IMPROVING CONCURRENT SOFTWARE RELIABILITY VIA AN EFFECT-ORIENTED APPROACH

by

Wei Zhang

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 06/14/2013

The dissertation is approved by the following members of the Final Oral Committee:

Shan Lu, Assistant Professor, Computer Sciences

Mark Hill, Professor, Electrical and Computer Engineering

Thomas Reps, Professor, Computer Sciences

Karthikeyan Sankaralingam, Assistant Professor, Computer Sciences

Michael Swift, Assistant Professor, Computer Sciences

To my parents.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Shan, my advisor, for the endless effort that she spent on me. Shan always pushed me to think, which is the most important lesson that I have learnt from the graduate school. I feel extremely honored to have become Shan's first Phd student and to have had the great opportunity to observe and (hopefully) learn how she works and succeeds in every aspect of her life. What I have learnt from Shan far exceeds what I had ever expected from graduate school.

I would like to thank Tom for co-authoring the ConSeq and ComMem (journal version) paper. I have been constantly awed by Tom's serious attitude towards research and his professionalism. Tom's words "There gotta be a principled way of doing this." will always inspire me.

I would like to thank Karu for co-authoring the ConAir paper. In the later days of my graduate school, Karu taught me many things beyond doing research, such like how to find a job and how to make a decision. Karu's words always keep me calm.

I would like to thank Mike for giving me great advice for giving presentation, solving technical issues and passing qualify exam. I am particularly thankful that Mike is always willing to help. I am leaving Wisconsin with an unshakable belief that Mike knows everything.

I would like to thank Mark for giving me great advice for my dissertation and helping make my dissertation stand on a solid foundation.

Overall, I would like to thank Shan, Tom, Karu, Mike and Mark for serving on my thesis committee.

In Wisconsin, I was fortunate enough to have taken classes from these professors: Remzi, David, Susan, Somesh, Jeff, Jerry, and Ben. I also had the privilege of co-authoring the AFix and CFix paper with Ben. I would also like to thank Perry and Angela for making my graduate school life smooth.

This dissertation would not be possible if were not for my student co-authors: Chong, Ramya, Junghee, Joel, Guoliang, Dongdong, Ang and Marc. I would also like to thank every member of Shan's group (Guoliang, Linhai, Dongdong, Po-Chun and Joy) for making our

group an exciting and fun one. It is rather miraculous that my office-mates, Bill and Drew in particular, stay sane despite all the mess caused by me.

Bo is my roommate for 3 years and has been like a brother to me. Shengqi is one of the most intellectually intimidating people that I have met, yet he becomes one of my best friends. Ning, another Beigongda survivor, shared many ups and downs with me throughout our graduate school career. I would also like to thank my other dear graduate school friends – Arka, Asim, Cong, Haris, Junghee, Ramya, Yiqing, Yiying and Yupu – for making the pain less noticeable.

Marrying Yiwen is the luckiest thing that happened to me in graduate school. And this dissertation would mean nothing to me if it did not make my parents proud.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Software reliability is critical, as it directly affects every aspect of people's daily life. Software bugs are the primary factor that impairs the software reliability. Among all types of software bugs, concurrency bugs are the most difficult to detect and the failures caused by them occur randomly. That is because concurrent programs need to handle multiple tasks at the same time, so meticulously designed synchronization is needed. It is difficult to have correct synchronizations as they require non-sequential thinking, which is complicated to reason about. As a result, programs are likely to contain concurrency bugs. Furthermore, concurrency bugs manifest themselves only under rare timing, which makes them even harder to be reproduced and understood. Concurrency bugs have caused real world damage, such as the 2003 Northeast blackout. The concurrency bug problem is only getting worse because we are in a multi-core era – servers, desktop computers, laptops and mobile devices are already.

To improve concurrent software reliability, this dissertation focuses on two aspects of fighting concurrency bugs: (1) Bug detection, i.e., find the bugs before the software is released; and (2) Failure recovery, i.e., help programs recover from failures caused by concurrency bugs during production runs. Both are known to be difficult tasks. To detect them, existing approaches focus on finding the root cause (i.e., buggy interleaving pattern) of the bug. However, there are many different interleaving patterns; bug detectors designed for one type will miss other types. This will lead to false negative problems. In addition, the interleaving pattern in doubt might not be harmful and cause failures, thus it creates the false positive problem. To recover from the failures, existing approaches checkpoint all threads and rollback all of them upon a failure. This is clearly every expensive.

This dissertation takes a dramatically different perspective and makes the following three contributions:

First, it conducts a comprehensive characteristics study to understand the effect of concurrency bugs, i.e., types of errors and failures caused by concurrency bugs and how an error propagates to a failure. This characteristics study nicely complements the existing understanding of the causes of concurrency bugs to give a complete picture of a concurrency

bug's whole life cycle. The findings reveal several previously unknown characteristics of the concurrency bugs' error propagation process. In particular, three findings are made: (1) Errors/Failures caused by concurrency bugs are similar to those by sequential bugs. (2) Error propagation of concurrency bugs is usually within one thread. (3) Error propagation distance is usually short. These three findings are instrumental in this dissertation because they indicate that concurrency bugs' effects are easier to understand and analyze compare to their causes. Thus one could fight concurrency bugs using an *effect-oriented* approach rather than the traditional *cause-oriented* approach.

Second, guided by the *effect-oriented* insight, this dissertation proposes bug detection tools that are able to detect concurrency bugs with different types of root cause (e.g., data races, atomicity violations, order violations and other types). Different from existing approaches that use multi-threaded analysis to identify suspicious root causes; our approach starts from potential failure/error sites in programs and conducts well-developed single-threaded analysis before complicated multi-threaded analysis is applied. Our approach can detect many real-world concurrency bugs and discover concurrency bugs that were never reported before. Our tools achieve 10 times lower false positive rate than state of the art bug detection technologies, while incurring reasonable runtime overhead for in-house testing.

Third, guided by this *effect-oriented* insight, this dissertation proposes a failure recovery tool that is able to survive a wide variety of hidden concurrency bugs. Previous approaches periodically checkpoint whole program states across all threads; on the contrary, our approach only checkpoints program state for each single thread at places close to the potential failure sites. Thus, it incurs negligible overhead and guarantees not to introduce any new bug. Furthermore, it does not require any hardware or OS modifications.

The bug detection tools and failure recovery tool, guided by the *effect-oriented* principle, complement each other to significantly improve concurrent software reliability.

Terminology

In this thesis, we use the following definitions.

