge of these pre-conditions; therefore it cannot enforce them. The complementary schedules might alleviate this problem to some extent. To fully address this problem, one possible solution would be to automatically derive pre-conditions for a given iRoot [37]. We leave this as a future work.

4.4 Memoization of iRoots

Past work on systematic testing and active testing ignore the information about the interleavings tested from previous test runs with different inputs. We believe that this is crucial in reducing the number of interleavings that need to be tested for a given program input. Therefore, we propose a memoization module in our active testing infrastructure. The memoization module is composed of a database of tested interleavings and a database of fail-to-test interleavings for each interleaving idiom as shown in Figure 4.1. This module is used to avoid testing the same interleaving again and again across different test inputs.

The candidate interleavings are pruned out depending on whether previous attempts were made to test them. If a candidate interleaving was tested before (i.e. it has been exposed by the active scheduler), it is filtered out by consulting the tested interleavings database. This optimization is sound if the bugs we are targeting are not value dependent. Also, if several attempts were made in the past to test a candidate interleaving and the active testing system failed to produce a legal execution exposing the desired interleaving, this candidate interleaving is filtered out using the fail-to-test interleavings database, which stores all such failed to expose candidate interleavings. This allows us to avoid trying to expose thread interleavings that can never manifest. Unlike memoization of tested iRoots, this is an unsound optimization even if a bug is not value dependent. However, the number of times a candidate interleaving is actively tested is configurable, and a programmer can choose to set it to a very high value if soundness is a concern.

4.5 Evaluation

This section evaluates Maple and discusses our experiences in using the tool for testing real world applications. We first describe the configurations of Maple we used in our experiments (Section 4.5.1). Then, we compare Maple with other general concurrency testing techniques in two different usage scenarios (Section 4.5.2 and Section 4.5.3), and show how memoization can be useful in both of these scenarios (Section 4.5.2 and Section 4.5.3). Finally, we discuss the efficiency and effectiveness of Maple (Section 4.5.4).

4.5.1 Maple Configuration

Maple is built to be highly configurable. We now describe the default configurations of Maple in our experiments.

In the profiling phase, the program is profiled using the best randomized testing technique (explained later in Section 4.5.3) a few number of times. For each profile run, the profiler observes what iRoots are exposed and predicts candidate iRoots to test. The profiling phase stops when no new candidate iRoot is found for N consecutive profile runs (we use an empirical value $N = 3$ throughout our experiments). Unless otherwise noted, Maple observes and predicts all iRoots in the program by default, including those iRoots from libraries such as Glibc. We believe this is necessary because we do find bugs that are related to the library code (e.g. `Bug #11` and `Bug #13` in Table 4.1).

In the testing phase, candidate iRoots are exposed using the active scheduler. Currently, the active scheduler tests all candidate iRoots starting from idiom1 and then proceeds to test iRoots for other idioms in the order of their complexity (one to five). For each idiom, the active scheduler always chooses to test those iRoots that are from the application code first. More sophisticated ranking mechanism may exist, but we leave that to future work. Each candidate iRoot will be attempted at most twice, as mentioned in Section 4.3.3.

4.5.2 Usage Scenario 1: Exposing Bugs with Bug Triggering Inputs

One scenario that Maple can be useful is when a programmer or a user accidentally exposed a non-deterministic bug for some input, but is unable to reproduce the failed execution. In that case, the programmer can use Maple with the bug triggering input to quickly expose the buggy interleaving. Once the buggy interleaving is found, Maple can also reproduce it faithfully by replaying the same schedule choices, which can be very useful during the debugging process.

To evaluate the ability of Maple in exposing concurrency bugs in such scenarios, we want to know whether Maple is able to expose the set of bugs we have listed in Table 3.5 given their corresponding bug triggering inputs. Ideally, we would like to evaluate all the bugs listed in Table 3.5. However, setting up environment for reproducing each bug is difficult and time consuming. Most of the time, it requires installing a specific version of the program which sometimes requires installing a specific version of the operating system. Because of that, we decided to evaluate only a subset of the concurrency bugs listed in Table 3.5 whose reproducing environments are easy to setup on our test machine (e.g. those bugs that do not require re-installing the operating system on our test machine). We try to evaluate at least one bug from each popular application.

Table 4.1 lists the 13 bugs that we have evaluated in our experiments. Column-2 shows the name of each bug which matches that in Table 3.5. Among them, 4 (`Bug #1` to `Bug #4`) are synthetic bugs, 1 (`Bug #5`) is a code snippet extracted from a real buggy program, and 8 (`Bug #6` to `Bug #13`) are real bugs from real executions. There are 3 previously unknown bugs (`Bug #11 - Bug #13`), which are accidentally found during our experiments. We will discuss our stories of finding these three bugs in Section 4.5.2.

We want to know whether Maple is able to expose these bugs and how fast it can expose these bugs when comparing to other general concurrency testing tools. We compare Maple with two random testing techniques, `PCT` and `PCTLarge`. `PCT` [12] is a recently proposed random testing technique that provides probabilistic guarantee in exposing concurrency bugs. In `PCT`, threads are randomly assigned a non-preemptive strict priority (similar to that used in the active scheduler of Maple); during execution, `PCT` changes the priority of the currently running thread to lowest at random points d times. The authors state that most of the concurrency bugs can be exposed with a small value of d. In our experiment, we choose to use $d = 3$. `PCTLarge` is a variation of `PCT` that we proposed. It has the same algorithm as that in `PCT` except that it uses non-strict priorities instead of strict priorities. For instance, in Linux, we use nice values to serve as non-strict priorities. Higher priority threads will have more time quantum than lower priority threads. Interestingly, we found that `PCTLarge` usually performs better than `PCT`. More details are provided in Section 4.5.3.

For each bug, we run it repeatedly using its bug triggering input until the bug is triggered. Each time, a different testing technique is used. We compare the time needed by each testing technique to expose the bug. For Maple, we assume no previously built memoization database is available. The effect of memoization is discussed in Section 4.5.2. Table 4.1 shows the results. As shown in the table, Maple can expose all 13 bugs, including 3 previously unknown bugs (`Bug #11` to `Bug #13`). In contrast, `PCT` and `PCTLarge` can only expose 7 and 11 bugs respectively before timeout (24 hours) in reached. Moreover, Maple can expose all the real bugs much faster than `PCT` and `PCTLarge`. Maple uses more time to expose `Bug #5` than `PCT` and `PCTLarge`. This is because `Bug #5` is an idiom4 bug and a lot of time is spent testing irrelevant idiom1, idiom2 and idiom3 iRoots according to our current ranking mechanism. We found that `PCT` or `PCTLarge` expose bugs faster than Maple on some applications with small execution lengths (e.g. `Bug #3`). This is expected because the smaller the execution length, the higher the probability to expose the bug, but Maple has to pay a high cost for analysis. Nevertheless, the random techniques do not scale for long execution lengths (e.g. `Bug #8`). `Bug #10` does not have an idiom because it is value dependent.

Experiences in Finding Unknown Bugs

We found three previously unknown bugs. `Bug #11` was accidentally found when testing `Bug #9`, a documented bug in Aget. We observed a situation where the program hangs when testing `Bug #9`. We looked at the iRoot that caused the hang and tried the same iRoot again with the same random seed. In addition, we attached a tracer to the active scheduler to record the execution trace. The deadlock happened again in less than 5 runs [2]. With the help of the trace, we eventually found the root cause of this bug. The thread that handles signals is asynchronously canceled when holding an i/o lock (in `printf`), causing a deadlock in the main thread when it tries to acquire the same lock.

`Bug #12` is an intermittent bug in an CNC-based application that manifests as an assertion failure. CNC was developed by Intel and stands for Concurrent Collections. The particular application we examined is a server-client application that builds on Intel Thread Building Blocks (TBB) to synchronize threads. This bug was provided

[2]This is because we cannot faithfully replay some non-deterministic external events which are part of the program inputs.

ID	Bug Name	Type	Maple					PCT [12]		PCTLarge		# Non-Stack Mem Ops	# Thds	Native Time
			# Profile	Profile Time	# Test	Test Time	Total Time	# Runs	Time	# Runs	Time			
1	LogProcSweep	S	11	16.5	1	0.6	**17.1**	98511	**86400(TO)**	10169	**8188.6**	3.3K	3	0.1
2	StringBuffer	S	8	12.0	1	0.8	**12.8**	40	**56.4**	61	**49.1**	2.4K	2	0.1
3	CircularList	S	6	9.5	1	1.0	**10.6**	6	**9.1**	18	**14.6**	3.3K	3	0.1
4	BankAccount	S	6	9.0	1	0.9	**10.0**	12	**17.4**	44	**35.4**	3.6K	3	0.1
5	MySQL-1	E	8	13.2	100	120.8	**133.9**	18	**29.0**	15	**13.6**	4.9K	3	0.1
6	Pbzip2	R-K	8	151.9	2	3.2	**155.1**	26933	**86400(TO)**	3336	**6144.1**	32.1M	3	0.1
7	Apache-1	R-K	36	580.7	93	1544.2	**2124.9**	3485	**31688.0**	12951	**86400(TO)**	218.5K	5	3.6
8	MySQL-3	R-K	10	436.5	3975	43097.6	**43534.1**	11754	**86400(TO)**	10574	**81887.2**	1.8M	13	4.4
9	Aget-2	R-K	9	148.1	11	29.2	**177.4**	152	**355.0**	335	**619.5**	32.0K	3	0.1
10	Memcached	R-K	41	304.6	4	11.3	**316.0**	1010	**3635.1**	306	**782.5**	89.5K	4	1.2
11	Aget-1	R-U	9	74.7	18	123.9	**198.6**	32075	**86400(TO)**	47636	**86400(TO)**	529.5K	3	0.1
12	CNC	R-U	6	50.6	403	4163.8	**4214.4**	11063	**86400(TO)**	10012	**49046.8**	209.6K	3	1.1
13	Glibc	R-U	30	1120.4	20	36.6	**1157.0**	39560	**86400(TO)**	16147	**34349.1**	28.5M	4	0.1

Table 4.1: Bug exposing capability given bug triggering inputs. All the time reported in the table are in seconds. **TO** stands for timeout (24 hours). In the **type** column, **S** stands for synthetic bugs, **E** stands for extracted bugs, **R-K** stands for real bugs which are known, and **R-U** stands for real bugs which are unknown. * The root cause of `Bug #12` and bug `Bug #13` have not been confirmed yet. They are exposed when attempting idiom1 iRoots.

to us by a developer at Intel who could not expose it even after attaching a software deterministic record and replay tool to it [60]. Maple was able to expose the assertion failure in about 400 test runs, much faster than the two random testing techniques. However, because we do not have access to the source code, we could not help the programmer understand the root cause of the bug using iRoot.

The Glibc bug (`Bug #13`) was also accidentally discovered when testing `Bug #6` on a machine with glibc-2.5. It manifested as an assertion failure from the `free` function. We could reproduce the buggy interleaving using the same iRoot and the same random seed. The bug never showed up when a newer version of glibc was used. Since the memory management code in glibc is quite complex, the root cause has not been confirmed yet.

Memoization Help Expose Bugs Faster

We aware that applying memoization may affect the bug exposing capability of Maple. For example, if an iRoot cannot be exposed under some inputs, it does not mean that it is not feasible under other inputs. Since we put a limit on the total number of test runs on any iRoot in our current settings, the corresponding iRoot that leads to the bug might not be attempted when the bug triggering input is used, causing the bug to be missed. In order to see how memoization can affect the bug exposing capability, we evaluate 4 real bugs from Table 4.1 (`Bug #7` to `Bug #10`). Other real bugs are not chosen either because the bugs can be exposed using any input (`Bug #6`, `Bug #11` and `Bug #13`), or no other input is available (`Bug #12`). We first test the 4 bugs using inputs that do not trigger the bug to build the memoization databases. Then, we test the bugs using the bug triggering inputs. Table 4.2 shows the results. We can see that all the 4 bugs can be exposed when memoization is applied. More importantly, the time required to expose each bug

ID	Bug Name	Memo	# Profile	Profile Time	# Test	Test Time	Total Time
7	Apache-1	No	36	580.7	93	1544.2	2124.9
		Yes	22	357.6	2	18.3	375.8
8	MySQL-3	No	10	436.5	3975	43097.6	43534.1
		Yes	8	362.9	162	1953.6	2316.5
9	Aget-2	No	9	148.1	11	29.2	177.4
		Yes	6	100.5	8	21.8	122.3
10	Memcached	No	41	304.6	4	11.3	316.0
		Yes	36	272.6	5	12.1	284.8

Table 4.2: Memoization help expose bugs more quickly. All the time reported in the table are in seconds.

also reduces drastically. For instance, we save about 94% of the testing time for Bug #8. In fact, for the server application bugs, we save can a lot of testing time by memoizing those iRoots that are related to server start and server shutdown, clearly showing the benefit of memoization.

4.5.3 Usage Scenario 2: Coverage-Driven Testing

Another usage scenario that Maple can be helpful is when a programmer has a test input and wants to explore as many interleavings as possible for that input within the time budget. In this scenario, Maple can be used to cover more interleavings quickly. Also, memoization can prevent the programmer from testing the same interleaving multiple times, which helps reduce testing time.

Maple Achieves Higher Coverage Faster

The first question we want to address is whether Maple can expose more interleavings faster than other testing techniques. We use the interleaving idiom based coverage metric defined in Section 4.1. The coverage is represented using a tuple of numbers, each of which is the number of exposed iRoots for one idiom. For example, the following coverage $(1, 2, 5, 100, 50)$ means that the test has successfully exposed 1 idiom1 iRoot, 2 idiom2 iRoots, 5 idiom3 iRoots, 100 idiom4 iRoots and 50 idiom5 iRoots. We have implemented a tool, called observer, in our dynamic analysis framework to measure the coverage. The same observer is also reused in the profiler to observe exposed iRoots during profile runs so as to avoid testing these iRoots again during the test phase.

We compare it with 4 other testing techniques: `PCT`, `PCTLarge`, `RandDelay` and `CHESS`. `PCT` and `PCTLarge` have already been introduced in Section 4.5.2. `RandDelay` injects random time delay at random points during the execution. The number of points in which a delay is introduced is proportional to the execution length (one per 50K non stack memory accesses). The program is run on multi-core processors when `RandDelay` is used. `CHESS` [52] is a systematic testing tool. For a given program and a given input, it tries to explore all possible thread interleavings that have few preemptions. It was originally developed for Windows programs. We implemented it in our framework for Linux. Currently, it works for programs that use POSIX threads for synchronization. It employs the sleep-set based partial order reduction technique described in [50], and uses a fair scheduler discussed in [51]. We use a preemption bound of 2 throughout our experiments as suggested in [49] [3]. To handle a program that has data races, we run a dynamic data race detector first to find all racy memory accesses in the program, and then inform the `CHESS` scheduler so that it can explore different orderings of these racy memory accesses.

We use seven bug-free multi-threaded applications in this experiments, among which (`fft` and `radix`) are scientific applications from Splash2 [81], (`pfscan`, `pbzip2`, `aget`) are utility programs, and (`memcached` and `apache`) are server applications. For scientific and utility programs, we use random inputs (e.g. random number of thread, random files and directories, random URLs, etc.). For `memcached`, we wrote our own test cases which perform commonly used operations such as set/get keys and incr/decr keys. For `apache`, we use SURGE [9] to generate URLs and use `httperf` to generate parallel requests. Notice that when testing server applications, each test

[3]In fact, 13 out of 14 bugs studied in [49] are exposed with a preemption bound of 2. Using a preemption bound larger than 2 will drastically increase the number of test runs and exceed our time budget.

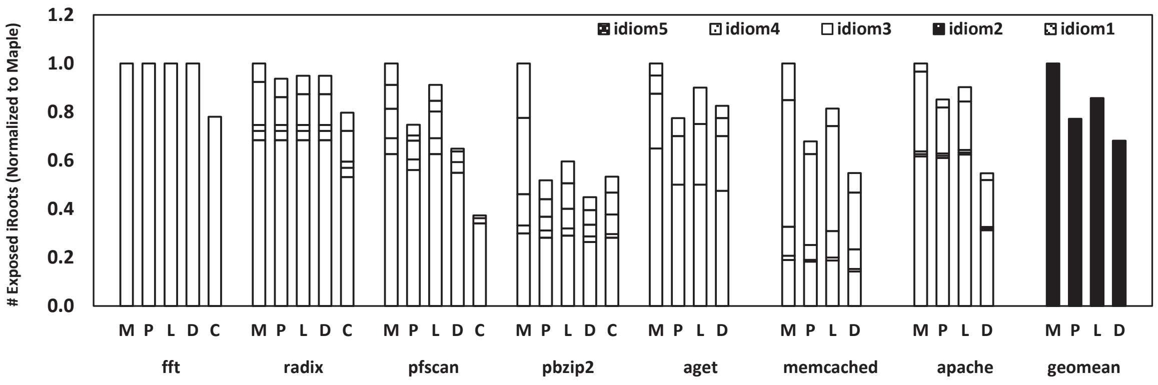


Figure 4.11: Comparison with different testing methods using the same amount of time. **M** stands for Maple, **P** stands for `PCT`, **L** stands for `PCTLarge`, **D** stands for `RandDelay`, and **C** stands for `CHESS`.

run consists of starting the server, issuing the requests, and stopping the server. This process is automated through scripting.

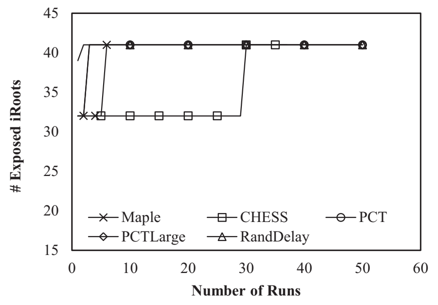
To compare Maple with these tools, we attach the same observer to each tool to collect the coverage after each test run. The current implementation of `CHESS` cannot identify the low level synchronization operations used in Glibc. Though we can treat those unrecognizable synchronization operations as racy memory accesses and still run `CHESS` on it, we believe this approach is unfair to `CHESS` as the number schedules to explore will increase unnecessarily comparing to the case in which we can recognize those synchronization operations. As a result, we decide to only consider iRoots from application code and ignore library code in this experiment to ensure a fair comparison.

Figure 4.11 shows the coverage achieved by these tools using the same amount of time as Maple does. We run Maple till its completion. For `apache`, as we are not able to run Maple till completion due to its scale, we test it for 6 hours. The observer overhead is excluded from the testing time. Y-axis is normalized to the coverage achieved by Maple. We are not able to run `CHESS` on `aget`, `memcached` and `apache` because in these applications, some non-deterministic events (e.g. network package arrival) are not controllable by `CHESS` [4]. From the results shown in Figure 4.11, we find that Maple achieves higher coverage faster than all the tools we have analyzed. On average, it achieves about 15% more coverage than the second best tool in our experiment. Also, we find `CHESS` only achieves about 60% of the coverage that is achieved by Maple using the same amount of time as Maple does. Be aware that the results shown in Figure 4.11 do not mean that Maple is able to explore more interleavings than random testing tools and systematic testing tools. In fact, `CHESS` explores a different interleaving in each test run. The results shown here convey a message that if we believe the interleaving coverage we come up with is a good coverage metric for concurrent testing, a specially engineered tool like Maple is able to achieve higher coverage faster than a more general testing tool such as `CHESS` and `PCT`.

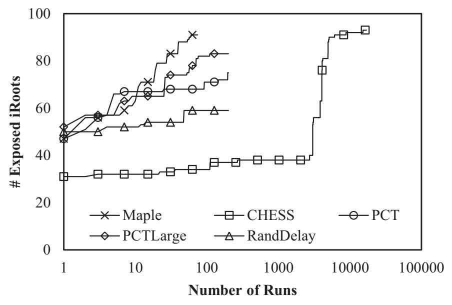
We also notice that `PCTLarge` performs better than other random testing techniques like `PCT` and `RandDelay`. We believe the reason is because `PCTLarge` has more context switches than others. As a result, we choose to use `PCTLarge` to randomize the profile runs in Maple.

Figure 4.12 shows the rate of increase in coverage using different testing tools. The X-axis is the number of test runs and the Y-axis is the total number of iRoots exposed so far. We only show results for those applications on which we are able to run `CHESS`. We run `CHESS` till its completion in this experiment. The results in Figure 4.12 further justify the fact that Maple is able to gain coverage faster than random testing tools and systematic testing tools. Also, we notice an interesting fact that `CHESS` experiences a slow start in gaining coverage. We believe this is due to the use of depth-first search strategy in `CHESS`. A best-first search strategy may

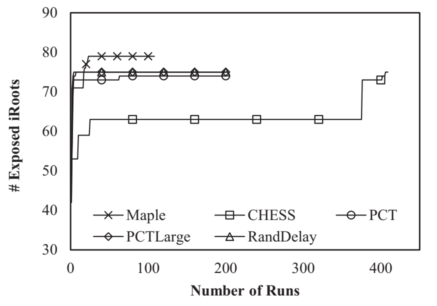
[4]Such programs are called non closed programs.



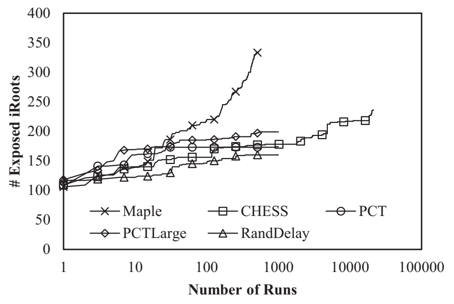
(a) `fft`



(b) `pfscan`



(c) `radix`



(d) `pbzip2`

Figure 4.12: Comparison with different testing methods. X-axis is the number of test runs, and Y-axis is the total number of iRoots exposed.

alleviate this problem at the cost of storing more states [15].

Memoization Help Reduce Testing Time

The next question we want to address is how much testing time we can save when memoization is applied under this usage scenario. To do that, for each bug free application, we test it using 8 different inputs ($input_i, i \in [1, 8]$). When testing with $input_{i+1}$, we compared the testing time between the following two methods: (1) without memoization database; (2) using the memoization database built from $input_1$ to $input_i$. We choose to memoize both the exposed iRoots and the fail-to-expose iRoots (we set the threshold to 6, i.e. each iRoot will not be attempted more than 6 test runs). For this experiment, we only test for idiom1 iRoots due to time constraints. Figure 4.13 shows the results. The Y-axis represents the testing time of the method that uses memoization (normalized to the testing time without memoization). The line plotted in red shows the average of the applications we

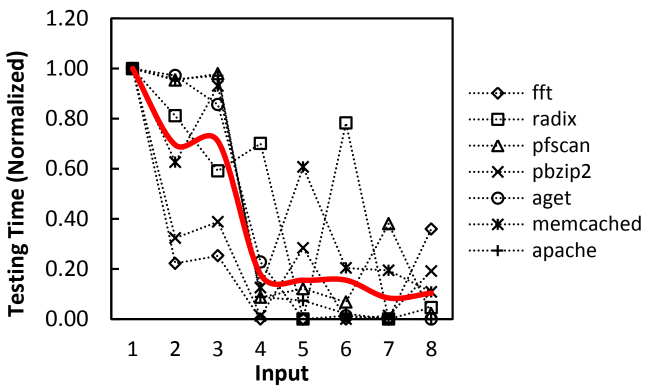


Figure 4.13: Memoization saves testing time. Y axis is normalized to the execution time without memoization.

analyzed. We observe that, with memoization, the testing time reduces gradually when more and more inputs are tested. For $input_8$, the average saving on testing time is about 90%. This clearly shows the benefit of memoization in reducing testing time.

4.5.4 Characteristics of Maple

In the following, we discuss the characteristics of Maple in terms of its efficiency (Section 4.5.4) and effectiveness (Section 4.5.4).

Performance Overhead

App.	Pinbase	Profiler	Active Scheduler
fft	6.9X	30.9X	16.3X
radix	6.9X	67.7X	17.8X
pfscan	8.3X	31.9X	27.7X
pbzip2	9.5X	183.3X	45.4X
aget	13.7X	34.4X	98.8X
memcached	2.1X	4.8X	4.1X
apache	1.7X	6.2X	6.0X
mysql	1.6X	15.7X	2.5X

Table 4.3: Runtime overhead of Maple comparing to native execution time.

Table 4.3 shows the average performance overhead of the profiler and the active scheduler for each application. We also include the Pinbase overhead, which is the overhead of PIN itself without any instrumentation. All the numbers shown in Table 4.3 are normalized to the native execution time. The overhead of the profiler varies depending on the applications. The overhead ranges from 5X (I/O bound applications) to 200X (memory intensive applications), and on average is at about 50X. The overhead of the active scheduler also varies, ranging from 3X to 100X. The average overhead is about 30X. We identify two major factors that contribute to the overhead of the active scheduler. One is due to the extra time delay that we introduce to solve the asynchronous external events problem. The other is because the candidate instructions of some infeasible iRoots are reached so frequently. We believe we still have room to improve the performance of the active scheduler.

Effectiveness of the Active Scheduler

App.	Idiom1	Idiom2	Idiom3	Idiom4	Idiom5
fft	87.5%	100.0%	40.0%	36.0%	25.0%
radix	66.7%	0.0%	0.0%	16.9%	5.6%
pfscan	13.3%	6.7%	5.6%	4.8%	6.4%
pbzip2	23.4%	23.1%	8.0%	5.6%	12.8%
aget	13.6%	3.6%	7.0%	2.7%	8.6%
memcached	7.8%	1.4%	8.4%	2.7%	7.1%
apache	6.0%*	1.0%*	0.0%*	7.0%*	4.0%*
mysql	5.0%*	0.0%*	1.0%*	1.0%*	2.0%*

Table 4.4: The success rate of the active scheduler (# successfully exposed iRoots / # total predicted iRoots). For `apache` and `mysql`, we experimented with 100 randomly selected candidate iRoots.

Finally, we discuss how effective the active scheduler is in exposing iRoots. For each application and each idiom, we collect the success rate of the active scheduler (# successfully exposed iRoots / # total predicted iRoots). We run Maple till its completion except for `apache` and `mysql` which exceed our time budget. For these two applications, we randomly sample 100 candidate iRoots for each idiom and report the success rate. On average, the active scheduler achieves about 28% success rate on idiom1, 17% on idiom2, 9% on idiom3, 10% on idiom4 and 9% on idiom5. We realize that the success rate for the active scheduler is not satisfactory. We identify three major reasons: (1) the profiler algorithm in not accurate in the sense that it cannot detect user customized happens before relations, producing many infeasible iRoots; (2) the active scheduler cannot deal with pre-conditions which might exist for some iRoots. (3) no dynamic information being associated with each iRoot combining with the fact that the currently candidate arbitration mechanism is not sophisticated enough causes some iRoots unlikely to be exposed. Nonetheless, Maple succeeds at exposing concurrency bugs faster than the state of the art randomization techniques, as previously demonstrated. We plan to further improve the accuracy of the profiler and the active scheduler in future.

4.6 Summary

Maple is a new coverage-driven approach to test multi-threaded programs. To this end, we discussed a coverage metric based on a generic set of interleaving idioms. We discussed a profile-based predictor that determines the set of untested thread interleavings that can be exposed for a given input, and an active scheduler to effectively expose them. A key advantage of our approach over random and systematic testing tools is that we avoid testing the same thread interleavings across different test inputs. Our experience in using Maple to test real-world software shows that Maple can trigger bugs faster by exposing more untested interleavings in a shorter period of time than conventional methods.

CHAPTER V

Avoiding Untested Interleavings I: PSet

Even with a tool like Maple discussed in Chapter IV, untested interleavings are inevitable and are the major cause of a majority of concurrency bugs. In this chapter, we discuss PSet, a runtime technique for avoiding untested interleavings in production runs. We first provide an overview in Section 5.1. Then, we introduce Predecessor Set (PSet) constraints, the way we use to encode tested interleavings in Section 5.2. We discuss our architectural support for enforcing PSet constraints in Section 5.3. Finally, we show our evaluation results in Section 5.4.

5.1 Overview

Even with a state-of-the-art testing tool like Maple discussed in Chapter IV, the interleaving space of most multi-threaded programs is so large that, a programmer cannot practically test all the legal interleavings of a program. In a well tested production code, a programmer would have tested most of the frequently occurring interleavings, but many of the infrequently occurring interleavings would remain untested. These rare untested interleavings are major cause of a majority of concurrency bugs in the production code. Thus, one of the fundamental problems with the shared-memory multi-threaded programming model is that, it exposes too many legal interleavings to the parallel runtime system, which makes it difficult for programmers to guarantee correctness.

In this chapter, we proposes to constrain a shared-memory multi-processor system, such that it avoids the untested thread interleavings during a production run. This improves correctness, because a tested interleaving is more likely to be correct than an untested interleaving. Also, the untested interleavings tend to occur infrequently during an execution, because otherwise they would have been tested in a high quality production code. Therefore, constraining a parallel runtime system to avoid the rare untested interleavings does not degrade performance significantly.

We face two challenges when designing such an interleaving constrained system. The first challenge is to develop a method for encoding the set of all tested thread interleavings in a program's executable using ISA extensions. We define a thread interleaving of an execution to be the order between the memory operations executed by all the threads. A thread interleaving is therefore unique to an execution of a program for a particular input. The challenge is to derive interleaving constraints from each tested interleaving. These interleaving constraints should be generic enough for different program inputs, so that enforcing them does not result in too many unnecessary constraint violations during the production runs. At the same time, the interleaving constraints should also be able to capture the set of tested interleavings, so that by enforcing them, we can avoid a majority of concurrency bugs due to the untested interleavings. Finally, the interleaving constraints should not be too complex so that the processor can easily enforce them at runtime.

In this chapter, we present an interleaving constraint called the Predecessor Set (PSet) constraint, which meets the above requirements. The PSet constraint is actually derived from the idiom-1 interleaving idiom discussed in Chapter III. Every static memory instruction in the program's binary has a PSet constraint defined for it. It encodes the set of exposed idiom-1 iRoots involved for that given static instruction. In other words, the PSet constraint for an instruction I specifies the set of all valid remote memory operations over which I can be immediately dependent upon. Intuitively, PSet constraints captures the tested interleavings between two dependent memory instructions. If a memory operation I was never immediately dependent upon another remote memory operation P in any of the test runs, then that dependency is avoided during the production runs.

We only consider the simplest interleaving idiom (idiom-1) in this chapter because we want to simplify the hardware design. We believe this simplification will not affect the bug avoidance capability significantly for two reasons. First, as shown in Table 3.2, about 60% of the bugs we have observed are idiom-1 bugs. Second, avoiding a concurrency bug is much easier than exposing it because we only need to avoid *one* of the inter-thread dependencies that lead to the bug. While to expose a concurrency bug, we need to find an interleaving that satisfies *all* the inter-thread dependencies that lead to the bug.

The second challenge is to efficiently enforce PSet constraints during production runs in order to avoid untested interleavings. We discuss extensions to a shared-memory multi-processor design, which efficiently detects PSet constraint violations and avoids them. Without processor support, we find that the performance overhead for enforcing the PSet constraints is very high, which would render the proposed interleaving constrained execution model to be impractical.

Our interleaving constrained processor detects PSet violations by keeping track of the last accessor information for every memory word and piggy-backing the coherence messages with that additional information. The processor recovers from a PSet constraint violation by either stalling the violating thread until the constraint is satisfied, or by re-executing the program from an earlier checkpoint with an alternative thread interleaving. The violated constraint is logged and sent back to the developer. The developer can then test the violated interleaving seen in the production run. If it is indeed a correct interleaving, then a binary patch to relax the relevant PSet constraint could be released.

Sufficient testing is required for the interleaving constrained processor to retain reasonable performance. However, it is likely that the production environments (e.g. hardware platforms, OS workloads) are substantially different from the testing environments, and the frequencies with which particular interleavings occur might be different from what we observed during testing. If that happens, the interleaving constrained processor may encounter more PSet constraint violations, causing performance problems. In such scenarios, we always have a choice to turn off the protection to recover performance if we find the performance overhead is too high and outweighs the extra reliability we get.

We discuss a software tool built using PIN [46], which we use to profile applications during the test runs and derive the PSet constraints. It is also used to detect and avoid the PSet constraint violations during real executions. This tool is used to analyze the effectiveness of the proposed mechanism in avoiding 17 bugs in the open source programs such as MySQL, Mozilla, Apache, etc. We show that by enforcing PSet interleaving constraints, we can effectively avoid not only bugs due to data races and atomicity violations, but also other forms of concurrency bugs that cannot be found using the existing dynamic bug detection tools.

We also analyze the accuracy and performance of our tool using Splash [81] and Parsec [10] benchmark suits. We show that the number of PSet constraint violations in a bug free execution is very less, and as a result, the performance cost of enforcing the PSet constraints is negligible.

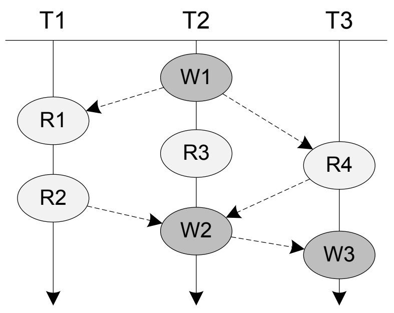
5.2 Encoding Tested Interleavings

This section discusses the Predecessor Set interleaving constraints. They are derived from correct test runs, and are encoded in the program binary. We then discuss the effectiveness of PSets in avoiding concurrency bugs using several real examples. Finally, we discuss the limitations of the PSet constraints.

5.2.1 Predecessor Sets (PSets)

We are taking the first step towards constraining a shared-memory multi-processor system to follow the tested interleavings. In order to simplify the hardware design, we focus on constraining just the interleaving between two dependent memory operations using the Predecessor Set constraints. The PSet constraints are derived from the idiom1 interleaving idiom, the simplest idiom, discussed in Chapter III. We show that even with this simplification, the PSet constraints are powerful enough to avoid most of the data race bugs, atomicity violation bugs, and also other concurrency bugs.

A Predecessor Set (PSet) is defined for each static memory operation. It encodes the set of exposed idiom1 iRoots involved for that given static memory operation. More precisely, the PSet for a memory instruction specifies the set of all memory operations over which it can be immediately dependent upon. We consider true (read-after-write) as well as false (write-after-write and write-after-read) dependencies. We consider all thread local memory dependencies (where the two dependent memory operations were executed in the same thread) to be valid during production runs. Therefore, a PSet constraint specifies only the set of valid remote dependencies for an instruction.



```
PSet(W1) = {}
PSet(W2) = {R2, R4}
PSet(W3) = {W2}
PSet(R1) = {W1}
PSet(R2) = {}
PSet(R3) = {}
PSet(R4) = {W1}
```

Figure 5.1: PSet constraints for an interleaving.

A static memory operation M contains another static memory operation P in its PSet only if the following conditions are satisfied. Either P or M should be a write. Also, there should be at least one dynamic instance in any of the tested correct interleavings, such that (a) P and M were executed in two different threads (say, `T1` and `T2`), (b) M was immediately dependent on P in that interleaving, and (c) neither `T1` nor `T2` executed a read or a write (to the same memory location as M) that interleaved between P and M. During a production run, the runtime system detects a violation while executing a memory operation M, if M is memory-dependent on a remote memory operation P, but P is not in the PSet of M.