1. *Software bug*: A software bug is a flaw in a computer program that prevents the program from producing the correct results.

2. *Failure*: A system failure [65] occurs when the delivered service deviates from fulfilling the system function, the latter being what the system is aimed at.

3. *Error*: An error [65] is that part of the system state which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred.

4. *Fault*: The adjudged or hypothesized cause of an error is a fault [65].

5. *Sequential bug*: A sequential bug is a bug whose fault, error, and failure all involve only one thread in a software.

6. *Concurrency bug*: A concurrency bug is a bug whose fault involves multiple threads.

7. *Memory error*: A memory error is an incorrect program state that leads to invalid memory accesses. Invalid memory access occurs when an instruction references memory that is logically or physically invalid [1].

8. *Memory bug*: A memory bug is a sequential bug that leads to a memory error.

9. *Semantic bug*: Any sequential bug that is not a memory bug is a semantic bug [137].

10. *Semantic error*: Errors caused by semantic bugs are semantic errors.

11. *Critical read*: A critical read is the first shared memory read that returns an incorrect value as a result of a buggy interleaving.

12. *Concurrency error*: Concurrency errors are errors caused by concurrency bugs.

13. *Error propagation process in concurrency bugs*: We define error propagation process in concurrency bugs as the propagation process from the critical read to the failure.

1 INTRODUCTION

1.1 Motivation

Software industry is a large industry, as of the year of 2007, it contributed over 260 billion dollars to the US economy annually [95]. Software bugsmake software misbehave and cause huge financial loss [90] and security breaches [16].

Compared to sequential bugs, concurrency bugs (we have defined concurrency bug in the Terminology section) are much harder to find and avoid. Many concurrency bugs are only manifested under very rare timing, thus they are called Heisenbugs [48]. Due to their non-deterministic nature, concurrency bugs are very hard to reason about. Concurrency bugs have caused severe real world damage. A concurrency bug in Thera-25 claimed several lives in the 1980s [70]. A concurrency bug in the power grid caused the 2003 Northeast blackout [112], which is arguably the worst power outage in American history. Most recently, the Facebook IPO was delayed for half an hour, which caused over 500 million dollars of loss for the trading firms [133]. It was later confirmed that a concurrency bug caused the delay [55].

Problems caused by concurrency bugs are projected to get worse because we are in a multi-core era [3]. Moore's law came to an end for single-core CPUs [37]. Soon any serious software developer has to develop parallel code, explicitly or implicitly, to exploit the parallelism of the hardware [3]. However, concurrent programming is notoriously hard to get right [52, 82]. Thus, it is increasingly important to improve concurrent software reliability.

This dissertation proposes novel bug detection techniques to help developers find concurrency bugs before the software is released. This dissertation also proposes novel failure recovery technique to help end users so that the software can transparently recover from the failures caused by concurrency bugs. The proposed bug detection technique and failure recovery technique complement each other to make concurrent software more reliable.

1.2 Traditional approaches to improving concurrent software reliability

Bug detection and failure recovery are two main approaches to improving concurrent software reliability. Bug detection aims at detecting bugs before the software is released. To locate the bug is the first step to correct the buggy program. Only when a bug is reported to developers can it be fixed and a more robust program can be deployed. Failure recovery focuses on hardening the production run software. It is very common for undetected bugs to slip into production run and cause failures. The goal of failure recovery is to mask the failures during the production run. It serves as the last defense against the bugs.

1.2.1 Concurrency bug detection

Existing concurrency bug detection techniques focus on understanding the root cause (i.e. buggy interleaving[1]) of the concurrency bugs and detecting the abnormal interleaving patterns. Among them, data race detection and deadlock detection are the most studied ones. A data race occurs when different threads simultaneously access the same memory location and at least one of the accesses is a memory write. A deadlock occurs when two or more threads block each other by holding a lock that is acquired by the other thread. In the past three decades, much effort has been devoted to find such bugs. However, they still widely exist [34].

Atomicity violation and order violation concurrency bugs also widely exist. Atomicity violation [40, 75, 120, 73] is caused by the violation of the desired serializability among multiple memory accesses. Figure 1.1 illustrates an atomicity violation bug example. The null assignment to the shared pointer gCurrentScript from thread 2 breaks the atomicity of accessing the same pointer in thread 1, a null pointer dereference occurs and causes the program to crash. Order violation [74] happens when the desired order of two (groups of) memory accesses is flipped. Figure 1.2 illustrates an order violation bug example. The initialization of global pointer mThd in thread 2 comes late and the dereference of this uninitialized pointer in thread 1 causes program to crash.

[1]Interleaving refers to the thread execution sequence in this dissertation.

```
1  //Thread 1
2  void LoadScript(nsSpt *aspt){
3    Lock(l);
4    gCurrentScript = aspt;
5    LaunchLoad(aspt);
6    UnLock(L);
7  }
8
9  void OnLoadComplete(){
10   /*call back function of LaunchLoad*/
11   Lock(l);
12   gCurrentScript->compile();
13   UnLock(l);
14 }
```

```
1  //Thread 2
2
3
4
5
6
7  Lock(l);
8  gCurrentScript = NULL;
9  UnLock(l);
```

Figure 1.1: A real bug in the Mozilla application suite, slightly simplified for illustration. When thread 2 violates the atomicity of thread 1's accesses to gCurrentScript, the program crashes.

```
1  //Thread 1
2  Get(){
3    ...
4    tmp=GetState(mThd);
5  }
6
7  GetState(THD *thd)
8  {
9    return(thd->state &
10       THREAD_DETACHED);
11 }
```

```
1  //Thread 2
2  //mThd is shared
3  //between two threads;
4  //it is 0 before
5  //initialized below.
6
7  InitThd(){
8
9    mThd =
10     CreateThd(..);
11 }
```

Figure 1.2: A real bug in the Mozilla XPCOM module. Program crashes when the order between thread 2 initializing the global pointer mThd and thread 1 dereferencing the same pointer is not maintained.

Many concurrency bug detection research projects yield exciting results [92, 111, 32, 40, 75, 73]. However, two problems remain. The first one is the high false positive rate. According to a recently study, only 2% – 10% reported data races are true bugs and cause the program to fail [89], which means the rest are benign bugs that deserve no special attention. Under the pressure of releasing software on time and facing a long list of bug reports, developers are reluctant to study each of them and fix the few true bugs. The other problem is high false negative rate. Concurrency bugs are caused by many different types of interleaving patterns. Bug detectors designed for one specific pattern will inevitably miss bugs caused by other patterns. For example, many atomicity violation bugs, such as the example shown in Figure 1.1, are not data races; many order violation bugs, such as the example shown in Figure 1.2, are not atomicity violations; and so on. In addition, there are still many real-world

concurrency bugs caused by interleaving patterns that are not targeted by any bug detectors yet (Chapter 4 will provide such examples).