Figure 5.1 shows a tested interleaving, and the resultant PSets. Assume that all the memory operations shown in the figure are to the same memory location, and each of them is a different static memory operation. Reads are labeled with the prefix `R` and writes are labeled with the prefix `W`. The PSet for `W2` contains two reads due to two write-after-read dependencies. Note that the PSet for `R2` is empty even though it is immediately dependent on the remote memory operation `W1`, because `R1` interleaves between `W1` and `R2` (refer condition (c) listed above). In this tested interleaving, the values read by `R1` and `R2` would be the same. An empty PSet for `R2` ensures that the value read by `R1` and `R2` are the same even in the production runs. Including `W1` in the PSet of `R2` would not guarantee this property during production

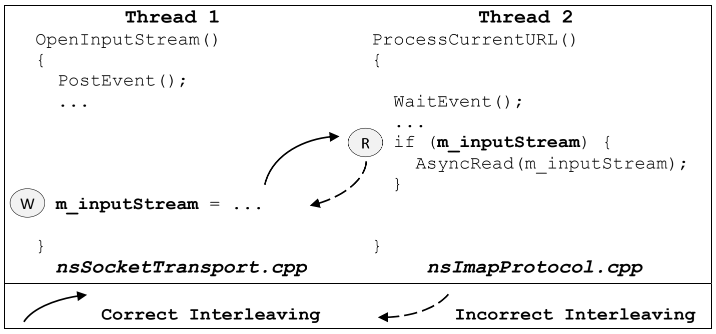


Figure 5.2: A data race bug in Mozilla (Mozilla-7 in Table 3.5).

runs.

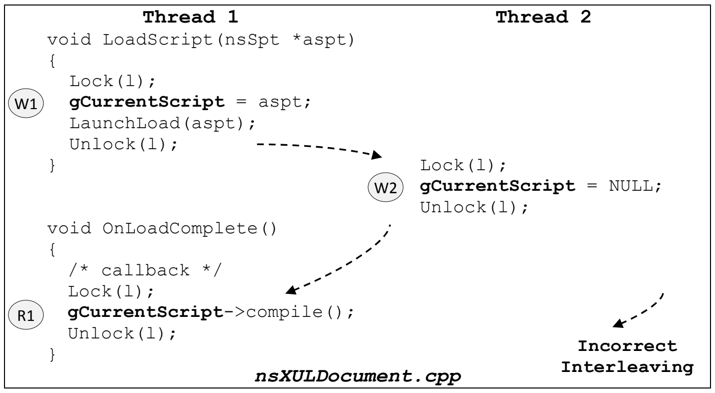
5.2.2 Effectiveness of PSets in Avoiding Concurrency Bugs

We now describe how enforcing the PSet constraints avoids harmful data race bugs (while still allowing benign data races for performance), atomicity violations and other forms of concurrency bugs.

Enforcing PSet Constraints Avoids Data Races

Figure 5.2 shows a data race bug in Mozilla. In this example, the variable `m_inputStream` points to a heap location that is dynamically allocated on receiving an input. During the correct test runs, the only valid interleaving between `W` and `R` is `W` $\rightarrow$ `R`. Therefore, the PSet for `R` contains `W`, and the PSet for `W` is a null set. For this data race bug to manifest in a production run, `R` should precede `W`. However, such an interleaving would result in at least one PSet constraint violation. The predecessor for `W` in an incorrect interleaving would be `R`. Since `R` is not in the PSet for `W`, a PSet constraint violation would be detected.

A benign data race could occur frequently in a program's execution. Therefore, if we use a data race detector to avoid data races at runtime, we might hurt performance. However, a PSet constraint based mechanism does not have this issue, as PSets can capture the fact that a benign data race interleaving is a correct interleav-



```
        Thread 1                              Thread 2
  void LoadScript(nsSpt *aspt)
  {
    Lock(l);
W1  gCurrentScript = aspt;
    LaunchLoad(aspt);
    Unlock(l);
  }                                       Lock(l);
                                     W2   gCurrentScript = NULL;
                                          Unlock(l);
  void OnLoadComplete()
  {
    /* callback */
    Lock(l);
R1  gCurrentScript->compile();
    Unlock(l);
  }
```

Figure 5.3: An atomicity violation bug in Mozilla [43] (Mozilla-1 in Table 3.5).

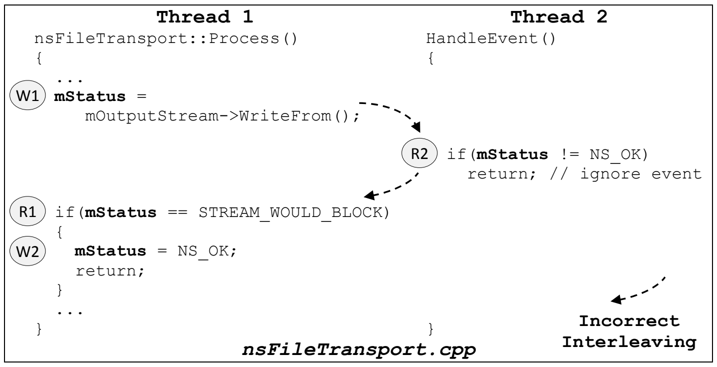
ing, provided that interleaving is seen in a correct test run.

Enforcing PSet Constraints Avoids Atomicity Violations

Figure 5.3 shows an atomicity violation bug in Mozilla. The memory operations `W1-R1` are expected to execute atomically. `W2` would never be immediately dependent on `W1` in any of the correct test runs. Therefore, the PSet for `W2` would not contain `W1`. In an incorrect execution, the atomicity property of `W1-R1` could be violated by an interleaving `W2`. However, this would cause a PSet constraint violation at `W2`, as the PSet of `W2` would not contain `W1`.

For this example, we would detect a PSet violation at `W2`, whereas AVIO [43], a state-of-the-art atomicity violation detector, can only detect the violation later at `R1`. A PSet constraint violation can be detected at least as early as an AVIO constraint violation, as the PSet constraints are a super-set of the AVIO constraints.

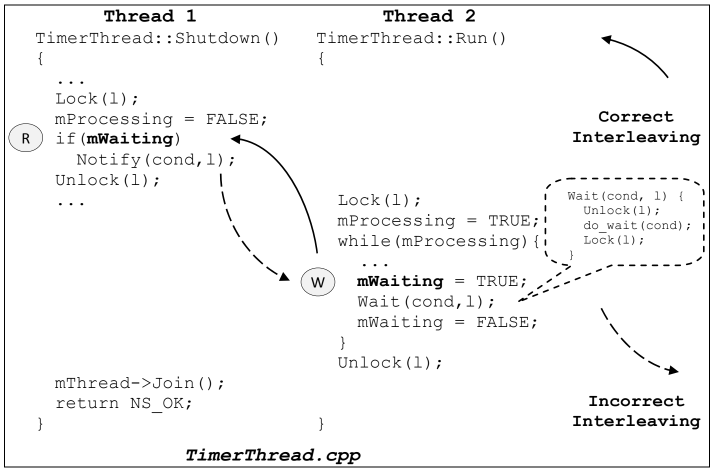
Figure 5.4 shows an atomicity violation bug that AVIO [43] cannot detect. The programmer expects that the operations `W1-R1-W2` be executed atomically. Therefore, the PSet for `R2` learned from all the correct test runs would not contain `W1`. When the required atomicity property for the operations `W1-R1-W2` is violated by `R2` in a production run, a PSet violation would be detected at `R2`.



```
nsFileTransport::Process()
{
  ...
  mStatus =
      mOutputStream->WriteFrom();

  if(mStatus == STREAM_WOULD_BLOCK)
  {
    mStatus = NS_OK;
    return;
  }
  ...
}
```

```
HandleEvent()
{

  if(mStatus != NS_OK)
     return; // ignore event

}
```

Figure 5.4: An atomicity violation bug in Mozilla, which will not raise an AVIO [43] invariant violation. (Mozilla-4 in in Table 3.5).



```
TimerThread::Shutdown()
{
  ...
  Lock(l);
  mProcessing = FALSE;
  if(mWaiting)
    Notify(cond,l);
  Unlock(l);
  ...

  mThread->Join();
  return NS_OK;
}
```

```
TimerThread::Run()
{
  Lock(l);
  mProcessing = TRUE;
  while(mProcessing){
    ...
    mWaiting = TRUE;
    Wait(cond,l);
    mWaiting = FALSE;
  }
  Unlock(l);
}
```

```
Wait(cond, l) {
  Unlock(l);
  do_wait(cond);
  Lock(l);
}
```

Figure 5.5: Order violation bug in Mozilla, which is neither a data race nor an atomicity violation. (Mozilla-9 in Table 3.5).

Enforcing PSet Constraints Avoids Order Violations

Figure 5.5 shows a concurrency bug in Mozilla that neither a data race detector nor an atomicity violation detector can detect. The function `Notify()` in `Thread1` should be invoked only after `Thread2` executes the `Wait()` function. Otherwise, `Thread2` would block forever. The PSet for `W` learned from the correct test interleavings would not contain `R`. In an incorrect interleaving during a production run, `R` would be `W`'s predecessor, and therefore a PSet constraint violation would be detected at `W`.

5.2.3 Deriving and Encoding PSets Constraints

The predecessor sets are constructed from the test runs using a profiling tool that we built using PIN [46]. The programmer has to ensure that the test run is correct. This could be done by verifying the program output and by checking the test run using dynamic bug detection tools.

Figure 5.6 shows the format of an instruction with its PSet information for a 32-bit ISA. The field P-Type has two bits. If the P-Type value for an instruction is three, then the next field specifies the number of instructions in the PSet for that instruction. Each of the remaining fields specify an instruction in the PSet. An instruction is represented using a concatenated value of the identifier for its library and its relative offset that refers to the instruction's location in the library. This is necessary to support programs with dynamically loaded libraries.

The worst case space complexity for the PSets of a program is $O(N^2)$, where `N` is the number of static memory instructions in the program. The reason is that, each static instruction can have at most `N` elements in its PSet. However, in Section 5.4 we show that, on average, about 95% of static instructions have a PSet of size zero, as a huge proportion of memory operations are thread local accesses. For such instructions, there is no additional space overhead.

Programmers commonly use a testing metric called basic block coverage. It measures the percentage of static instructions that were executed in at least one test run. Even for high quality production code, basic block coverage is typically less than 100%. For instructions that were never tested even once, we could assume that its PSet is a null-set. This could ensure a high degree of fault tolerance. Alternatively, one could choose not to enforce PSet constraints for such untested instructions to reduce the number of false constraint violations during production runs. However,

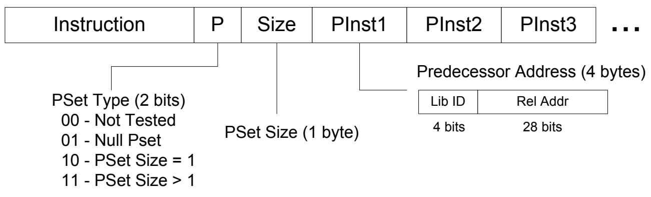


Figure 5.6: Format for encoding an instruction's PSet.

untested instructions are also likely to occur rarely, because otherwise it would have been tested in a well tested program. Therefore, assuming null PSets for untested instructions in a well tested program would not result in significant number of false constraint violations.

5.2.4 Limitations

The PSet constraints described does not account for the interleavings between two or more memory operations accessing different memory locations due to hardware complexity concern. As a result, it may not be able to avoid certain bugs due to multi-variable atomicity violations. Another limitation of PSets is that they are context insensitive. Additional context such as the calling stack could help avoid more untested interleavings. In future, we plan to extend the PSet interleaving constraints so that we can avoid most of the untested interleavings. However, our analysis in Section 5.4 shows that the PSets constraints are powerful enough to avoid 15 out of 17 concurrency bugs that we analyzed.

5.3 Enforcing Tested Interleavings

In this section, we first discuss methods to detect and avoid PSet constraint violations. Using these methods we ensure that most of the untested interleavings are avoided during production runs. We then discuss the architectural support for detecting and avoiding PSet constraint violations.

5.3.1 Detecting and Enforcing PSet Constraints

During a production run, whenever a memory operation is immediately dependent on a remote memory operation, the runtime system checks to see if the remote memory operation is in the predecessor set of the current memory operation. If not, a PSet violation is detected.

To repair the violation, we evaluate two approaches. In one approach, the violating memory operation is stalled until the violation gets resolved. When the violating thread is stalled, other threads continue to make progress. If another thread executes a memory operation to the same memory location as the violating memory operation, the violated PSet constraint is checked again. If the check succeeds, the stalled thread continues its execution.

A repair mechanism based on stalling the violating threads is easier to support and is also performance efficient. However, not all PSet constraint violations can be avoided using this mechanism. Because, for a constraint to get resolved, another thread should be able to make progress so that it eventually accesses the same memory location as the stalled memory operation. But, it is possible that the other thread needs a lock before it can access that memory location, and that lock might be currently held by the stalled thread. Thus, stalling the violating thread might not resolve the violation. Consider another example where a violating memory operation's PSet is a null-set. In this case, any remote memory dependency would cause a violation, and therefore waiting for the other threads to execute a different memory operation is never going to resolve the violation. To ensure forward progress while using a stalling mechanism, we use a time-out scheme, where the stalled thread is released to continue its execution (or the second recovery scheme is triggered, if available) when the stall time has reached a particular threshold.

We also evaluate another recovery mechanism to avoid PSet constraint violations. It is based on a checkpoint and rollback mechanism. On detecting a violation, the program is re-executed from an earlier checkpoint. During re-execution, the thread schedule is perturbed to induce an alternative interleaving. Since, the constraint violations are likely to be rare events, it is unlikely that the same violation would be encountered again during re-execution.

Not all PSet constraint violations can be avoided by just perturbing the thread schedule. It is possible that the only legal interleaving for an input is one that is untested. Such violations would cause repeated rollbacks to the same checkpoint. To ensure forward progress, the maximum number of rollbacks to a checkpoint is set to a threshold. When the number of rollbacks to a checkpoint has reached the threshold, that checkpoint is discarded, and another checkpoint is taken at the point where a PSet violation is detected. The system then logs the violation and continues with the execution. This ensures forward progress. The log is sent back to the developer to test the untested interleaving and determine if it is a cause of a bug or not. If it is not a bug, then the relevant PSet is updated in order to allow the newly tested interleaving at runtime.

5.3.2 Architectural Support

We implemented a profiler that learns PSets from the test runs using PIN [46]. We also implemented a runtime monitor to detect PSet constraint violations and avoid concurrency bugs using PIN [46]. The runtime overhead for this runtime monitor is about 100x for server applications such as MySQL and Apache, but it is over 200x for memory intensive applications like Splash [81] and Parsec [10]. Therefore, to constrain the interleavings at runtime using PSet constraints, adequate processor support is a must.

We discuss architectural support for detecting PSet constraint violations in this section. In addition, we also need checkpoint support for rollback and re-execution. This is a well researched problem. A copy-on-write mechanism can be supported in the operating system [64] or in the processor [72, 63]. During re-execution, we induce a different thread interleaving. The execution cannot be rolled back past a committed system state. But as we show in Section 5.4, the rollback window length required to avoid a majority of concurrency bugs is small.

We now discuss architectural support for detecting PSet constraint violations. The instruction set architecture (ISA) needs to be extended to let the developers specify the PSet constraints. Section 5.2 discussed an instruction encoding for specifying the PSet constraint for an instruction. A processor needs to execute a check for a memory operation, if it has a PSet constraint specified in the instruction code.

Tracking Last Writer and Last Reader(s)

To execute the checks, the processor needs to keep track of either the last writer or the set of last readers for every memory location. We propose to extend the caches to keep track of this additional meta-data for every memory location. When a cache block is evicted, the information is lost. But as described in Section 5.2, most of the concurrency bugs are tightly interleaved. Therefore, the loss in information due to a cache eviction is not significant.

The coherence reply messages (write-update replies and acknowledgments for invalidations) are piggy-backed with the meta-data corresponding to the cache block. The processor core receiving the reply, stores the received information in its private cache along with the information that the last reader or the writer information belongs to a different thread.

Checking PSet Constraints

We propose to use DISE [16] for efficiently executing the PSet check for every memory operation that has a PSet constraint. A check needs to be executed for a memory operation, only if the last reader(s)/writer to the memory location accessed by the current instruction belongs to a different thread. Thus, in the common case, no check needs to be executed for a memory operation. Also, for a majority of instructions, the PSet is a null-set (including all thread local accesses). We show that less than 5% of static memory operations have a PSet size greater than one, and therefore the check for a memory operation could be very efficient. If a check is executed for a memory operation, it checks if the last writer or the last set of readers is a member of the current memory operation's PSet.

5.4 Evaluation

We discuss several results in this section. First, we discuss the bug avoidance capability of an interleaving constrained shared-memory multi-processor that enforces the PSet constraints. We analyze its capability in detecting *and* avoiding 17 concurrency bugs in several multi-threaded applications such as Mozilla, MySQL, Apache, etc. We also analyze if these bugs can be detected by a happens-before based data race detector [21] and AVIO [43]. Second, we analyze the number of tests it takes to learn the PSet constraints adequately, and compare it with another test based AVIO bug detection tool [43]. Third, we discuss the number of PSet constraint violations in real executions (using input different from the ones used for training), and the overhead in resolving the PSet constraint violations using stalling and rollback mechanisms. Finally, we analyze the size of PSets and the memory space overhead to express PSet constraints in the binary and to keep track of them during production runs. These results are based on our PSet constraint tool implemented using PIN [46].

5.4.1 Bug Avoidance Capability

We analyzed 17 bugs that were known to us at the time we performed our experiments. These bugs are listed in Table 5.1. Column-2 shows the unique name of each bug used in our experiments which matches that in Table 3.5. A short description about each bug we have analyzed can be found in Table 3.5. In our experiments, we evaluated four real bugs (`Bug #1`, `Bug #2`, `Bug #4` and `Bug #5`) and one injected bug (`Bug #3`). For the rest of the bugs, we analyzed their extracted versions, as these bugs manifest only under a very specific interleaving that is very difficult to reproduce and analyze.