1.2.2 Failure avoidance/recovery

Despite much effort recently spent on concurrency bug detection, concurrency bugs do slip into the production run. It is mainly because that program state of a multi-threaded program is extremely large and it is prohibitively expensive to check all these states and find all bugs. As a result, many concurrency bugs go undetected and may cause failures. Thus, it is important to either dynamically avoid the failures or automatically allow the software to recover from the failures.

Failure avoidance Traditional approaches focus on proactively restraining thread scheduling to disable the interleavings that *might* lead to failures. Among them, atomicity violation failure avoidance is the most studied [79, 78]. Such an approach instruments the program and restrains accesses to the same memory locations from different threads, thus enforcing atomicity. Similar to the traditional concurrency bug detection approaches, this kind of scheme suffers from the false positive problem because the interleavings it deems questionable may not be harmful. It suffers a runtime performance penalty because it requires orchestrating different threads to follow some specific scheduling order. Avoiding all such benign interleavings incurs a large program runtime overhead, while does not improve software reliability at all. To reduce the performance penalty, one has to modify the OS or the hardware. Such a scheme also suffers from the false negative problems in that it is very difficult to design a scheme that can avoid all harmful interleaving patterns.

Failure recovery Another genre of tackling production run failure problem is via failure recovery approaches. This kind of approach periodically checkpoints the whole program memory state [105, 87, 117, 122]. Upon a failure, the program is rolled back to the most recent checkpoint. One benefit of this approach is that it can allow programs to recover from many types of concurrency bugs, as long as it can identify the failure symptoms. However, the cost of rollback and re-execution is usually high, as it needs to checkpoint the whole memory

state, which is both time-consuming and space-consuming; and rollback all the threads, which effectively stalls the whole program. To reduce the cost, one also needs to modify the OS [105] or the hardware [87], which makes it impractical for the software that run on commodity systems.

1.3 Contributions

1.3.1 Principles behind this dissertation

This dissertation studies both concurrency bug detection and concurrency failure recovery problems. It tackles these two problems by using a simple principle – **effect-oriented** analysis.

The lifecycle of a software bug consists of three parts [65]: *fault*, *error* and *failure*. We have defined fault, error and failure in the Terminology section. In this dissertation, the terms fault and bug are used interchangeably. In the example shown in Figure 1.1, fault is the atomicity violation; when the null pointer is loaded, the fault is triggered and the program is in an error state; in the end the program crashes (i.e., fails). Fault is the cause of a software bug, error and failure are the effect of a software bug, a complete software bug life cycle is depicted in Figure 1.3.

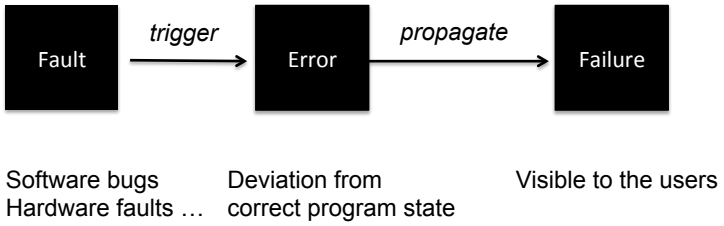


Figure 1.3: Bug life cycle: fault-error-failure

Existing research focuses on the root cause (fault) of concurrency bugs. It is shown that there are many different types of causes for concurrency bugs, such as data races, atomicity violations, order violations, and deadlocks. They are very different from the causes of sequential bugs. Additionally, they are very difficult to understand. On the contrary, little research has been done to understand the effect of a concurrency bug's life cycle. Three key questions

regarding the concurrency bug effect remained unanswered: (1) what kinds of errors are there, (2) how does the failure manifest itself, and (3) how does an error propagate to a failure. This dissertation first answers these three questions, then makes the observation that the effect of concurrency bugs are much easier to understand, analyze, thus can be leveraged to solve bug detection and failure recovery problems.

Overall, this dissertation makes three contributions:

1. A characteristics study that aims to understand the effect of concurrency bugs and answer the above questions. The observations of this study will guide the other parts of this dissertation.

2. Two concurrency-bug detection tools, ConMem and ConSeq, that conduct effect-oriented bug detection, guided by the above characteristics study. These two tools complement each other by handling two dominant types of errors caused by concurrency bugs. By using the effect-oriented principle, ConMem and ConSeq are able to accurately detect concurrency bugs of different root causes, regardless of data races, atomicity violations, order violations, or other odd interleaving patterns. They can detect many real-world concurrency bugs and discover concurrency bugs that were never reported before. And they achieve 10 times lower false positive rate than the state of the art bug detection technologies.

3. A failure recovery technique, named ConAir, that is able to help software recover from a wide variety of unknown hidden concurrency bugs. ConAir also leverages the common patterns of real-world concurrency bugs' effect. It incurs negligible overhead, without any hardware or OS modification.

The proposed concurrency bug detection technique and concurrency failure recovery technique share the same design principle, which is supported by the characteristics study; and complement each other to improve concurrent software reliability. The content of our characteristics study, concurrency bug detection technique and concurrency failure recovery technique are previewed in the following chapters.

1.3.2 The characteristics of real-world concurrency bugs' effects

As we will elaborate in Chapter 3, our characteristics study reveals that most concurrency bugs have the following error propagation process (as shown in Figure 1.4): First, the error is triggered by reading a bad shared variable value, which we will refer to as "critical read". The formal definition of critical read was given in the Terminology section. As an example, a null pointer read in the examples shown in Figure 1.1 and Figure 1.2 is a critical read. Next, error propagates for a short distance within one thread until the program fails. As discussed in the Terminology section, we define error propagation process in concurrency bugs as the propagation process from the critical read to the failure. In the end, program fails in the same way as sequential program such as crashes due to memory errors, or assertion failures due to semantic errors. We gave the definitions of memory error and semantic error in the Terminology section.

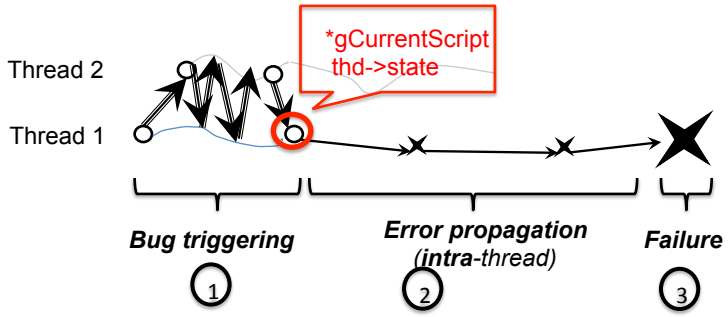


Figure 1.4: Concurrency bug life cycle.

Three observations are made out of this error propagation process. They drive the design of concurrency-bug detection technique and concurrency-bug failure recovery technique presented in this dissertation:

Observation 1: The patterns of concurrency errors resemble those of sequential bugs' errors, including both memory errors and semantic errors. Furthermore, we have observed almost all common patterns of memory errors, including null-pointer dereferences, dangling-pointer dereferences, buffer overflows and uninitialized reads. This indicates that it is not difficult to identify potential failure/error sites. For example, one can treat an assert call as the potential

failure site for an assertion failure.

Observation 2: Error propagates within one thread, starting from the critical read and ending at the failure site. This indicates that single-threaded analysis can be applied to concurrency bug detection. In addition, to recover from concurrency-bug failures the program often only needs to rollback one thread.

Observation 3: Error propagation distance is short. This indicates that even sophisticated single-threaded analysis can be applied without scalability concerns. In addition, recovering from concurrency-bug failures may not require a long rollback.

In a word, the observations indicate that we can start from the potential failure or error sites of a program and leverage mature single-threaded analysis to the full-extent to tackle the concurrency bug detection and failure recovery problem.

1.3.3 Effect-Oriented Concurrency-Bug Detection

The concurrency bug detection technique presented in this thesis consists of two pieces of work: ConMem, which handles memory errors caused by concurrency bugs; and ConSeq, which mainly handles semantic errors caused by concurrency bugs. ConSeq can also detect some concurrency bugs that cause memory errors. For example, an uninitialized read, which is a type of memory error, can lead to an assertion failure. Since ConSeq works from the potential failure site (as discussed later in this chapter), it is able to identify a buggy interleaving that can cause the assertion failure thus finds the same bug as ConMem does. Both ConMem and ConSeq take an effect-oriented approach: start from potential failure/error sites (they are structurally identifiable thanks to Obs1) of a program and detect concurrency bugs in a backward fashion. Both of them leverage the findings that well-known single-threaded analysis can be applied to facilitate the concurrency bug detection (thanks to the Obs2) and such analysis is inexpensive (thanks to the Obs3).

ConMem ConMem is a toolset that starts from potential memory error sites (e.g., pointer dereference) and conducts dynamic analysis to identify the buggy interleaving that leads to the memory errors, such like null pointer dereferences, uninitialized reads, dangling pointers

and buffer overflows. Such memory bugs usually cause severe failures (e.g., program crashes) and they contribute to a significant portion of all concurrency bugs. There are 2 steps in ConMem:

Step1 Structurally identify the potential critical reads (i.e., the superset of critical reads) for memory errors in the program, including shared pointer dereferences, shared memory reads, and shared buffer indexing.

Step2 After the potential critical reads are identified, ConMem explores which interleaving causes the memory errors at the critical reads. For example, to find a null-pointer deference error caused by a concurrency bug, ConMem does the following: it tries to find a buggy interleaving that feeds the NULL value to a shared pointer. Once a buggy interleaving is identified, ConMem has an additional component to insert artificial delays at the corresponding places to automatically trigger the bug.

ConSeq ConSeq is a toolset that starts from potential failure sites (e.g., assertion failures), and conducts intra-thread analysis to locate potential critical reads, and then identifies the buggy interleaving that leads to such failures. Such bugs have explicit failure sites (e.g., assertion failure, error message display, etc) that can be structurally identified.

ConSeq conducts three steps to find such a bug.

Step 1 ConSeq locates potential failure sites.

Step 2 Starting from the failure sites, ConSeq analyzes the program backwards to locate the potential critical reads.

Step 3 After the potential critical reads are identified, ConSeq searches for the buggy interleaving that makes the critical reads get erroneous values and causes program to fail. ConSeq conducts this step similar to ConMem.

Both ConMem and ConSeq achieve great results: for the set of real-world non-deadlock concurrency bug benchmarks that we evaluated (10 for ConMem and 12 for ConSeq), the result shows that both ConMem and ConSeq can find those bugs in the benchmark and

moreover they can find bugs that were never reported. Also, they achieve 10 times lower false positive rate than that of the state of the art race detector and atomicity violation detector, while detecting many bugs that are missed by those tools.

Such a good result benefits from the effect-oriented design philosophy. ConMem/ConSeq report the bugs that cause failures, thus they have much a smaller false positive problem. ConMem/ConSeq report bugs regardless of their root cause, thus they have much a smaller false negative problem compared to the existing bug detector that focuses on one specific type of root cause.

Both tools incur at most 40 times runtime overhead, which is sufficiently low for the in-house testing.

1.3.4 Featherweight concurrency-bug failure recovery

ConAir is a tool that transparently hardens software against failures caused by concurrency bugs. It also leverages our three observations: (1) ConAir can identify potential failure sites (such as the assert calls) caused by concurrency bugs in the same way as identify potential failures caused by sequential bugs (implied by Obs1); (2) rolling back a single thread is sufficient to recover from most concurrency-bug failures (implied by Obs2); (3) re-executing a short code region (implied by Obs3) is sufficient to recover from many concurrency-bug failures. ConAir workflow includes 3 steps:

Step 1 A static analysis component that automatically identifies potential failure sites.

Step 2 A static analysis component that automatically identifies the idempotent code regions[2] around every failure site.

Step 3 A code-transformation component that inserts rollback-recovery code around the identified idempotent regions.

ConAir also achieves great results. To evaluate Conair, we used 10 real-world concurrency bugs, which represent different root causes and failure symptoms. The evaluation result shows

[2]More details about idempotent regions are presented in Chapter 5

ConAir helps software survive all the failures caused by these bugs, while incurring negligible run-time overhead (less than 1%) and short recovery time. ConAir can also help programs recover from failures caused by unknown bugs. It guarantees that program semantics remain unchanged and requires no change to operating systems or hardware.

Such a good result also benefits from the effect-oriented design philosophy and our key observations: (1) ConAir can help harden a wide variety of potential failure sites by identifying them as in the sequential programs. (2) ConAir incurs very little overhead because it leverages the observations that re-executing a small code-region within the failure thread is sufficient to survive many failures.