As shown in Table 5.1, the proposed PSet constraint based detection tool detected all the bugs, except the last two bugs listed in Table 5.1. One bug (`Bug #16`) is related to an incorrect interleaving between memory operations accessing different locations. The other one (`Bug #17`) is a deadlock bug. In order to detect this bug, the PSet constraint needs to be context sensitive. AVIO [43] can detect 6 atomicity violation bugs, but cannot detect one atomicity violation bug (`Bug #10`), which we discussed in Section 5.2. A happens-before data race detector can detect all the bugs, except five data race free bugs (`Bug #3`, `Bug #7`, `Bug #15`, `Bug #16` and `Bug #17`).

Our PSet based tool detected `Bug #3` and `Bug #15`, which neither the data race detector nor AVIO [43] could detect. Thus, PSet constraint based concurrency bug detector is effective in detecting all the concurrency bugs that traditional tools find, and also has the potential to detect other memory ordering related concurrency bugs.

We now analyze the bug *avoidance* capability of the proposed constrained shared-memory multi-processor runtime system. Table 5.2 shows all the 15 bugs that were

ID	Bug Name	D.R.D	AVIO	PSET
1	Pbzip2	Yes	No	Yes
2	Aget-2	Yes	Yes	Yes
3	Pfscan	No	No	Yes
4	Apache-1	Yes	Yes	Yes
5	MySQL-1	Yes	Yes	Yes
6	MySQL-8	Yes	No	Yes
7	Mozilla-1	No	Yes	Yes
8	Mozilla-2	Yes	Yes	Yes
9	Mozilla-3	Yes	Yes	Yes
10	Mozilla-4	Yes	No	Yes
11	Mozilla-5	Yes	No	Yes
12	Mozilla-6	Yes	No	Yes
13	Mozilla-7	Yes	No	Yes
14	Mozilla-8	Yes	No	Yes
15	Mozilla-9	No	No	Yes
16	Mozilla-10	No	No	No
17	OpenLDAP	No	No	No

Table 5.1: Bug detection capability. Comparing PSet with a happens-before data race detector and AVIO [43].

detected by the PSet violation detector. Six bugs were avoided using the stalling mechanism that we described in Section 5.3, and the rest of the bugs require support for a rollback and re-execution mechanism. Table 5.2 also lists the number of static and dynamic PSet constraint violations detected by our tool. The constraint violations are classified into true and false constraints. The true constraint violations are related to the bug. The false constraint violations are due to insufficient training during testing. The performance impact due to false constraint violations is discussed in Section 5.4.3. Table 5.2 also lists the number of instructions that need to be rolled back to avoid the bugs that we analyzed. As expected, the required rollback window size is small. This is because, most of the concurrency bugs are due to temporally tight interleaving between the memory operations. A rollback window of size zero means that the bug was avoided by just stalling the violating thread.

5.4.2 Learning PSet Constraints

For the runtime system to be efficient, PSet constraints should be complete enough to allow valid frequent interleavings between memory operations at runtime. In this section we discuss how soon the number of new PSets learned reaches a saturation point, and compare it with another profiling based bug detection tool called

ID	Bug Name	Type	Stall	Rollback	True Constraint Violations		False Constraint Violations		Rollback Window Size
					Static	Dynamic	Static	Dynamic	
1	Pbzip2	Real	Yes	Yes	1	1	3	3	0
2	Aget-2	Real	No	Yes	1	1	2	2	11
3	Pfscan	Injected	No	Yes	1	1	0	0	51
4	Apache-1	Real	No	Yes	2	20	1	1	358
5	MySQL-1	Real	Yes	Yes	1	7	3	6	0
6	MySQL-8	Extract	No	Yes	1	1	0	0	4760
7	Mozilla-1	Extract	No	Yes	1	1	3	3	1664
8	Mozilla-2	Extract	No	Yes	2	2	1	1	1224
9	Mozilla-3	Extract	No	Yes	1	1	0	0	1210
10	Mozilla-4	Extract	Yes	Yes	1	1	0	0	0
11	Mozilla-5	Extract	Yes	Yes	1	1	0	0	0
12	Mozilla-6	Extract	Yes	Yes	1	1	0	0	0
13	Mozilla-7	Extract	No	Yes	1	1	0	0	821
14	Mozilla-8	Extract	Yes	Yes	1	1	0	0	0
15	Mozilla-9	Extract	No	Yes	2	2	1	1	1674

Table 5.2: Avoiding bugs using PSet constraints. True constraint violations are related to the bug.

AVIO [43].

Testing Methodology

PSet constraints used in Section 5.4.1 were learned from the correct test runs. Here we describe the input we used to test our multi-threaded programs and learn the PSet constraints. These input are different from the ones used for the bug avoidance (Section 5.4.1) and the false positive analysis (Section 5.4.3). `Pbzip2` is a parallel implementation of `Bzip2`, which does file compression and file decompression. We compressed a random file in each test run. `Aget` is a download accelerator that spawns multiple threads to download different chunks of a file in parallel. For each test run, we downloaded a random file from the Internet. `Pfscan` is a multi-threaded file scanner, which combines the functionality of `find`, `xargs` and `fgrep`. We searched a random string from a randomly chosen file or directory in each test run. We also evaluated two server applications, `Apache` and `MySQL`. For `Apache`, each test run consists of issuing a session of requests to a set of static web pages using `httperf`. For `MySQL`, each test run consists of running the regression test suite that is available for public. In parallel with the regression test suite, we also continuously run the OSDB (Open Source Database Benchmark [6]) multi-user test to emulate a concurrent workload. For these five programs, we used the same version as the ones used for the bug avoidance analysis (from `Bug #1` to `Bug #5`), and the PSet constraints derived in this section are used in the bug avoidance analysis. In addition, we also evaluated six bug free applications, four of them (`FFT`, `LU`, `Radix`, `FMM`) are from the Splash2 [81] benchmark suite, and two of them (`Blackscholes` and `Canneal`) are from the Parsec [10] benchmark suite. For these programs, we chose a random input parameter for each test run.

Programs	Stall	Rollback	Cannot Resolve	Total PSet Constraint Violations	Inst. Count
pbzip2	1	5	0	6	1.3E+9
aget	0	0	0	0	1.1E+7
pfscan	1	2	0	3	7.4E+7
apache	1	4	0	5	2.8E+8
mysql	0	2	2	4	9.7E+8
fft	0	0	0	0	2.3E+8
fmm	1	0	0	1	1.6E+9
lu	0	1	0	1	1.6E+8
radix	0	0	0	0	6.4E+7
blackscholes	0	0	0	0	8.1E+8
canneal	1	0	0	1	7.0E+9

Table 5.3: PSet constraint violations in bug-free executions.

Tests Required to Learn PSet Constraints

Figure 5.7 shows the number of new PSet pairs learned in each test run. Each point along the x-axis represents a unique test run, and the y-axis represents the number of new PSet pairs derived from a particular test run. PSet takes more test runs to stabilize than AVIO, because it captures more constraints than AVIO. These results show that the tests used during the quality assurance process should be adequate to learn the PSets.

5.4.3 PSet Constraint Violations in Bug Free Executions

We now discuss the number of false PSet constraint violations in bug free executions. We used the same set of benchmarks that we used for the results in Section 5.4.2. The input used to analyze the false PSet constraint violations is different from the training input. For `Pbzip2`, we used a different set of files as input. For `Aget`, some new files were downloaded. For `Pfscan`, we searched some new strings from different files and directories. For `Apache`, we used `httperf` to issue concurrent requests to a set of static web pages which are not used in the test runs. For `MySQL`, we used the tool in OSDB to randomly generate a new database with a size different from the one used for training, and ran the OSDB multi-user test. For Splash2 and Parsec programs, we randomly selected inputs and parameters not used in the test

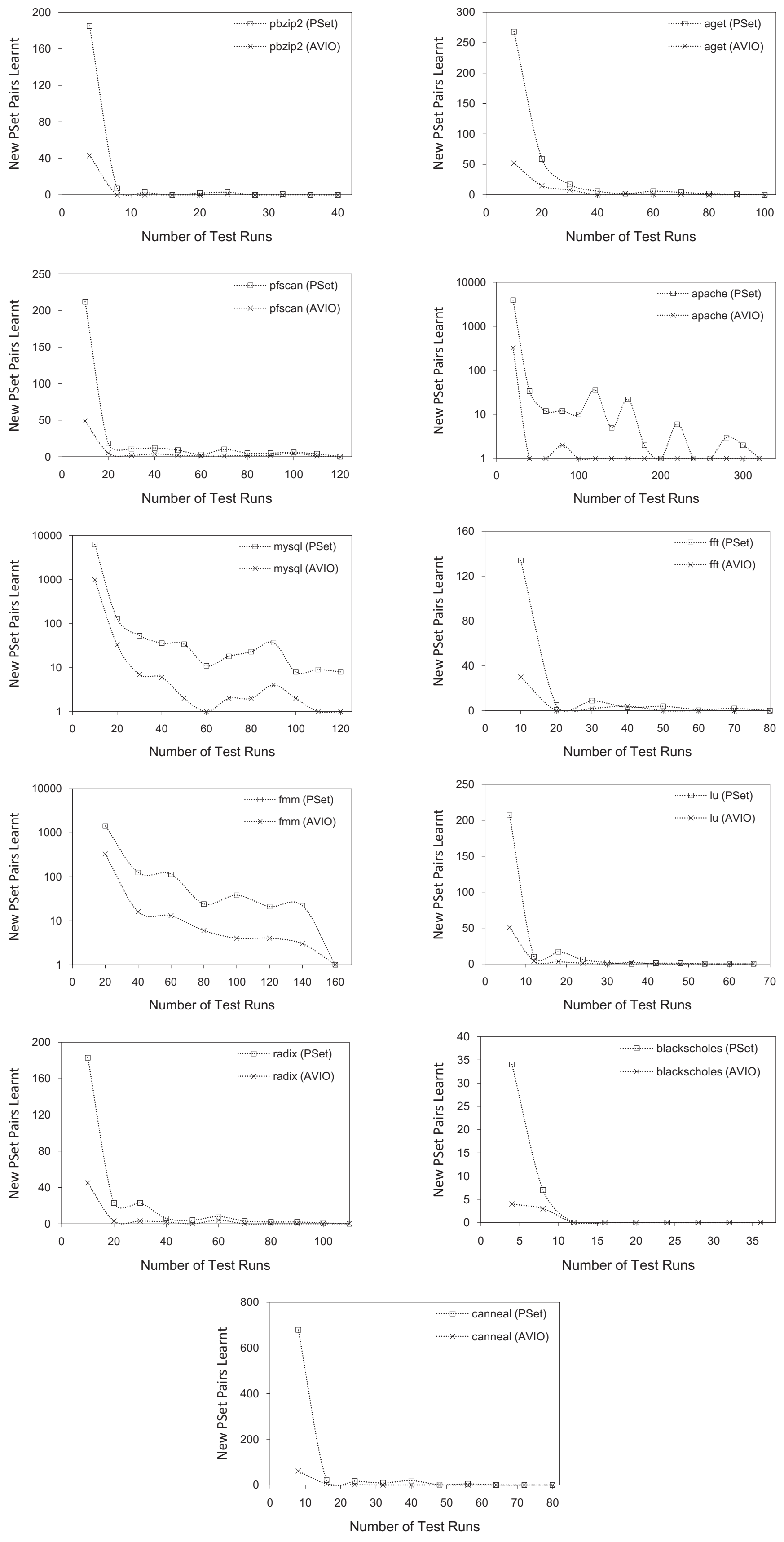


Figure 5.7: Number of test runs required for learning PSets and AVIO invariants.

runs.

Table 5.3 shows the static and dynamic number of PSet constraint violations, and also the total number of instructions executed. All the data shown in the table are the cumulative results of 10 runs, except for `MySQL`. For `MySQL`, we run the OSDB multi-user benchmark once. The results show that the number of false constraint violations are very few. For example, for `Pbzip2`, we detected 6 constraint violations while executing over 1.3 billion instructions. We avoided one constraint violation by just stalling the violating thread. The other five constraint violations needed rollback and re-execution to resolve them. Even for a rollback window of length 100,000 (which is more than sufficient for resolving most of the concurrency bugs as discussed in Section 5.4.1), five violations would result in additional 0.5 million instructions being executed at runtime. But this is a small fraction (0.04%) when compared to 1.3 billion instructions executed by the original program. Thus, false constraint violations are infrequent enough that they are likely to not impact performance.

Detecting PSet invariant violations and maintaining checkpoints can also degrade performance, but both of these costs can be ameliorated using processor support (discussed in Section 5.3).

We also noticed that there are two PSet constraint violations in `MySQL` that cannot be resolved by both stall and rollback mechanisms, because there is no legal interleaving that would not violate the PSet constraints. This is due to insufficient testing. We found that these two violations are from the function `bmove512()` (in `bmove512.c`). This function moves 512 bytes in the heap. To improve performance, the programmer has manually unrolled a loop in the function 128 times resulting in 128 reads and 128 writes within the loop. In a test run, only very few bytes are touched in the heap before the function `bmove512()` gets executed. As a result, in a

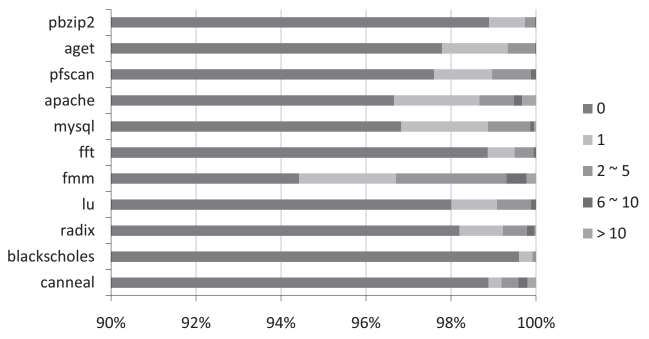


Figure 5.8: Proportion of static memory instructions with a particular PSet size (normalized to the total number of static memory instructions in the application binary and libraries that were executed in at least one test run).

test run, only a few memory instructions within the loop have a predecessor memory operation. This makes it difficult for us to learn all the legal interleavings involving this function using our current testing methodology. However, if we can perform more complete industry-level testing, such violations should also disappear.

When we detect a PSet violation that we cannot avoid, in addition to logging it for post-mortem analysis, the corresponding PSet is updated so that no future violations due to the same interleaving would be detected. A PSet violation that we manage to avoid are also logged for post-mortem analysis, but the corresponding PSet is *not* updated so that the system continues to avoid a potential bug related to the PSet violation.

5.4.4 Memory Space Overhead

We now discuss the memory space overhead in expressing the PSet constraints in the binary and tracking them at runtime. We use the same set of benchmarks that we described in Section 5.4.2.

Figure 5.8 shows the distribution of the PSet sizes for several programs that we analyzed. The x-axis is normalized to the total number of memory instructions in the application binary and the libraries that were executed at least once in the test

Programs	App. Size	App+Lib Size	# PSet Pairs	PSet Size	Overhead w.r.t App.	Overhead w.r.t App+Lib
pbzip2	39KB	3.70MB	201	0.84KB	2.16%	0.02%
aget	90KB	2.04MB	365	1.53KB	1.69%	0.08%
pfscan	17KB	2.08MB	295	1.25KB	7.34%	0.06%
apache	2435KB	8.60MB	4119	16.80KB	0.69%	0.20%
mysql	4284KB	8.19MB	6604	27.58KB	0.64%	0.34%
fft	24KB	2.59MB	158	0.67KB	2.74%	0.03%
fmm	73KB	2.64MB	1764	7.39KB	10.13%	0.28%
lu	24KB	2.59MB	244	1.03KB	4.31%	0.04%
radix	21KB	2.59MB	255	1.07KB	5.00%	0.04%
blackscholes	54KB	3.65MB	41	0.17KB	0.32%	0.00%
canneal	59KB	3.66MB	752	3.10KB	5.24%	0.08%

Table 5.4: Binary Size Increase.

runs. Over 95% of the instructions have PSets of size zero. These instructions either accessed only thread local memory locations, or, they were never dependent on a remote memory operation. Less than 2% of static memory instructions have a PSet of size greater than two. This result shows that the performance overhead in the executing the PSet constraint checks should be very small.

Table 5.4 lists the sizes of the applications' binaries and also the sizes of the libraries they use. It also lists the number of PSet pairs learned from the test runs. The percentage of code size increase with respect to just the application binary size is about 10% in the worst case. This is the code size increase for an application. However, the PSet pairs for an application also includes the static instructions from the libraries. The increase in binary size with respect to the total size of application and libraries is negligible. As a result, at runtime, we expect that the increase in the size of the instruction memory footprint to be also negligible. The reason for this result is that, only a small fraction of the instructions access shared-memory locations. And, only the shared-memory instructions could have a PSet of size greater than zero.

5.5 Summary

Testing and verifying a multi-threaded program is more difficult than a single-threaded program, because the number of possible interleavings is exponential over the number of memory operations executed by different threads. We make a case for an interleaving constrained shared-memory multi-processor which avoids untested interleavings.

We make the first step towards designing an interleaving constrained multi-processor. To detect untested interleavings we need a set of invariants fundamentally different from the ones used to detect incorrect interleavings such as a data race invariant or AVIO. We focused on constraining the runtime interleaving such that, no two remote memory operations are allowed to depend on each other at runtime, unless that dependency was observed in at least one of the test runs. We built a software tool to detect PSet constraints and enforce them, but as expected, it incurs significant runtime slowdown. We proposed extensions to a multi-processor design, which enables efficient detection of PSet constraint violations. On detecting a violation, checkpoint support is used for re-executing the program with an alternative interleaving and resolve the PSet constraint violations.

We analyzed several bugs in real applications, and showed that the proposed system can avoid not only data races and atomicity violations, but also other unstructured memory order related concurrency bugs. The number of false constraint violations in a well tested program is very small, and as a result, the resulting performance overhead is also negligible.