1.4 Dissertation Organization

Chapter 2 provides background and related work integral to this dissertation. Chapter 3 presents the characteristic study of concurrency bugs' error propagation process. The findings of this study are key to the design principle of this dissertation's proposed techniques. Chapter 4 presents the design, implementation, and evaluation of ConMem and ConSeq. Chapter 5 presents the design, implementation, and evaluation of ConAir. Chapter 6 concludes.

2 BACKGROUND AND RELATED WORK

This chapter presents the related work in the concurrent software reliability research literature. Chapter 2.1 presents the related work of empirical studies of concurrency bugs. These works are closely related to this dissertation's Chapter 3. Chapter 2.2 presents related work of concurrency bug detection. These works are closely related to this dissertation's Chapter 4. Chapter 2.3 presents related work of concurrency failure recovery and avoidance. These works are closely related to this dissertation's Chapter 5.

2.1 Empirical studies of concurrency bugs

Due to the lack of concurrency bug sources, only a few studies [38, 74] have been done, and they mostly focus on the interleaving patterns. Most recently, interesting studies have been conducted to evaluate how new synchronization primitives (such as Transactional Memory) can be used in concurrent programs [109]. This dissertation complements previous studies by looking at the error-propagation process of concurrency bugs.

2.2 Concurrency bug detection

2.2.1 Cause-oriented concurrency bug detection

Many techniques have been proposed to detect data races [111, 34, 136, 41], atomicity violations [75, 19, 40, 73, 78, 60], and order violations [44, 76, 134, 116]. In general, ConMem/ConSeq complement these tools by starting from effects rather than causes. ConMem/ConSeq have a smaller false-positive rate than most traditional tools, and can effectively detect many concurrency bugs caused by complicated interleavings that are difficult for traditional tools to detect.

Recently, several invariant-based tools have been proposed for debugging concurrency bugs with complicated causes, such as definition-use violations and order violations [116, 100]. These tools rely on observing many training runs to identify abnormal interleavings. As a

result, insufficient training will cause false positives for them. Furthermore, they cannot report bugs until the buggy interleavings are observed. On the contrary, ConMem/ConSeq requires no training.

2.2.2 Interleaving testing

ConMem/ConSeq share the same goals of interleaving-testing tools [31, 85, 113, 99]. Such tools all try to explore the *interleaving* space for each *input* provided by developers during in-house testing. ConMem and ConSeq use synchronization analysis and perturbation-based interleaving enforcement in common with some of these tools [99]. The difference between this dissertation and previous tools is that this dissertation uses different methods to guide its exploration of the interleaving space.

RaceFuzzer [113] and CTrigger [99] use race/atomicity-violation detection results to guide their interleaving testing. ConMem and ConSeq can complement them by exposing concurrency bugs that cannot be detected by race/atomicity detectors. In terms of performance, the high false-positive rates of these bug-detection tools determine that RaceFuzzer and CTrigger will need to test many more interleavings than ConMem and ConSeq for *each* input.

CHESS [85] guides its interleaving testing by bounding the number of preempting context switches. Although this is an effective heuristic, CHESS still faces the challenge of balancing coverage and performance, because its testing space increases exponentially in the number of potential context-switch points during the whole execution. If it considers each shared memory access to be a potential context-switch point, CHESS still cannot effectively explore the large interleaving space of applications like MySQL, as shown in previous work [125]. Therefore, CHESS often limits context switches only to synchronization points. This constraint will fail to expose many concurrency bugs, including many bugs discussed in this paper. Recently, people also proposed using randomized schedulers [11] to probabilistically expose concurrency bugs. It works very well for simple concurrency bugs, such as many order-violation bugs, but will be ineffective for many other bugs in large applications, such as atomicity-violations or more complicated bugs.

In general, ConMem/ConSeq and previous interleaving-testing tools complement each other by looking at the interleaving space from different perspectives.

2.2.3 Concurrent program analysis and model checking

Much inspiring research has been conducted on static analysis and model checking of concurrent programs. A recent study [20] inventively proposes leveraging race detection to improve data-flow analysis in concurrent programs. The idea is promising; however, due to pointer-aliasing and other issues, there are still as many as 40% of all pointer dereferences in the program that cannot be proved to be safe in their experiments. ConMem/ConSeq have a completely different design goal from static-analysis tools. ConMem/ConSeq do not aim to provide any guarantee. However, as dynamic bug-detection tools, ConMem/Conseq naturally have the advantage of no pointer-aliasing problem and can achieve better accuracy and scalability.

Model checking can also be used to validate certain properties in concurrent programs. Significant progress has been made [45, 103, 43, 85] in model checking large concurrent programs. However, the state-space-explosion problem still exists. We expect the effect-oriented approach and the error-propagation characteristics studied in this dissertation will help provide heuristics that can be used in future model checkers.

2.2.4 Input generation and symbolic execution

ConMem/ConSeq and other testing tools [31, 85, 113, 99] all rely on input test-case generation to provide good test suites. Recently, symbolic execution [14, 138, 46, 115] has been used to generate high-coverage inputs for unit testing. DDT [63] and ESD [138] further extended this approach for concurrent programs. Using symbolic execution, DDT can detect synchronization problems caused by untimely interrupts in device drivers, and ESD can generate bug-triggering inputs and interleavings based on core-dumps from deadlock situations. These inspiring works can potentially help all testing and diagnosis tools, including ConMem/ConSeq. However, due to the scalability constraints of symbolic execution and theorem provers, previous work only experimented with relatively small applications or relatively simple interleaving problems.

2.3 Failure recovery and failure avoidance

2.3.1 Failure recovery

Software checkpoint and replay

Checkpoint and replay are useful techniques for failure recovery. Many techniques have been proposed to checkpoint and replay multi-threaded software deterministically or non-deterministically [2, 62, 54, 67]. To achieve good performance, these techniques often require sophisticated operating-system support or hardware support. ConAir only rolls back an idempotent region in one thread and does not require these sophisticated techniques.

Rollback recovery

Several rollback-recovery systems have been built before, such as Rx [105], ASSURE [117], and Frost [122]. They all change operating systems to support whole program checkpoint and rollback. Rx changes the program environment during reexecution to handle deterministic bugs. ASSURE rolls back a failed software to an existing error-handling path. It is designed to mitigate the impact of deterministic bugs, and cannot help software generate correct results after the manifestation of a non-deterministic concurrency bug. Frost [122] proposes a novel solution to survive data races. With OS support, it executes multiple replicas of the program with complementary thread schedules at the same time. Periodically, it compares the states of different replicas and tries to survive state divergence caused by data races. In general, these systems all require checkpointing the whole program states and rolling back all threads during a failure. Consequently, they all require sophisticated changes to operating systems.

Microreboot [15] is a recovery technique that reboots only application components, instead of the whole program, when failures occur. To benefit from microreboot, the programmers have to manually separate their systems into components (groups of objects) that can be individually restarted, such as Enterprise Java Beans components in J2EE applications. ConAir shares a common high-level philosophy with microreboot of not rolling back the whole program. However, the similarity ends there. ConAir focuses on concurrency-bug

failure recovery. It works on any C/C++ multi-threaded software without manual changes. It automatically identifies reexecution points and conducts automated code transformation.

Apart from rollback recovery, a recent work studies the phenomenon that some software is able to automatically recover from state corruption, because they overwrite the corrupted states with new input data. This type of software is called self-stabilizing programs [33]. To some extent, ConAir can transform a multi-threaded program to become self-stabilizing.

Idempotency

While the idea of leveraging idempotency for recovery is not new [25, 27, 39, 81, 50, 24, 61, 26, 107], ConAir is the first to apply it towards the problem of recovery from concurrency bugs. Additionally, most previous work on idempotency has assumed hardware support for recovery with a focus on hardware exceptions [25, 81, 50], hardware faults [39, 24], and hardware mis-speculation [61]. ConAir requires no hardware support. While the general paradigm of idempotent processing [27], which allows programs to be executed entirely as sequences of idempotent regions, does not strictly require hardware support to enable various features, such technique does *not* work for general multi-threaded programs. This technique allows an idempotent region to store to shared variables. Such a region cannot be considered idempotent in the presence of data races and hence their algorithm cannot be used. Instead of splitting the entire program into idempotent regions, ConAir only identifies idempotent regions that end at potential concurrency-bug failure sites. This focused approach allows ConAir to achieve negligible overhead (<1%), while previous work could incur more than 10% overhead [27].

2.3.2 Failure avoidance

Run-time failure avoidance has also been studied for concurrency bugs. Dimmunix [59] learns lessons from previous deadlocks to avoid future deadlocks. Compared to ConAir, Dimmunix focuses on a specific root cause (i.e. deadlocks), where ConAir works for bugs of many different root causes. Atom-Aid [79] and PSet [134] provide ways to survive concurrency bugs by prohibiting certain patterns of interleavings at run-time through hardware support, whereas

ConAir works directly on commodity hardware. Aviso [77] is a cooperative approach to avoiding failures caused by concurrency bugs. Aviso requires a much more controlled environment (e.g., many copies of deployed software, centralized failure information collection/distribution server, etc) and incurs a relatively large runtime overhead (up to 30%).

Deterministic execution [28, 72, 4, 7, 96, 22] pushes failure recovery to an extreme. Software-only tools like Grace [8] and Kendo [97] achieve similar goals for certain types of multi-threaded programs at run-time. This promising approach still faces challenges, such as overhead, integration with system non-determinism, language design, etc. In general, these tools address different problems from ConAir. Even inside a deterministic run time, concurrency bugs can still occur and require recovery.

3 CHARACTERISTICS STUDY OF CONCURRENCY BUG LIFE CYCLE

3.1 Overview

3.1.1 Motivation

Previous work focuses on understanding the interleaving patterns of concurrency bugs (aka the root cause of concurrency bugs). It turns out the interleaving patterns are very hard to reason about. On the contrary, very little work has been done to study the effect of concurrency bugs.

This dissertation is the first work, to our best knowledge, that studies the error-propagation process of concurrency bugs' life cycle. The findings of this study can help both concurrency bug detection and failure recovery.

3.1.2 Methodology

We manually studied a set of 70 non-deadlock concurrency bugs, collected in a previous study [74]. All of these 70 bugs are reported by users and fixed by developers from four widely used C/C++ open-source applications: Apache HTTP server, MySQL database server, Mozilla web browser, and OpenOffice office tool-kits. These bugs are collected by previous researchers through random sampling among all fixed bugs in the bug databases. We choose to focus on non-deadlock concurrency bugs, because deadlocks have much more regular effects and are better understood and addressed than non-deadlock bugs. Specifically, we have 40 bugs from Mozilla, 15 from MySQL, 10 from Apache, and 5 from OpenOffice.

3.2 Background and terminology

Concurrency bug life-cycle A concurrent program's execution is a mix of inter-thread communication and intra-thread calculation. Consequently, in rough terms the life-cycle of concurrency bugs consists of three phases: triggering phase, propagation phase and the failure phase. We can better understand these three phases by looking at a real-world bug example

in Mozilla. This bug is depicted in Figure 3.1. In this example, there are two shared variables – InProgress and URL . InProgress is a flag that is true when the web browser has something to work on. URL is a pointer, which points to the website that the browser is working on. Thus when InProgress is true, URL should be a non-null pointer. When InProgress is false, URL should be a null pointer. These two variables are correlated and need to be accessed in the same atomic region. However this atomicity is not enforced. In a buggy interleaving, thread 2 can see InProgress be true but URL be a null pointer, thus this assertion fails.

```
//Thread 1                          //Thread 2
S1: runningURL = NULL;
                                    S3: if(InProgress)
                                          isBusy = TRUE;
S2: InProgress = FALSE;

                                        if(isBusy)
                                    S4:   NS_ASSERT(runningURL);
```

Figure 3.1: A concurrency bug in Mozilla

In this example, the corresponding three phases are as shown in Figure 3.2.

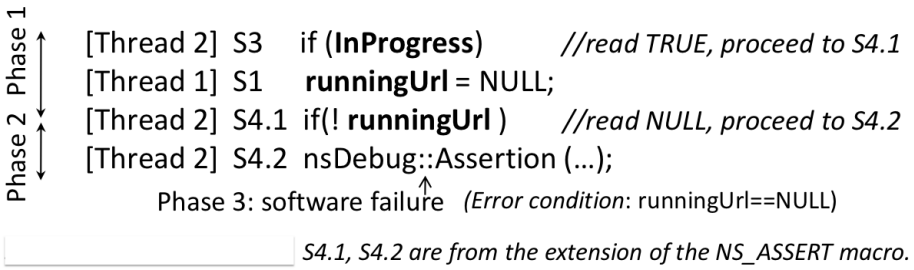


Figure 3.2: Error propagation in a concurrency bug.

Phase 1 is the triggering phase (aka the root cause phase). A concurrency bug is triggered by a specific execution order among a set of shared memory accesses (referred to as a buggy interleaving).