CHAPTER VI

Avoiding Untested Interleavings II: LifeTx

Chapter V takes the first step towards constraining the parallel runtime system to follow tested interleavings in order to avoid concurrency bugs in production runs. However, it cannot avoid concurrency bugs that involves multiple variables and dependencies, and it requires a custom processor design. As Hardware Transactional Memory (HTM) becoming a reality [2], one interesting research question is whether we can leverage the emerging HTM to tolerate concurrency bugs.

In this chapter, we discuss another runtime technique, called LifeTx, to tolerate concurrency bugs. This technique is based on a new type of interleavings constraint, called Lifeguard Transaction (LifeTx). LifeTxes are designed to be enforcible by HTM. This chapter is organized as follows. We first discuss LifeTx constraint and present an algorithm for deriving LifeTxes from test runs in Section 6.2. Then, we discuss our TM based hardware support for enforcing LifeTxes in Section 6.3. Finally, we present our evaluation results in Section 6.4.

6.1 Overview

We observed that in a well tested program, a programmer is likely to have tested at least the interleavings that manifest frequently when the program is executed. But many rare interleavings are likely to remain untested. Such rare untested interleavings tend to be the common cause for a majority of concurrency bugs that manifest at the production site. Given this, a parallel runtime system that avoids untested interleavings by biasing the runtime thread schedule to select a tested interleaving, whenever possible, could potentially avoid a majority of concurrency bugs from manifesting at the customer site. The performance cost of disallowing rare untested interleavings would not be significant, because though they are many in number, they are only likely to manifest infrequently (if they do manifest frequently, then they are likely to have been tested).

The key challenge in realizing an interleaving constrained parallel runtime system is in devising the right interleaving constraint that can be learned from the test runs and communicated to the runtime system. The interleaving constraint should be such that it disallows untested interleavings at runtime and avoids a majority of concurrency bugs, but at the same time it does not unnecessarily restrict parallelism by disallowing correct thread interleavings in production runs.

The Predecessor Set (PSet) constraint described in Chapter V serves this purpose. However, PSet interleaving constraints require a fairly complex hardware support to enforce them efficiently in production runs. Also, PSet constraints cannot avoid concurrency bugs due to incorrect interleaving of memory accesses to different locations such as the multi-variable atomicity violations [41]. For example, the atomicity violation bug shown in Figure 3.3 cannot be detected and avoided by PSet constraints because it involves multiple variables. As Hardware Transactional Memory (HTM) becoming a reality [2], one interesting research question is whether we can leverage the emerging HTM to tolerate concurrency bugs.

In this chapter, we present a new interleaving constraint called Lifeguard Transaction (LifeTx). LifeTxes are designed to be enforcible by HTM. They are similar to the programmer specified transactions [30, 32] in that it instructs the runtime to execute them in a serializable order. The difference is that LifeTxes are automatically derived based on interleavings observed during testing. When enforced, they are likely to avoid concurrency bugs. But, the runtime may also choose not to enforce a LifeTx constraint either because the performance is being adversely affected or to ensure forward progress. We exploit this relaxed requirement of LifeTx constraints to significantly reduce the hardware support required to enforce them.

We describe a profiling algorithm to determine LifeTxes from all the tested correct executions. Before testing a program, a thread's main function is contained in a LifeTx. Of course, this is overly constrained as a thread's entire execution needs to be serializable with respect to all the other threads. But, when a programmer tests a new interleaving for which there is no serializable execution that satisfies the current set of LifeTxes learned till that test run, we split one of the existing LifeTx such that the resulting LifeTxes are serializable for the newly tested interleaving. Thus, the newly tested interleaving would be allowed in future production runs. A programmer would test as many interleavings in a program as practically possible. The more a programmers tests, smaller and less constrained the LifeTxes will be. Once the testing is done, the LifeTx constraints are encoded in the program binary and shipped to the production site.

To enforce LifeTx constraints in production runs we propose a simple but efficient hardware support similar to the conflict detection mechanism in a hardware transactional memory (HTM) system [32]. Conflicts between concurrent LifeTxes are eagerly detected by tracking the cache blocks accessed by a LifeTx and monitoring the coherence messages. Since the runtime is not obligated to enforce the LifeTx constraint, we avoid the complexity of versioning, rollback and unbounded region support required in TM systems. Instead, we simply stall the coherence reply on detecting a conflict till one of the conflicting LifeTx finishes its execution or the stall time exceeds a predefined threshold. The threshold can be configured by the end user to make a trade-off between performance and reliability. We show that a majority of LifeTx constraints including those that encapsulate buggy code regions can be enforced by simply stalling coherence replies on detecting a conflict. The constraint violations detected during production runs and beta-testing could be logged and communicated back to the developer so that the programmer could test those interleavings and relax LifeTx constraints for future executions.

We implemented a PIN tool [46] to profile the test runs and determine LifeTxes. We studied a set of applications that includes Apache, MySQL, Parsec, and a few micro-kernels. We tested these applications as much as we can using the regression test suits and/or randomized input. Insufficient testing could result in unreasonably large LifeTxes. But, we observed that even with our less than industrial strength testing effort, LifeTxes are on the order of only a few hundred instructions in length.

To study the performance impact and bug avoidance capability, we modeled hardware support for runtime conflict detection and conflict avoidance support using Simics [47]. Thus, the simulated environment is significantly different from the test environment. Yet, we find that the runtime overhead is less than 0.6%. In the worst case, for one of the MySQL's execution, we detected only 178 constraint violations during an execution of about 2.1 billion instructions. We also analyzed 14 documented bugs in our benchmark suite, out of which 12 were atomicity violations. Out of the 12 atomicity violation bugs, 11 bugs (two of them were multi-variable atomicity violation bugs) were successfully avoided by enforcing LifeTx constraints learned during testing.

LifeTx constraints are useful for tolerating concurrency bugs in programs written using traditional form of synchronization operations such as locks, barriers, etc. We believe that they will also be useful for programs written using programmer specified transactions as well. Because, even when programmers use transactions, they could still introduce atomicity bugs by not encapsulating all the instructions that need to be atomic in a single transaction. LifeTx constraints can avoid such atomicity bugs.

In addition to helping us avoid concurrency bugs in production systems, LifeTx constraints derived from tested executed could also help improve the testing process of multithreaded programs. One common practice is to blindly stress test as many rare interleavings as possible. Instead, programmers and automatic testing tools can prioritize their efforts on exposing more interleavings for code regions contained in the larger LifeTxes. Also, LifeTx constraint violations logged during production runs could help programmers prioritize their test efforts and also determine the root cause of a program crash.

6.2 Algorithm for Determining LifeTxes

In this section we define LifeTx interleaving constraints and describe an online profiling algorithm for automatically determining those constraints from correct test runs.

6.2.1 Lifeguard Transactions (LifeTxes) and Profiling Algorithm Overview

Our goal is to tolerate concurrency bugs in production system by constraining the runtime system to execute tested thread interleavings as much as possible instead of allowing it to execute any legal interleavings permissible by user specified synchronizations. To achieve this, we require techniques to determine interleaving constraints from test runs, encode them in a program binary and enforce them at runtime. By enforcing these constraints during production runs, untested interleavings would be avoided.

We propose Lifeguard Transaction (LifeTx) interleaving constraint. Every instruction in a program is part of some LifeTx. A LifeTx constraint is similar to a programmer specified transaction [32] and is defined for a static code region (a code region is a consecutive sequence of instructions in the source program). An execution is said to satisfy the LifeTx constraints, if there exists an equivalent execution where the LifeTx protected code regions are executed in a serial total order.

We learn LifeTx constraints from correct test runs. We start from a conservative set of LifeTxes assuming that the entire execution of each thread is part of one LifeTx. Clearly, this initial constraint is too strict. As a programmer tests an execution for which there is no serializable execution of LifeTx protected code regions, we split the current set of LifeTxes. This is done by introducing what we call a cutpoint in the source program. During an execution, a cutpoint serves to terminate the previous LifeTx and then start a new LifeTx. Thus, a set of cutpoints represents the LifeTx constraints. They are encoded in the binary. As a programmer tests more and more interleavings, we progressively construct smaller and smaller LifeTxes such that the final set of LifeTxes are conflict serializable for all the tested interleavings. The runtime, which we describe in Section 6.3, would then try to enforce a serializable order between the LifeTxes during a production run.

In fact, tested interleavings are encoded in the set of learnt LifeTx constraints. By successfully enforcing these LifeTx constraints during production runs, we can ensure that if the execution of a code region was atomic in all the test executions, then it will be atomic in production runs, preventing untested unserializable interleaving with respect to this code region from manifesting. Consider the atomicity violation bug shown in Figure 3.3. In any correct execution, code regions in `Thread-1` would have executed atomically and therefore will be part of the same LifeTx. Even though, the programmer has not correctly synchronized the critical sections, our LifeTx constraints would automatically ensure that property and thereby avoid a potential concurrency bug in production runs.

The algorithm can be divided into two major parts, checking conflict serializability for the current set of LifeTxes and splitting LifeTxes when a conflict is detected, which will be addressed in the subsequent sections.

6.2.2 Checking Conflict Serializability for LifeTxes

To determine whether a test run satisfies the set of LifeTx constraints learnt until that test run, we check conflict serializability for these LifeTxes. To check conflict serializability, we construct a directed graph called conflict serializability graph. Each node in the graph represents a LifeTx. When a new LifeTx begins its execution during testing, a new node is added to the graph. All the nodes executed by a thread are connected according to the execution order. A conflict edge is added between two LifeTxes if they executed conflicting memory operations. Two concurrent memory operations executed in different threads are said to conflict if they accessed the same memory location and at least one of them is a write. Duplicated edges are not allowed. A conflict serializability violation is detected when we detect a cycle while adding a new edge to the conflict serializability graph [29]. The cycle indicates that the tested execution is not conflict serializable with respect to the LifeTxes learnt till that point.

Figure 6.1 shows a thread schedule and the corresponding conflict serializability graph. `Thread1` and `Thread2` are currently executing LifeTxes `T1` and `T2` respectively. `W(X)` represents a write to memory location `X`, and `R(X)` represents a read to memory

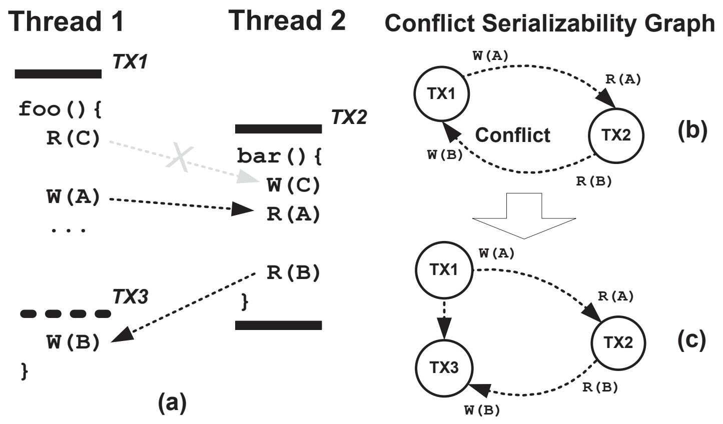


Figure 6.1: A conflict serializability violation.

location `X`. An conflict serializability violation is detected at the point of `W(B)` in `Thread1`, because a cycle is detected in the conflict serializability graph.

Conflict serializability violation checks only report a violation when there is an unserializable memory accesses. Whereas, other serializability checks such as strong strict two-phase locking do not guarantee this. We want our profiling algorithm to be conservative – do not split a LifeTx unless an real unserializable interleaving is observed during testing.

To detect conflicting memory operations, for each memory location, we maintain a data structure to store the latest write operation and the latest read operations for each thread to that location. Each time a conflict edge is added to the conflict-serializability graph, we check whether a cycle exists in the graph. Since the complexity of cycle detection is linear to the number of nodes in the graph, it is impractical to keep all the nodes in the graph. We employ an optimization [22] that removes those nodes that do not have incoming edges and the corresponding LifeTxes have terminated, because they cannot be part of any future cycle.

6.2.3 Splitting LifeTxes On a Conflict

On detecting a conflict serializability violation for the current set of LifeTxes during a test run, our tool decides to split one of the conflicting LifeTx by introducing a cutpoint into that LifeTx. The LifeTx that executed the most recently conflicting memory operation is chosen as the victim for the split.

In order to decide the location of the cutpoint in the source program, we need information about the memory accesses involved in the conflict. This is obtained by tracking conflicting memory access information on every conflict edge in the graph used for conflict serializability check. It is possible that we may detect multiple conflicts between two LifeTxes during an execution. Since we do not allow duplicated edges, we choose to maintain the most recent conflict. This could help us pinpoint the location of the cutpoint more precisely when a conflict serializability violation is detected. For example, in figure 6.1, conflict edge `R(C)` → `W(C)` is discarded when conflict `W(A)` → `R(A)` is detected.

The LifeTx chosen for the split will contain at least two memory operations that participate in the conflict-serializability violation. Otherwise, there cannot be a cycle in the graph. For example, in Figure 6.1, `Tx1` is the LifeTx that executed the most recent conflicting memory operation `W(B)`. It also contains another memory operation `W(B)` that participates in the conflict-serializability violation. The cutpoint to split `Tx1` can be introduced anywhere between these two memory accesses. We choose to always insert the cutpoint just before the last conflicting memory operation `W(B)`. After a cutpoint is inserted, we also split the corresponding node of the LifeTx in the conflict-serializability graph by terminating the current LifeTx and introducing a new LifeTx. Consider the example in figure 6.1, figure 6.1(b) shows the conflict-serializability graph before splitting, and figure 6.1(c) shows the graph after splitting.

Inserting a cutpoint in effect relaxes the atomicity constraints between all the memory operations that happen-before the cutpoint and all the memory operations that happen-after the cutpoint in a LifeTx. Ideally, we should relax the atomicity constraint only for memory operations involved in the conflict instead of inserting a cutpoint. However, that would require complex runtime hardware support for enforcing them. We leave such a design for future work.

Splitting LifeTxes Spanning Multiple Code Levels. The LifeTx chosen for splitting could span across multiple semantic segments, which requires special handling while inserting a cutpoint. All the instructions executed in a function are considered to be part of a semantic segment. Similarly, all the instructions of a loop are considered to part of another semantic segment. Instructions of an iteration of a loop together constitute a different semantic segment. Figure 6.2 shows a thread interleaving where the LifeTx `TX1` spans across two semantic segments, functions `foo()` and `bar()`. As discussed before, a cutpoint could be inserted anywhere between conflicting accesses `R(A)` and `W(C)`. Our algorithm picks the outermost semantic segment that contains the conflicting accesses, and inserts a cutpoint at the point in the source program where the next semantic segment starts. In Figure 6.2, the cutpoint is inserted just before the function call `bar()`. Inserting a cutpoint inside inner semantic segments such as `bar()` are more likely to allow interleavings that are not tested. For example, if we insert a cutpoint inside `bar` to resolve the conflict in our example, then when `bar()` is invoked from a different function, the LifeTx executed at that time would be terminated. This could prevent us from avoiding concurrency bugs. In effect, our heuristic for inserting cutpoints biases against inserting context insensitive cutpoints. The algorithm for loops and loop-iteration semantic segments is similar.

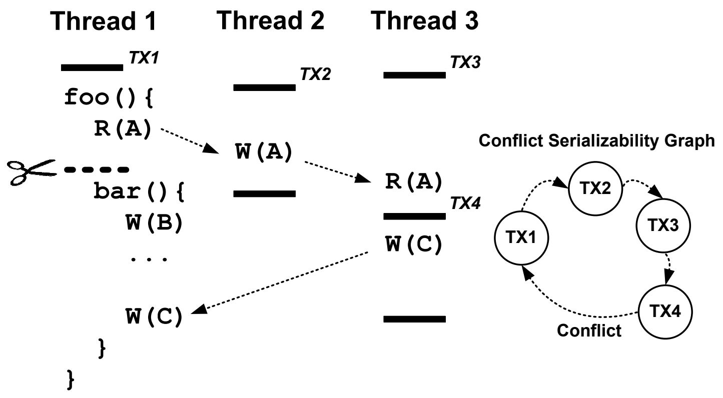


Figure 6.2: A conflict-serializability violation detected across multiple semantic-segments.

To track semantic segments, we instrument the entries and exits for each semantic segment in the program. For example, we instrument each function call and return. For loops and iterations, we statically identify loop entries, exits and back edges using goose tool [5], and then instrument them. To decide which semantic segment is the outermost one that contains the conflicting accesses, we assign a thread local counter (monotonically increasing) for each memory operation executed by the thread, and maintain a per-thread stack to track the value of the counter when the thread enters a semantic segment. By comparing the counter values of the conflicting memory operations and the counter values stored in the stack, we can easily identify the outermost semantic segment that contains conflicting memory operations.

6.2.4 Practical Issues

In this part, we discuss a few practical issues when applying our LifeTx inference algorithm to a real world multi-threaded program that is written using explicit synchronizations.

Relaxing Conflict Detection For Synchronization Functions

As we discussed in Section 6.2.3, inserting a cutpoint in effect relaxes the serializability constraints between all memory accesses before and after the cutpoint.

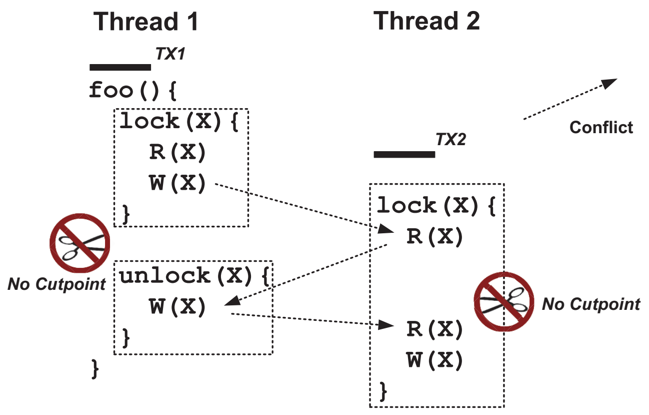


Figure 6.3: Conflicts due to memory operations executed in synchronization functions.