Phase 2 is the propagation phase. It starts with **critical reads** that fetch problematic values from shared memory locations as a result of the buggy interleaving. The effect of these values begins to propagate, usually within one thread, through data and control dependences. Note that the corresponding thread may read many shared variables during the error-propagation phase, but only those that impact the failure through control/data

dependences are considered to be 'critical'. In this dissertation, we measure the **distance** of error propagation through data- or control-dependence edges. During this phase, some variable values begin to deviate from the correct program state. These are called *errors*. Note that races and atomicity violations do not belong to our definition of an error.

Phase 3 is the failure phase. Propagation leads to more severe errors, and finally to an error with an externally visible symptom (i.e., a **failure**) such as a crash or a hung process. To simplify the discussion, the rest of the dissertation will refer to instructions where failures/errors occur as **failure/error sites**. We refer to the same set of instructions as **potential error/failure sites** during correct runs, because they indicate sites of errors that might occur in a different interleaving.

Phase 1 is the cause part of a concurrency bug life's cycle. It always involves multiple threads. Phase 2 and Phase 3 are the effect part of a concurrency bug's life cycle. Usually only the failure thread is involved in phase 2 and phase 3. Critical reads separate the life-cycle of concurrency bugs to two stages: concurrent stage (phase 1) and sequential stage (phase 2 and phase 3). Figure 3.3 depicts these three phases in a concurrency bug's life cycle.

Understanding the life-cycle of a concurrency bug is important for both bug detection and failure recovery. Because the goal of concurrency bug detection is to find an interleaving that can lead the program to an error state and cause program to fail; the goal of concurrency failure recovery it is to rollback and re-execute a program in a correct interleaving that will not lead to an error/failure.

3.3 Characteristics study and the findings

In this chapter, we will try to answer several research questions regarding three aspects of concurrency bugs: errors caused by concurrency bugs, failures caused by concurrency bugs, and the error propagation of concurrency bugs.

3.3.1 Errors caused by concurrency bugs

Question 1: What types of errors are caused by concurrency bugs?

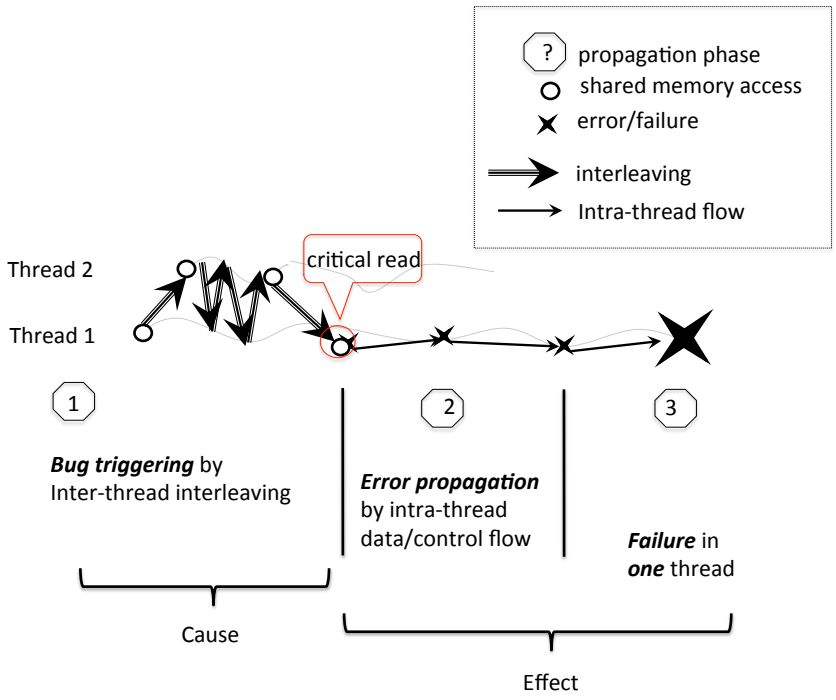


Figure 3.3: The common three-phase error-propagation process for most concurrency bugs.

Error is the starting point of the effect part of a concurrency bug's life-cycle. Is error caused by concurrency bug similar to those caused by sequential bugs? If so, we then could easily recognize them by leveraging techniques that are applicable to the sequential bugs. Little previous research literature surveyed this question.

Our characteristics study finds that errors caused by concurrency bugs can be categorized as two classes: memory errors and semantic errors. Table 3.1 shows the breakdown of each type of errors. All concurrency bugs in the benchmarks cause errors of these two types:

Concurrency memory bugs are concurrency bugs that lead to memory errors (see the Terminology section for the definition of memory errors). Concurrency memory errors occur when the buggy intereleaving changes the execution order of a set of shared memory operations and these operations directly instantiate a memory bug. Figure 3.4 illustrates such an example: a buggy interleaving can cause thread 1 to dereference a shared pointer, which was already nullified by another thread. Further study shows that there are four types of such concurrency memory bugs: NULL pointer dereferences, uninitialized reads, dangling pointers and buffer

overflows. Table 3.2 shows the number of each concurrency memory error type in our benchmarks.

```
//Thread 1                                  //Thread 2
S1:if(thd->proc_info){

                                            S3: thd->proc_info = NULL;
S2:  fputs(thd->proc_info, ...);

   }
```

Figure 3.4: A concurrency memory bug example

	concurrency memory errors	concurrency semantic errors
Mozilla	21	19
MySQL	5	10
Apache	6	4
OpenOffice	2	3
ALL	34	36

Table 3.1: Breakdown of concurrency memory bugs and concurrency semantics bugs

	NULL	UnInit	Dangling	Overflow
Mozilla	9	0	8	4
MySQL	3	1	1	0
Apache	2	0	3	1
OpenOffi	1	1	0	0
ALL	15	2	12	5

Table 3.2: Breakdown of concurrency memory errors

Concurrency semantics bugs cause semantic errors (see the Terminology section for the definition of semantic errors). Concurrency semantic errors occur when the buggy intereleaving causes new and unexpected program states that are not handled by the program. The failures caused by semantic bugs usually are assertion failures, error messages, wrong outputs and hangs. Figure 3.1 shows such a bug example. As we can see, the buggy interleaving does not immediately lead to a failure; rather it propagates to the failure within a few steps (i.e., in thread 2, check InProgress->set isBusy->check isBusy->assert on runningURL).

3.3.2 Failures caused by concurrency bugs

Question 2: What types of failures are caused by concurrency bugs?

Failure is the ending point of the effect of a concurrency bug's life-cycle. Similar to the motivation of studying errors caused by concurrency bugs, we want to investigate what kind of failures are caused by concurrency bugs.