Ideally, we should relax the constraint only for the memory accesses that are involved in the conflict. Such an approximation will become problematic, especially when we detect conflicts for memory accesses that are inside synchronization functions. Figure 6.3 illustrates the problem. In the example, two threads are contending for the same lock. Two conflicts will be detected according to our LifeTx inference algorithm. When the same lock functions are called by a different function F, the cutpoint inserted would terminate and restart the transaction containing F as well. As a result, some concurrency bugs like the one discussed in Figure 3.3 may escape.

This problem actually exists for any function, not just for synchronization functions we showed. However, using a general way to solve the problem is very difficult. Instead, we choose to address this problem for code regions that matter the most – synchronization functions. This is because synchronization calls are interleaved heavily with other synchronization calls in remote threads.

We disable conflict detection for synchronization calls both during testing and also in production runs. This is similar to the escape actions described in [76]. Thus, we ignore the unserializable dependencies between shared-memory operations inside synchronization functions. However, we determine the happens-before relation specified by the synchronization calls, and treat a happens-before relation as a dependency between transactions during testing. That is, a conflict edge is added for each happens-before relation in the conflict serializability graph that we construct during testing. This is necessary to allow tested interleavings between code regions synchronized using traditional synchronization operations during production runs.

Optimizations for Reducing Runtime Conflict Detection Complexity

To reduce the hardware support required for conflict detection in production runs we seek to limit the number of memory locations accessed by a LifeTx. For this, we employ a simple heuristic that profiles the loops that have a trip count greater than a threshold, and introduce a cutpoint before the back edge of those loops. The intuition here is that it is unlikely that a program would require atomicity property across a loop that iterates for a long time.

Speculating past certain system calls such as network I/O could be difficult for the runtime system. We introduce a cutpoint before such difficult to speculate system calls, so that the runtime system need not support speculation past those system calls. However, this optimization is not necessary for the runtime system which does not require speculation (Section 6.3.1).

Mapping Cutpoints to Source Code

As cutpoints are gathered for a program, we map them back to the statement in the source code. If a programmer makes a simple change to the program, then all the cutpoints gathered through testing will still be valid.

6.2.5 Discussion and Limitations

Testing Correctness

Programmers have to ensure that the test runs are correct. This could be done by verifying the program output or by checking the test runs using traditional dynamic bug detection tools.

Input dependency

We use very different input for testing and measuring performance to evaluate whether our approach is sensitive to program input. Our results (Section 6.4) indicate that LifeTxes could be learnt in a few test runs and the overhead observed during a simulated production run is negligible. However, though rare, a production run could encounter frequent conflicts for some input. In such a scenario, we could turn off the LifeTx protection for performance, and the conflicting LifeTxes could be logged and communicated to the developer. These logs could assist programmers prioritize their testing effort. It is rare for an execution with incorrect interleaving to produce a correct answer. But if such a test input exist, then we may incorrectly relax constraint by splitting a LifeTx. We did not encounter this scenario in our experiments.

Context

LifeTx constraints are context insensitive. As a result, some concurrency bugs might not be avoided due to the lack of context information such as MySQL-4 in Table 6.2. We could associate context information (e.g. calling stack) with LifeTxes to address this problem, but it will require significantly more test runs to learn and require complex runtime system support.

6.3 Runtime Support for LifeTxes

We now discuss the runtime support for enforcing LifeTxes. The goal of the runtime system is to ensure that the execution of LifeTxes is conflict serializable with respect to each other during production runs. We implemented a runtime detection and avoidance support using PIN, but it slows down an execution by several orders of magnitude and therefore are unrealistic for use in a production system. To efficiently enforce LifeTxes at runtime, architectural support is a must. In this section, we first present LifeTx-Stall, an architectural design that considers hardware complexity as a first-order constraint (Section 6.3.1). It is simple and lightweight while still very effective in enforcing LifeTxes. We also discuss an ideal design, which we later use for comparison. In Section 6.4 we evaluate the performance, conflict detection and bug avoidance capability of the proposed architectural design using our Simics based simulation model.

6.3.1 LifeTx-Stall Design

LifeTx-Stall design consider hardware complexity as a first-order constraint. It is actually a simplified Hardware Transactional Memory (HTM) system without speculation and unbounded TM support. Each LifeTx is treated as a hardware transaction. On encountering a cutpoint, the processor commits the current LifeTx and starts a new LifeTx.

To enforce LifeTxes, we need to check whether the LifeTxes executed at runtime are conflict serializable. However, hardware support for checking conflict serializability violations is fairly complex [66]. Therefore, we choose to detects conflicts for LifeTxes eagerly [48]. It tracks the cache blocks read and written by a LifeTx and detects a conflict by monitoring coherence requests. Although it could unnecessarily report a conflict between two LifeTxes while they are actually conflict serializable, it simplifies the design a lot. Our results (Section 6.4.5) also indicate that the number of extra conflicts are acceptable.

To resolve a conflict, instead of using the traditional recovery mechanism used in HTM designs, we propose to use a simpler scheme. The main idea is that a processor would delay the coherence reply when the requester has a conflict with the

LifeTx currently running on it until the current LifeTx commits. To ensure forward progress, we assign a threshold for the number of cycles to wait. Such a scheme does not guarantee to enforce all the LifeTx constraints, therefore, it is a best effort scheme. We have the luxury to design such a simple system because not all LifeTx constraints need to be enforced (unlike HTM systems). Such a design avoids the need for speculation and rollback support. As a result, neither version management nor checkpointing is needed, which avoids issues in traditional HTM systems such as speculative I/O buffering. Even with such a simple design, our results (Section 6.4.5) indicate that it is effective in avoiding most of the conflicts.

LifeTx-Stall assumes a snoop bus based MESI cache coherence protocol. There are two major functionalities that LifeTx-Stall needs to support. One is to detect runtime LifeTx conflicts. The other is to resolve the detected LifeTx conflicts so that the resultant execution is conflict serializable with respect to both LifeTxes. The following sections discusses these two functionalities in detail.

Detecting LifeTx Conflicts

Maintaining Transactional Meta-data. LifeTx-Stall needs to keep track of the memory locations accessed by a LifeTx so as to detect conflicts. Each private cache block is extended with two additional bits (TX-READ and TX-WRITE), which we call transactional meta-data, indicating whether the cache block is read and written by the current LifeTx. A processor clears all the meta-data in its private cache when it commits a LifeTx. Also, when a processor timeouts waiting for a conflict to resolve, all the processors clear the meta-data in their private caches. LifeTx-Stall maintains transactional meta-data in private caches. If the cache line is evicted from the private caches, the transactional information will be lost. To alleviate this problem, we assume a small victim cache along with each private cache, which similar to the one used in many processor implementations for improving performance by indirectly increasing the associativity of caches. The replacement policy for the victim cache is LRU based, but with the exception that it always prioritizes to hold cache blocks with transactional meta-data.

Monitoring Coherence Actions. LifeTx-Stall assumes a bus based design. LifeTx-Stall detects a read-write or write-write conflict by monitoring coherence requests broadcasted on the bus and by checking its private transactional meta-data. On detecting a conflict, a processor would set a dedicated wired-OR line (similar to the wired-OR line used for detecting whether a shared copy exists or not). This is useful in efficiently resolving the conflict using the stall mechanism described in Section 6.3.1.

Relaxed Memory Accesses. LifeTx-Stall does not update the transactional meta-data for synchronization accesses (discussed in Section 6.2.4) or memory accesses from OS. This is because we are not interested in the conflicts that are caused by the relaxed synchronization accesses or OS accesses. Those accesses are called relaxed memory accesses. Although LifeTx-Stall does not update transactional meta-data for relaxed memory accesses, it still needs to check conflicts for them. In other word, a relaxed memory access can cause a conflict and a resultant processor stall. This is because a cache block might be accessed by both regular and relaxed memory accesses. A normal memory access which indeed conflicts with the remote processors may not be able to trigger a bus transaction if the read or write permission of the cache block has already been obtained by a precedent relaxed memory access to the same cache block. The simple policy we employed may cause unnecessary stalling (false positives). However, our results indicate that the number of conflicts detected at runtime is still negligible. This problem can also be addressed if the compiler can separate the data that is accessed by normal memory accesses from the data that is accessed by synchronization accesses.

Granularity. Instead of detecting conflicts at the granularity of a cache block, LifeTx-Stall can be extended to support word level conflict detection, which would avoid false conflicts. To achieve this, we need to maintain transactional meta-data for each word in a block, and associate word offset information with coherence requests. However, a few issues arise in such a design. First, the transactional meta-data of a word in a cache block may get lost when the cache block is invalidated due to a remote write to a different word of the same cache block, which would affect the bug avoidance capability of our system. For example, a processor `P1` and a processor `P2` both have a shared copy of a cache block `B` initially. `P1` reads the first word `W1` in `B`, causing the TX-READ bit of `W1` be set. Then, `P2` writes to another word `W2` in `B`. Since the conflicts are detected at the granularity of word, no conflict would be detected. At this time, `P1` invalidates its copy of `B` to service the write request, and in the process loses the TX-READ bit for `W1`. Second, the permission of a cache block could migrate to a different processor causing a potential silent conflicting access. Consider the following example: a read to a word in a block does not generate a coherence request since the read permission is obtained by a previous read to the same block but not the same word. These problems will not occur if we detect conflicts at the granularity of block. Nevertheless, our results show that the bug avoidance capability of the word based implementation is not significant (Section 6.4.4). Furthermore, we observe that detecting conflicts at the granularity of word can significantly reduce the number of false conflicts at runtime (Section 6.4.5). Therefore, we choose to use word level conflict detection in our LifeTx-Stall design.

Resolving LifeTx Conflicts

A processor detects a conflict by monitoring the coherence requests and checking its transactional data. On detecting a conflict, a processor sets a dedicated wired-OR line that can be read by all the processors. The processor that initiated the coherence request reads the wired-OR line and determines that it needs to stall and re-issue the same request after a specified time period has elapsed. The wired-OR line also helps other processors with a valid read copy to not invalidate their cache block in response to an invalidation request that got stalled. The stalled processor would re-issue the coherence request after a specified time period has elapsed. If after requests fail to get a response after a specified number of attempts, the processor issues a special coherence request that cannot be ignored. Thereby, we ensure forward progress.

To enhance the capability of resolving a conflict, we could add rollback support to the LifeTx-Stall design. However, adding rollback support will significantly increase the hardware complexity. In addition to supporting speculation, we need to keep track of the dependencies among transactions caused by the relaxed memory accesses. Each time a transaction is about to commit, it has to wait until all the dependent transactions have committed. This not only increases hardware complexity, but also could hurt performance. Since one of the goals of LifeTx-Stall design is to reduce hardware complexity, we choose not to support rollback in our LifeTx-Stall design.

6.3.2 LifeTx-CS Design

As we mentioned in Section 6.3.1, LifeTx-Stall is optimized for hardware complexity. We also studied how effective an ideal hardware conflict detection and resolution mechanism could be. This would require an ability to detect conflict-serializability violation similar to the design proposed in DATM [66], and an ability to rollback and re-execute a LifeTx, which could potentially contain system calls. We call this ideal design as LifeTx-CS.

6.4 Evaluation

In this section, we evaluate our technique from several perspectives. We first access how much testing is required to learn LifeTxes (Section 6.4.2). Then, we study the characteristics of resultant LifeTxes in terms of their footprints and lengths (Section 6.4.3). After that, we discuss the bug avoidance capability of our technique using 14 documented concurrency bugs, and its trade-offs with runtime system designs (Section 6.4.4). Finally, we evaluate the performance of two proposed runtime system designs (Section 6.4.5).

6.4.1 Experimental Setup

We built our profiling tool using the PIN [46] binary instrumentation infrastructure. To study the runtime system, we modeled LifeTx-Stall design in Simics [47], a full system simulator. We also built a PIN based simulator to model LifeTx-CS design. All the experiments are conducted on a Quad-Core Dell T3400 workstation, with a 64-bit Redhat Enterprise Linux 5 on it.

Benchmarks

Two sets of benchmarks are used throughout our evaluation. The first set is called *Bug-Bench*, which is used to study the bug avoidance capability of our technique. Ideally, we would like to evaluate all the documented bugs that were available at the time we performed our experiment. However, setting up environment for reproducing each bug is difficult and time consuming because we may need to install a specific version of the program which sometimes requires installing an old kernel. Therefore, we decided to evaluate a subset of them. We mainly focus on evaluating atomicity violation bugs because LifeTx is optimized for tolerating atomicity violation bugs and does not actively seek to avoid other types of concurrency bugs.

Table 6.2 lists the 14 documented concurrency bugs we have studied, The names shown in Column-2 match that shown in Table 3.5 in which a short description of each bug is provided. Among these 14 bugs, 8 of them are *Bug Kernels*, which are code snippets extracted from real buggy programs. Some program details might be omitted in *Bug Kernels*. The remaining 6 bugs are *Real Bugs*. For these real bugs, we use the original programs, and study their real executions.

The second set of benchmarks, called *Perf-Bench*, is used to evaluate LifeTx characteristics and the runtime performance. It consists of selected parallel benchmarks from Splash2 [81] (`fft`, `radix`, `fmm` and `ocean`) and Parsec [10] (`blackscholes`, `canneal`), and several widely used multi-threaded applications (`pbzip2`, `pfscan`, `mysql`, `apache`). Notice that the versions of `mysql`, `apache` and `pbzip2` used in *Perf-Bench* are the same as that of `Bug #10`, `Bug #9` and `Bug #14` used in *Bug-Bench*. The corresponding testing methods for them are the same, saving us a little testing effort.

Testing Methodology

LifeTxes are learned from correct test runs. We build our profiling tool using PIN binary instrumentation tool. Our profiling tool does not require source code. We use goose [5], a PIN based tool, to automatically extract loop information from program binaries.

We perform testing for both *Bug-Bench* and *Perf-Bench*. For scientific programs in Splash2 and Parsec, randomly generated parameters are used (e.g. matrix size, number of threads, etc.) in each test run. For each MySQL benchmark (`Bug #10` to `Bug #13`, `mysql`), we use selected tests from the regression test suite that is shipped with MySQL source code. Also, we run OSDB [6] multi-user test in parallel with the regression test to create a parallel environment. Each version of MySQL is tested individually. For `Bug #9` and `apache`, we use httperf tool to generate concurrent requests to a set of html files. For `Bug #14` and `pbzip2`, we compress a randomly chosen files using random number of threads in each test run. For `pfscan`, we search a random string from a large collection of random files or directories in each test run. Finally, for all the bug kernels, we use random inputs (e.g. random loop count, random strings, etc.) and random number of threads.

Simulation Methodology

We use a simulator to study the proposed runtime system behavior. We built two simulators, LifeTx-Stall and LifeTx-CS. We model LifeTx-Stall in Simics. We extend the *g-cache* timing model in Simics. The configurations of the baseline system is listed in Table 6.1. The coherent caches are based on MESI protocol, and is implemented on a snoop bus. We model all the features we discussed in Section 6.3.1. Conflicts are detected at the granularity of word. We assume 1-cycle LifeTx commit latency and 100-cycle give up latency. The timeout threshold is set to 50K cycles.

We model another proposed design, LifeTx-CS, using PIN binary instrumentation infrastructure. The simulator models in-place memory update. The transactional undo logs are kept in the main memory as LogTM [48] does. The modeled system tests conflict serializability for LifeTxes, which is similar to MetaTM [66] does. Conflicts are detected at the granularity of word. The simulator also supports rollback and re-execution.

The inputs used in simulation are different from those used in testing. For benchmarks in Splash2 and Parsec, we use a set of input parameters that are not used during testing. For `pbzip2`, we compress a new file. For `pfscan`, we search a ran-

Processor	4 cores, 2.0GHz, in-order
L1 Cache	Private, 64KB I-cache, 64KB D-cache, 4-way set associative, 32B block size, 3-cycle latency, write-back, 1KB fully associative victim cache
L2 Cache	Shared, 8MB, 8-way set associative, 128B block size, 15-cycle latency, write-back
Main Memory	2GB DRAM, 200-cycle access
Interconnect	Bus based, latency not modeled

Table 6.1: Baseline configuration.

domly generated string from a different directory. For all the MySQL benchmarks in *Bug-Bench*, we use their bug triggering inputs. For `mysql` in *Perf-Bench*, we use OSDB to generate concurrent requests to a newly created database. For `Bug #9` and `apache`, we generate concurrent requests to a different set of html files (which could trigger the bug). Notice that the inputs used in the PIN based simulator are not identical to that used in the Simics based simulator.

6.4.2 Learning LifeTxes

To access the time required to learn LifeTxes, we perform testing for all the benchmarks using the methods described in Section 6.4.1. Figure 6.4 shows the testing results for all the *Perf-Bench* (*Bug-Bench* results are similar). Each point along the x-axis represents a unique test run [1], and the y-axis represents the cumulative total number of cutpoints learned after a particular test run. As the figure shows, our profiling algorithm reaches a stable state reasonably fast. That shows that the testing process during the normal software development should be adequate for our purpose.

6.4.3 Characteristics of LifeTxes

Once we have obtained LifeTxes from testing, the most important question to answer is how "big" each LifeTx is. Most hardware transactional memory systems

[1]For `mysql`, one test run is a 12min run of the workload we described before. For `apache`, one test run means a session of concurrent requests generated by httperf.

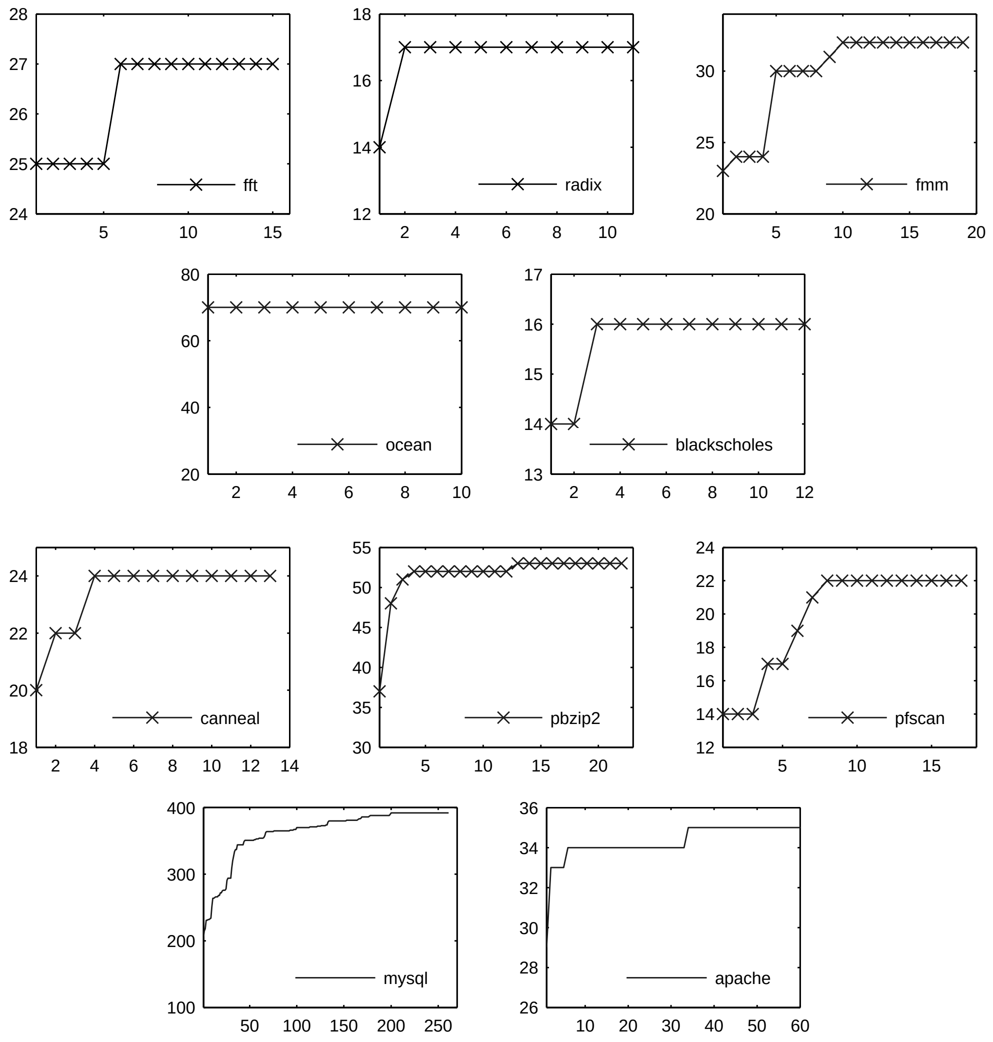


Figure 6.4: Number of test runs required for getting stable cutpoints.

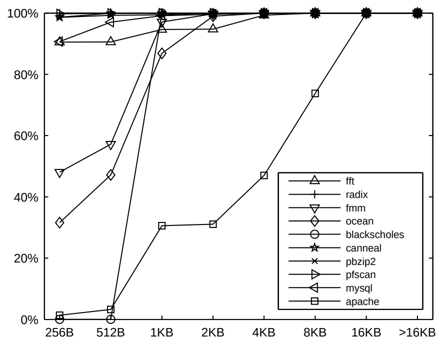


Figure 6.5: Footprint distribution (dynamic).

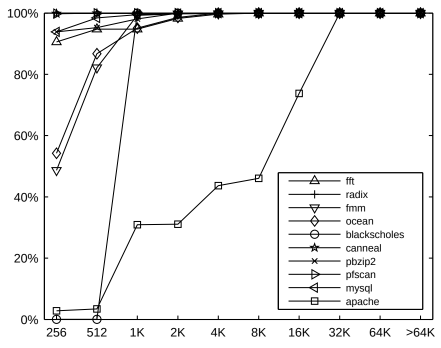


Figure 6.6: Length distribution (dynamic).

have limits on the size of each transaction and even for unbounded transactional memory systems, supporting large transactions is not efficient.

We use our PIN based simulator (described in Section 6.4.1) to obtain the memory footprints and lengths (measured in terms of number of dynamic instructions executed) of each LifeTx. All the results are the cumulative results of 10 simulation runs. Figure 6.6 and Figure 6.5 shows the results for *Perf-Bench*. Figure 6.6 shows the length distribution of all the committed dynamic LifeTxes. As shown in the figure, for all benchmarks (except `apache`), over 95% of the dynamic LifeTxes have lengths less than 1K instructions. Although the LifeTxes in `apache` are longer than that in other benchmarks, almost all of its LifeTxes are less than 32K instructions. Figure 6.5 shows the memory footprint distribution of all committed dynamic LifeTxes. Similarly, we observe that for almost all the workload (except `apache`), over 90% of the dynamic LifeTxes have memory footprint less than 1KB. And almost all the LifeTxes we observed have memory footprint less than 16KB. These results indicate that most of the LifeTxes are reasonable in size, both in terms of length and memory footprint. Therefore, LifeTxes can be supported efficiently because most of the time the execution of a LifeTx will not exceed the hardware resource limit. It is only for a few LifeTxes hardware resource limit would be reached.

6.4.4 Bug Avoidance Capability

There are several factors that contribute to the bug avoidance capability of our technique. We classify these factors into two major groups. One is coming from testing, and the other is from runtime systems.

Testing Impact

We infer LifeTxes from testing. For an atomicity violation bug, whether it can be avoided or not depends on whether the inferred LifeTxes enclose the critical path – the code execution path that need to be atomic but is not enforced by the program. In other words, to test the bug avoidance capability, we can check whether there exist cutpoints inside a critical path. If not, our runtime system will make best effort to enforce the atomicity of that critical path, avoiding the atomicity violation bug.

For each bug in *Bug-Bench*, we check each inferred cutpoint after testing is finished. We check to see whether there exists any cutpoint in the critical path. If not, the concurrency bug is under LifeTx protection. We compare LifeTx with data-race detectors and PSet [84] based systems. Table 6.2 shows the results. Among the 14 bugs we have analyzed, 12 of them are atomicity violation bugs. Out of these 12 atomicity violation bugs, LifeTx provides protection for 11 of them, including not only data-race free atomicity violations (`Bug #1`, `Bug #2` and `Bug #3`), but also multi-variable atomicity violation bugs (`Bug #3` and `Bug #12`). Those bugs cannot be easily avoided by traditional data race and atomicity violation surviving techniques.

Our technique cannot avoid `Bug #13`. That is because a cutpoint is found in a function which is called in the critical path. During testing, this function is found to be not serializable under a different calling context. In order to solve this problem, we could associate context information with each cutpoint. However, that will

	ID	Bug Name	Type	LifeTX Protection	Data Race Detector	PSet Protection
Kernel	1	BankAccount	Single-A.V.	Yes	No	Yes
	2	CircularList	Single-A.V.	Yes	No	Yes
	3	StringBuffer	Multi-A.V.	Yes	No	No
	4	LogProcSweep	Race, Single-A.V.	Yes	Yes	Yes
	5	Mozilla-2	Race, Single-A.V.	Yes	Yes	Yes
	6	Mozilla-3	Race, Single-A.V.	Yes	Yes	Yes
	7	Mozilla-4	Race, Single-A.V.	Yes	Yes	Yes
	8	Mozilla-9	Order Vio.	No	No	Yes
Real	9	Apache-1	Race, Single-A.V.	Yes	Yes	Yes
	10	MySQL-3	Race, Single-A.V.	Yes	Yes	Yes
	11	MySQL-5	Race, Single-A.V.	Yes	Yes	Yes
	12	MySQL-4	Race, Multi-A.V.	Yes	Yes	No
	13	MySQL-1	Multi-A.V.	No	No	No
	14	Pbzip2	Race, Order Vio.	No	Yes	Yes

Table 6.2: Bug avoidance capability.

significantly increase the runtime system complexity.

Runtime System Influences

Even if the critical path is under LifeTx protection, it is possible that the concurrency bug is not avoided since we assume a best-effort runtime system.

To avoid concurrency bugs, the runtime system must be able to detect them first. Usually, for atomicity violation bugs, a transactional conflict will be detected when the bug is triggered. However, under some circumstances, a runtime system might not be able to detect a bug. There are two major causes: 1) *Loss of transactional meta-data*, and 2) *Permission migration*. The problem has already been discussed in Section 6.3.1. Table 6.3 shows the characteristics of the four critical LifeTxes (for the four real bugs that have LifeTx protection) that are collected from LifeTx-Stall simulation (critical LifeTxes are the LifeTxes that contain the critical sections). In the table, the second column shows how many times a particular critical LifeTx gets executed in the simulation workload, and all the other columns show the average number for each LifeTx instance. As can be observed from the table, most of the time, critical LifeTxes have moderate memory footprint and do not suffer meta-data loss or permission migration problems. The statistics shown in table 6.3 are for all executions of the critical LifeTx. Triggering a real concurrency bug in Sim-

ID	Name	# Conflicts		Timeout	Footprint (# Blks)	Inst. Cnt.	Meta Loss (# Blks)	Migration (# Blks)
		# Instance	Resovled					
9	Apache-1	362	0.0	0.0	126.1	4215.6	0.0	0.0
10	MySQL-3	2	0.0	0.0	141.5	2296.5	0.0	0.0
11	MySQL-5	264	1.0	0.0	158.7	17405.9	1.0	2.0
12	MySQL-4	3	0.0	0.0	62.0	6137.7	0.0	0.0

Table 6.3: Characteristics of critical LifeTxes.

ics is challenging. However, we successfully managed to trigger `Bug #10` in Simics by implementing CTrigger [59] algorithm, and the bug was successfully avoided by LifeTx-Stall design.

Even if a concurrency bug (the conflict) can be detected, it is possible that the runtime system cannot resolve it. For LifeTx-Stall design, since it does not have rollback support, some bugs may not be avoided. One example is two concurrent read-modify-writes race for a shared variable, stalling on either write will not resolve the bug. Even with rollback support (such as LifeTx-CS design), some bugs still may not be avoided. For example, two LifeTxes may have cyclic dependency, preventing each other from making forward progress. However, we argue that under such cases, the bug is less likely to be an atomicity violation bug since no valid execution can be found to be serializable with the code region that is intended to be atomic.

6.4.5 Performance Study

Finally, we study the performance of our runtime system. We evaluate both LifeTx-Stall and LifeTx-CS designs. We model LifeTx-Stall in Simics. For each benchmark in *Perf-Bench*, we start the simulation from the middle of the program execution, usually from the program point after which all worker threads have been created (e.g. after the first barrier). Table 6.4 shows the simulation statistics. We find that most of the LifeTx conflicts can be resolved by just stalling one conflicting thread. In worst case (`pfscan`), only 4 timeouts are reported. We also list the average waiting cycles for resolved conflicts. Except for the conflicts in `pfscan`, most of the

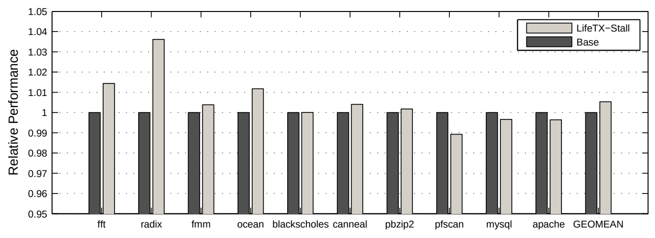


Figure 6.7: LifeTx-Stall performance overhead.

conflicts can be resolved in 2000 cycles. `pfscan` stalls longer than others because it spends relatively more time in OS handling I/O operations which take longer time to finish. The percentage of total waiting cycles (with respect to total simulated cycles) for each benchmark is negligible, indicating small runtime overhead. Figure 6.7 compare the performance between the baseline system and LifeTx-Stall system. The average overhead is less than 0.6%, which is negligible.

We also model LifeTx-CS runtime system design using a PIN based simulator. The runtime performance overhead of LifeTx-CS can be broken down into two parts. The first part is the cost in supporting a TM system (e.g. version management and conflict detection). However, post work in hardware TM [66] has shown that such cost can be minimized by using hardware support. The second part of the overhead is determined by the runtime conflicts. Each runtime conflict would result in LifeTx abort and re-execute instructions. We measured the number of aborts and the number of additional instructions that need to be executed due to those aborts. Table 6.4 shows the number of runtime conflicts (or false conflicts) and the number of aborted instructions. For example, for `mysql`, we detect 27 constraint violations for a program that executed about 800 million instructions, which resulted in re-execution of 0.019% of instructions. And for the worst case (`pfscan`), only 0.027% of the instructions are re-executed. Therefore, the runtime overhead due to transactional abort and re-execution is negligible. Notice that there are a few conflicts which

Programs	LifeTx-Stall					LifeTx-CS			
	# Conflicts		Avg. Resolved Wait Cycles	% Waiting Overhead	Inst. Cnt.	# Conflicts		% Re-exec Inst.	Inst. Cnt.
	Resolved	Timeout				Resolved	w/ Syscall		
fft	0	0	0	0.000%	543M	0	0	0.000%	130M
radix	0	0	0	0.000%	35M	0	3	0.013%	360M
fmm	178	0	631	0.125%	225M	1	2	0.000%	2270M
ocean	141	0	1116	0.563%	44M	0	0	0.000%	339M
blackscholes	0	0	0	0.000%	92M	0	0	0.000%	810M
canneal	0	0	0	0.000%	16M	32	2	0.007%	45M
pbzip2	0	0	0	0.000%	658M	0	40	0.003%	892M
pfscan	13	4	12288	1.407%	43M	1	2	0.027%	13M
mysql	178	3	470	0.002%	2.1B	23	4	0.019%	794M
apache	0	0	0	0.000%	444M	0	1	0.017%	374M

Table 6.4: Runtime statistics for (a) LifeTx-Stall, (b) LifeTx-CS.

Programs	Block		Word	
	Conflict Resolved	Timeout	Conflict Resolved	Timeout
fft	0	0	0	0
radix	8	1	0	0
fmm	255	7	178	0
ocean	757	125	141	0
blackscholes	0	0	0	0
canneal	0	0	0	0
pbzip2	0	1	0	0
pfscan	19	13	13	4
mysql	22	41170	178	3
apache	0	0	0	0

Table 6.5: Granularity of conflict detection: Block vs Word.

we cannot resolve. This is because our current PIN based implementation of our simulator cannot undo the effect of certain system calls. However, this could be resolved by integrating operating system support [61].

Block vs. Word

To decide the conflict detection granularity for LifeTx-Stall, we implemented both versions. Table 6.5 shows the comparison results. For each design, we simulate the same number of instructions. As we can see, word level conflict detection reports far less conflicts than the block level counterpart. For example, for `mysql`, block based implementation reports 41192 conflicts while the word based implementation only reports 181 conflicts. Another interesting finding is that block based implementation is more likely to cause an timeout than the word based implementation. This is reasonable since our profiling tool detects conflicts at the granularity of word. Detecting conflict on block at runtime introduces more dependencies than that seen during testing, leading to more timeouts. Therefore, we choose to detect conflicts at the granularity of word.

6.5 Summary

As we enter the multi-core era, providing support for developing reliable parallel programs is crucial. Most of the concurrency bugs manifest when a rare interleaving manifests in a production run. The traditional approach has been to test the program as much as possible and try to expose as many rare interleavings as possible. While testing for corner cases in single-threaded programs is absolutely necessary, it not so necessary for multi-threaded programs, especially given the fact that there are too many of those corner cases. Our approach is based on the insight that we can exploit the inherent non-determinism in parallel systems, and use it to our advantage. That is, bias the non-deterministic thread schedule to pick a tested interleaving as much as possible.

Sun has already incorporated hardware support for transactions. The proposed design could be simpler than a conventional HTM support. We proposed an algorithm for automatically deriving Lifeguard transactions from tested executions. We showed that the performance overhead of our stall based mechanism is negligible, and that we can avoid 11 out of 12 atomicity violations in programs like MySQL and Apache.

CHAPTER VII

Future Work

Parallel programming becomes increasingly important nowadays due to the prevalence of multi-core hardware. As it is inherently more difficult than sequential programming, we are in great need of tools and techniques that can help programmers build more reliable concurrent software. In this dissertation, we have made several contributions towards this goal. However, it is still a relatively open area and many problems remain unsolved. In this chapter, we discuss our future work.

Among all the possible future directions, one that interests us most is to develop tools and algorithms for improving the reliability of concurrent programs running on mobile platforms. Mobile computers are becoming increasingly important. The number of mobile devices has already exceeded 1 billion. For many people, smartphones or tablets are the primary platform for interacting with computer systems. Driven by the need of running increasingly sophisticated software, mobile computers are becoming more and more powerful with more and more cores being added. Mobile applications, thereby, need to be concurrent to better utilize the multi-core processors in modern smartphones. They also need concurrency to process event streams from a rich array of sensors.

Mobile applications are commonly programmed using an event-based concurrency model. This model, used in popular platforms such as Android, arises naturally on mobile devices that have a rich array of sensors and user input modalities. In an event-based program, there exists at least one thread which continuously polls an event queue to dequeue events and executes the handlers associated with those event. Such type of threads are called Looper threads [3]. For example, in Android, the UI thread (a.k.a. the main thread) is a Looper thread. It processes all events generated by the user (e.g., button clicks). The computation in a Looper thread is structured as a sequence of event callbacks invoked to process the events received by that thread.

One interesting aspect about event-based programs is that event handlers invoked by the same Looper thread can be *logically concurrent* even if they are always executed in order. For example, two events being posted into an event queue by two independent threads may be processed in different orders by a Looper thread in different executions. These two event handlers are therefore logically concurrent in the sense that a programmer should not rely on a particular execution order between them.

This unique feature in event-based programs may cause problems for existing concurrent programming tools which always assume that operations executed by the same thread cannot be concurrent. For instance, a happens-before date race detector [21] assumes a happens-before relation between any two operations executed by the same thread. Though working well for thread-based programs, this assumption is too conservative for event-based programs.

Alternatively, one could, for the purpose of analysis, transform an event-based program into a multi-threaded program by regarding the handling of each event as a separate, short-lived thread. However, we argue that such a reasoning of concurrency for event-based programs is also not precise and overly aggressive. One problem is that it neglects the ordering constraints that could be imposed by an event queue. For example, if the event queue of a Looper thread uses a first-in-first-out (FIFO) dequeue policy, a later added event will not be processed until all the previously added events are processed, and a programmer may rely on this invariant to establish orders between event handlers. Another problem is that the model assumes any two event handlers can interleave as if they are two independent threads. However, in the case where two event handlers are invoked by the same thread, they need to be mutually exclusive.

Therefore, we propose to redefine the causality, which is the *happens-before* ordering, for event-based programs. This is essentially the key for many concurrent programming tools such as data race detectors. Our main idea is to relax the happens-before ordering caused by the program order between event handlers executed by a single Looper thread. The reason is that one should not assume that two event handlers are ordered simply because they appear sequentially in the same Looper thread. However, it would also be incorrect to assume that there is never any order between event handler. To solve that, we introduce a few rules for defining causality for event-based programs. Our initial work has defined the following causal orders:

- All instructions in a regular thread (i.e. non-Looper thread) are causally ordered by the program order. Instructions within an event handler are also causally ordered by the program order.
- We account for all causal orders due to any resource release and the subsequent resource acquire. The resource here can be any kind of resource such as synchronization variable, pipe, socket, file or user defined resource.
- Two event handlers in a Looper thread are ordered if the instructions that created them are ordered. This assumes that the events posted will be processed

in FIFO order. Where alternate orders are employed, we will need to develop and substitute other ordering relations.

- Event handlers executed in a looper thread are atomic with respect to one another. This assumes that a looper thread always completes the execution of an event handler before starting another handler.

In the future, we plan to generalize these rules to handle a broad set of event-based programs. For example, some event-based program use a time-based event queue in which each event is associated with an earliest dispatch time, some uses a priority-based event queue in which each event is associated with a priority, and some uses a thread pool to serve an event queue. We would like to handle all such cases. Our idea is to introduce a general event handling model such that all types of event-based programs can be equivalently mapped to that model, and we will extend our current rules to capture its causality.

Base on the new causal model, we propose a few tools for finding and tolerating concurrency bugs for event-based programs. One promising direction is to detect data races. We propose to analyze an execution of a program and predict data races that might exist in the program. We say that two operations in an execution over an input are conflicting if there is only one correct order of execution between them for that input. Two conflicting operations are racy if there is no happens-before order between them, where the happens-before order is determined according to the new causal model. One problem we need to address in the future is false positives. We expect more false positives under the new causal model since races can happen even between operations within the same thread. Another challenge is to design an algorithm to efficiently check happens-before order between operations because directly applying the traditional vector clock based algorithm does not scale.

Another promising direction is to actually expose concurrency bugs in real executions. For example, we can employ a similar technique described in RaceFuzzer [69] to expose concurrency bugs. The main idea is to first detect races as we discussed above for a given input, and then produce alternative executions with the same input by injecting extra delays at certain points during the executions according to the race report, hoping to actually expose concurrency bugs in an event-based program. We can also use a technique similar to that described in Chapter IV to expose concurrency bugs. The key challenge is to devise a new way to encode tested interleavings which is suitable for event-based programs.

Finally, we can build runtime techniques to tolerate concurrency bugs in event-based programs. In fact, we envision that the bug avoidance technique for event-based programs is more likely to be successful than that for traditional thread-based programs as we discussed in Chapter V and Chapter VI. The reason is because an event-based program is executed in such a regulated way that it is much easier to control its runtime behaviors to bypass certain bad interleavings. For example, we can constrain the order in which event handlers are invoked by using an efficient and software only solution to prevent untested event dispatching orders from manifesting in production runs. Such a software only solution is more likely to be adopted by customers. Again, the key challenge is a way to encode tested interleavings that is optimized for event-based programs.

CHAPTER VIII

Conclusion

Shared-memory multi-threaded programming is inherently more difficult than single-threaded programming. The main source of complexity is that, the threads of an application can interleave in so many different ways. To ensure correctness, a programmer has to test all possible thread interleavings, which, however, is impractical. Many rare thread interleavings remain untested in production systems, and they are the major cause for a majority of concurrency bugs.

This dissertation discusses two ways to tame concurrency bugs. The first way is to expose untested interleavings during testing so that we can discover concurrency bugs in a program before it is shipped to customers. However, the interleavings space of a multi-threaded program is usually so huge that we cannot practically expose all untested interleavings. For the remaining untested interleavings, we propose to avoid them during production runs such that most of the concurrency bugs can be tolerated.

The central part of this dissertation is an efficient and effective way to encode and remember tested interleavings. We make two hypotheses about concurrency bugs: the small scope hypothesis and value independent hypothesis. Based on these two hypotheses, we define a set of interleaving idioms which we use to encode tested interleavings. Our empirical analysis shows that such an encoding scheme is able to capture most of the concurrency bugs we have analyzed.

Based on the set of interleaving idioms, we build a testing tool, called Maple, that seeks to expose untested interleavings as much as possible. It memoizes tested interleavings and actively seeks to expose untested interleavings in order to expose more interleavings faster. Our experience in using Maple to test real-world applications shows that Maple is able to trigger concurrency bugs faster by exposing more untested interleavings in a shorter period of time than other conventional testing techniques.

For inevitable untested interleavings of a program, we propose two runtime techniques for avoiding them during production runs to tolerate concurrency bugs. First, we propose a customized shared memory multi-processor design for tolerating concurrency bugs. It is optimized for the simplest interleaving idiom which only involves one inter-thread dependency. We encode the tested interleavings in a program's binary executable using Predecessor Set (PSet) constaints. These constraints are efficiently enforced at runtime using processor support, which ensures that the runtime follows a tested interleaving. We analyze several bugs in open source applications such as MySQL, Apache, Mozilla, etc., and show that, by enforcing PSet constraints, we can avoid not only data races and atomicity violations, but also other forms of concurrency bugs.

Then, we discuss another hardware design for tolerating atomicity violation bugs. It is based on a new interleaving constraints called lifeguard transaction (LifeTx). LifeTx constraints can be efficiently enforced by a new hardware design similar to the eager conflict detection capability that exist in a conventional hardware transactional memory (TM) systems, but without the need for versioning, rollback and unbounded

TM support. We show that 11 out of 14 real concurrency bugs in programs like Apache, MySQL and Mozilla could be avoided using the proposed approach for a negligible performance overhead.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Collection of Concurrency Bugs. `http://www.eecs.umich.edu/~jieyu/bugs.html`.

[2] Haswell Microarchitecture. `http://en.wikipedia.org/wiki/Haswell_(microarchitecture)`.

[3] Looper Thread in Android. `http://developer.android.com/reference/android/os/Looper.html`.

[4] Maple Open Source Project. `https://github.com/jieyu/maple`.

[5] The goose tool. `http://systems.cs.colorado.edu/~moseleyt/goose/`.

[6] The Open Source Database Benchmark. `http://osdb.sourceforge.net/`.

[7] Agarwal, R., Sasturkar, A., Wang, L., and Stoller, S. D. Optimized Run-time Race Detection and Atomicity Checking Using Partial Discovered Types. In *ASE* (2005), pp. 233–242.

[8] Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. Unbounded Transactional Memory. In *HPCA* (2005), pp. 316–327.

[9] Barford, P., and Crovella, M. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *SIGMETRICS* (1998), pp. 151–160.

[10] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT* (2008), pp. 72–81.

[11] Bron, A., Farchi, E., Magid, Y., Nir, Y., and Ur, S. Applications of Synchronization Coverage. In *PPOPP* (2005), pp. 206–212.

[12] Burckhardt, S., Kothari, P., Musuvathi, M., and Nagarakatte, S. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS* (2010), pp. 167–178.

[13] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI* (2008), pp. 209–224.

[14] Ceze, L., Tuck, J., Montesinos, P., and Torrellas, J. Bulksc: bulk enforcement of sequential consistency. In *ISCA* (2007), pp. 278–289.

[15] Coons, K. E., Burckhardt, S., and Musuvathi, M. GAMBIT: Effective Unit Testing for Concurrency Libraries. In *PPOPP* (2010), pp. 15–24.

[16] Corliss, M. L., Lewis, E. C., and Roth, A. DISE: a Programmable Macro Engine for Customizing Applications. In *ISCA* (2003), pp. 362–373.

[17] Devietti, J., Lucia, B., Ceze, L., and Oskin, M. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS* (2009), pp. 85–96.

[18] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., and Ur, S. Multithreaded Java program test generation. *IBM Systems Journal 41*, 1 (2002), 111–125.

[19] Engler, D. R., and Ashcraft, K. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP* (2003), pp. 237–252.

[20] Flanagan, C., and Freund, S. N. Atomizer: a Dynamic Atomicity Checker for Multithreaded Programs. In *POPL* (2004), pp. 256–267.

[21] Flanagan, C., and Freund, S. N. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI* (2009), pp. 121–133.

[22] Flanagan, C., Freund, S. N., and Yi, J. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI* (2008), pp. 293–303.

[23] Flanagan, C., and Godefroid, P. Dynamic Partial-order Reduction for Model Checking Software. In *POPL* (2005), pp. 110–121.

[24] Flanagan, C., and Qadeer, S. A Type and Effect System for Atomicity. In *PLDI* (2003), pp. 338–349.

[25] Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, vol. 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[26] Godefroid, P. Model Checking for Programming Languages using Verisoft. In *POPL* (1997), pp. 174–186.

[27] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *PLDI* (2005), pp. 213–223.

[28] Godefroid, P., Levin, M. Y., and Molnar, D. A. Automated Whitebox Fuzz Testing. In *NDSS* (2008).

[29] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[30] Hammond, L., Carlstrom, B. D., Wong, V., Hertzberg, B., Chen, M. K., Kozyrakis, C., and Olukotun, K. Programming with Transactional Coherence and Consistency (TCC). In *ASPLOS* (2004), pp. 1–13.

[31] Havelund, K., and Pressburger, T. Model Checking JAVA Programs using JAVA PathFinder. *STTT 2*, 4 (2000), 366–381.

[32] Herlihy, M., and Moss, J. E. B. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA* (1993), pp. 289–300.

[33] Huang, J., and Zhang, C. Persuasive prediction of concurrency access anomalies. In *ISSTA* (2011), pp. 144–154.

[34] Jackson, D., and Damon, C. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. In *ISSTA* (1996), pp. 239–249.

[35] Jagannath, V., Gligoric, M., Jin, D., Luo, Q., Rosu, G., and Marinov, D. Improved Multithreaded Unit Testing. In *SIGSOFT FSE* (2011), pp. 223–233.

[36] Joshi, P., Park, C.-S., Sen, K., and Naik, M. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *PLDI* (2009), pp. 110–120.

[37] Kahlon, V., and Wang, C. Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In *CAV* (2010), pp. 434–449.

[38] Krena, B., Letko, Z., and Vojnar, T. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In *RV* (2011).

[39] Lai, Z., Cheung, S.-C., and Chan, W. K. Detecting Atomic-set Serializability Violations in Multithreaded Programs Through Active Randomized Testing. In *ICSE (1)* (2010), pp. 235–244.

[40] Lu, S., Jiang, W., and Zhou, Y. A Study of Interleaving Coverage Criteria. In *ESEC/SIGSOFT FSE* (2007), pp. 533–536.

[41] Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R. A., and Zhou, Y. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *SOSP* (2007), pp. 103–116.

[42] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from Mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS* (2008), pp. 329–339.

[43] Lu, S., Tucek, J., Qin, F., and Zhou, Y. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS* (2006), pp. 37–48.

[44] Lucia, B., and Ceze, L. Cooperative Empirical Failure Avoidance for Multithreaded Programs. In *ASPLOS* (2013), pp. 39–50.

[45] Lucia, B., Devietti, J., Strauss, K., and Ceze, L. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA* (2008), pp. 277–288.

[46] Luk, C.-K., Cohn, R. S., Muth, R., Patil, H., Klauser, A., Lowney, P. G., Wallace, S., Reddi, V. J., and Hazelwood, K. M. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI* (2005), pp. 190–200.

[47] Magnusson, S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A Full System Simulation Platform. *IEEE Computer 35*, 2 (2002), 50–58.

[48] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. LogTM: Log-based Transactional Memory. In *HPCA* (2006), pp. 254–265.

[49] Musuvathi, M., and Qadeer, S. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI* (2007), pp. 446–455.

[50] Musuvathi, M., and Qadeer, S. Partial-Order Reduction for Context-Bounded State Exploration. Tech. Rep. MSR-TR-2007-12, Microsoft Research, 2007.

[51] Musuvathi, M., and Qadeer, S. Fair Stateless Model Checking. In *PLDI* (2008), pp. 362–371.

[52] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI* (2008), pp. 267–280.

[53] Naik, M., Park, C.-S., Sen, K., and Gay, D. Effective Static Deadlock Detection. In *ICSE* (2009), pp. 386–396.

[54] Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., and Calder, B. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI* (2007), pp. 22–31.

[55] Netzer, R. H. B. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Workshop on Parallel and Distributed Debugging* (1993), pp. 1–11.

[56] Olszewski, M., Ansel, J., and Amarasinghe, S. P. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS* (2009), pp. 97–108.

[57] Pacheco, C., and Ernst, M. D. Randoop: Feedback-directed Random Testing for Java. In *OOPSLA Companion* (2007), pp. 815–816.

[58] Park, C.-S., and Sen, K. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *SIGSOFT FSE* (2008), pp. 135–145.

[59] Park, S., Lu, S., and Zhou, Y. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS* (2009), pp. 25–36.

[60] Patil, H., Pereira, C., Stallcup, M., Lueck, G., and Cownie, J. PinPlay: a Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *CGO* (2010), pp. 2–11.

[61] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating Systems Transactions. In *SOSP* (2009), pp. 161–176.

[62] Prvulovic, M., and Torrellas, J. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *ISCA* (2003), pp. 110–121.

[63] Prvulovic, M., Torrellas, J., and Zhang, Z. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *ISCA* (2002), pp. 111–122.

[64] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating Bugs as Allergies - a Safe Method to Survive Software Failures. In *SOSP* (2005), pp. 235–248.

[65] Rajamani, S. K., Ramalingam, G., Ranganath, V. P., and Vaswani, K. ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs. In *ASPLOS* (2009), pp. 181–192.

[66] Ramadan, H. E., Rossbach, C. J., and Witchel, E. Dependence-aware Transactional Memory for Increased Concurrency. In *MICRO* (2008), pp. 246–257.

[67] Ratanaworabhan, P., Burtscher, M., Kirovski, D., Zorn, B. G., Nagpal, R., and Pattabiraman, K. Detecting and Tolerating Asymmetric Races. In *PPOPP* (2009), pp. 173–184.

[68] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. Eraser: a Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP* (1997), pp. 27–37.

[69] Sen, K. Race Directed Random Testing of Concurrent Programs. In *PLDI* (2008), pp. 11–21.

[70] Sherman, E., Dwyer, M. B., and Elbaum, S. G. Saturation-based Testing of Concurrent Programs. In *ESEC/SIGSOFT FSE* (2009), pp. 53–62.

[71] Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., and Zheng, W. Do I Use the Wrong Definition?: DeFuse: Definition-use Invariants for Detecting Concurrency and Sequential Bugs. In *OOPSLA* (2010), pp. 160–174.

[72] Sorin, D. J., Martin, M. M. K., Hill, M. D., and Wood, D. A. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *ISCA* (2002), pp. 123–.

[73] Sorrentino, F., Farzan, A., and Madhusudan, P. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *SIGSOFT FSE* (2010), pp. 37–46.

[74] Taylor, R. N., Levine, D. L., and Kelly, C. D. Structural Testing of Concurrent Programs. *IEEE Trans. Software Eng. 18*, 3 (1992), 206–215.

[75] Veeraraghavan, K., Chen, P. M., Flinn, J., and Narayanasamy, S. Detecting and Surviving Data Races Using Complementary Schedules. In *SOSP* (2011), pp. 369–384.

[76] Volos, H., Goyal, N., and Swift, M. Pathological Interaction of Locks with Transactional Memory. In *TRANSACT* (2008).

[77] Voung, J. W., Jhala, R., and Lerner, S. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/SIGSOFT FSE* (2007), pp. 205–214.

[78] Wang, C., Said, M., and Gupta, A. Coverage Guided Systematic Concurrency Testing. In *ICSE* (2011), pp. 221–230.

[79] Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., and Mahlke, S. A. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI* (2008), pp. 281–294.

[80] Weeratunge, D., Zhang, X., and Jagannathan, S. Analyzing Multicore Dumps to Facilitate Concurrency Bug Reproduction. In *ASPLOS* (2010), pp. 155–166.

[81] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA* (1995), pp. 24–36.

[82] Xu, M., Bodík, R., and Hill, M. D. A Serializability Violation Detector for Shared-memory Server Programs. In *PLDI* (2005), pp. 1–14.

[83] Yang, C.-S. D., Souter, A. L., and Pollock, L. L. All-du-path Coverage for Parallel Programs. In *ISSTA* (1998), pp. 153–162.

[84] Yu, J., and Narayanasamy, S. A Case for an Interleaving Constrained Shared-memory Multi-processor. In *ISCA* (2009), pp. 325–336.

[85] Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., and Reps, T. W. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS* (2011), pp. 251–264.