Based on bugs' failure symptoms, we categorize the failure types as crash, hang, and wrong functionalities. Table 3.3 gives the number of each failure type in our benchmarks.

	Crash	Hang	Wrong Func.	Total
Mozilla	24	4	12	40
MySQL	5	0	10	15
Apache	7	2	1	10
OpenOffice	1	1	3	5
ALL	37	7	26	70

Table 3.3: Failure types of concurrency bugs

Further study leads to two additional findings: (1) Crashes are mostly related to the memory errors, in fact 31 out of 37 crash bugs cause memory errors. The rest crashes are usually caused by assertion violation. (2) The wrong function bugs include three common types: wrong outputs, error messages and infinite loops. As we can see, all these failure types are similar to those caused by sequential bugs.

Finding 1: The patterns of concurrency errors resemble those of sequential bugs' errors, including both memory errors and semantic errors. Furthermore, we have observed almost all common patterns of memory errors, including null-pointer dereferences, dangling-pointer dereferences, buffer overflows and uninitialized reads. This indicates for concurrency bug detection and failure recovery, we can identify the (potential) error or failure sites just as how we identify them in a sequential program.

3.3.3 Error propagation process

Question 3: Do concurrency-bug errors propagate and fail in one thread?

After answering the questions regarding the starting and ending points of effect part of concurrency bug life cycle, we need to further study the error propagation process itself to examine how complicated it is. We raise Question 3 because we want to understand whether

the error propagation process involves multiple threads and is hard to reason about, just like the root cause of concurrency bugs.

In our study, we find in 66 out of the 70 examined bugs, error propagates and triggers failures in one thread. The only exceptions are four MySQL bugs that cause inconsistent behavior between two threads: the thread that completes a database operation first finishes its logging second. The behavior of each thread is correct. Only when comparing them with each other would we notice the misbehavior – the global log puts them in a different order from what really happened. For most bugs, propagation starts with multiple threads and ends with one thread, as shown in Figure 3.3.

Question 4: Can software recover from a concurrency bug by rolling back the failure thread?

After studying Question 3, we know that most concurrency bugs propagate and fail in one thread. Therefore, we ask whether rolling back this single failure thread can help software recover from failures caused by these bugs. We study atomicity violations, order violations, and deadlock bugs one by one to answer this question.

Recovering from atomicity-violation bugs Atomicity violations contribute to about 70% of real-world non-deadlock bugs [35]. They occur when two code regions R1 and R2 from two threads interleave unserializably, which violates the expected atomicity of one or both regions. Clearly, if we can rollback and reexecute any one involved thread, the execution of R1 and R2 will be serialized and the failure will be survived. Therefore, to understand whether single-threaded recovery works, we need to know whether the failing thread is involved in the unserializable interleaving. We checked all 50 real-world atomicity-violation bugs in our benchmark set. About 92% of them cause failures in a thread that is involved in the unserializable interleaving. Re-executing one of the two threads will effectively make it happen after the other thread, thus code region R1 and R2 are executed atomically with respect to each other. This implies that failures caused by atomicity violation can potentially be survived by single-threaded recovery. The above observation can be better understood through bug examples shown in Figure 3.5. This figure depicts the most common types of real-world atomicity violations [75]. As we can see, an atomicity violation usually causes an involved

```
/*Thread 1*//*Thread 2*/
log=CLOSE;  if(log!=OPEN)
log=OPEN;     {//output failure}
```

(a) Violating WAW atomicity (Rollback Thread 2 to recover)

```
/*Thread 1*//*Thread 2*/
ptr=aptr;
tmp=*ptr;   ptr=NULL;
```

(b) Violating RAW atomicity (Rollback Thread 1 to recover)

```
/*Thread 1*/ /*Thread 2*/
if(ptr)
  fputs(ptr);ptr=NULL;
```

(c) Violating RAR atomicity (Rollback Thd 1 to recover)

```
/*Thread 1*/            /*Thread 2*/
cnt+=deposit1;
printf("Balance=%d",cnt);cnt+=deposit2;
```

(d) Violating WAR atomicity (Rollback Thread 1 to recover)

Figure 3.5: Most failures caused by atomicity violations can be recovered by rolling back one thread, the failing thread (Different checkpoint/sandbox techniques may be needed to guarantee correctness.).

thread to read an unexpected value from a shared variable, such as log in Figure 3.5a, ptr in Figure 3.5b and Figure 3.5c, and cnt in Figure 3.5d. This incorrect value quickly leads to a failure in that thread. Clearly, rolling back and reexecuting that thread can recover from the failure.

Recovering from order-violation bugs Order violations contribute to nearly 30% of real-world non-deadlock concurrency bugs [35]. They occur when an operation A is expected to execute before an operation B, but instead executes after B due to lack of synchronization. Clearly, if we can rollback and reexecute the thread of B, the occurrence of B will be effectively delayed and the failure will be recovered. Since single-threaded recovery always rolls back the failing thread, we need to understand whether the thread of B is the failing thread. We checked all 20 real-world order-violation bugs in our benchmark set. We found that about 50% of order-violation bugs lead to failures in the thread of B, and hence can be recovered by the single-threaded recovery. Failures of the other bugs manifest in the thread of A and occasionally some other threads. To better understand this observation, one can consider a common type of order-violation bugs: thread tB reads a shared-variable V before V is initialized by thread tA. In this case, the uninitialized value in V usually leads to a failure in tB. By rolling back tB, we can postpone the read of V until V is initialized.

Recovering from deadlock bugs Deadlock contributes to about 40% of all concurrency bugs [74]. Our bug benchmark set does not contain deadlocks. In fact, every deadlock bug fails in

more than one thread. However, recovering from deadlock bugs only requires rolling back one thread. That is because when a deadlock occurs, making any thread release a resource will break the circular dependence and remove the deadlock.

In summary, single-thread rollback-recovery can help recover from all deadlocks, almost all atomicity violations (46 among the 50 studied), and half of the order violations (10 among the 20 studied).

Finding 2: Error usually propagates within a thread. Rolling back one thread can help recover from many concurrency bug failures. This finding implies that we can apply many mature single-threaded techniques to concurrency bug detection and failure recovery problem.

Question 5: How long is the error propagation process?

From finding 1 and finding 2, we know that we can start from the potential error or failure sites and apply single-threaded bug detection or failure recovery technique. To further understand how costly these techniques may be and how much overhead they may incur, we further study the length of the error propagation process. The longer the error propagation process is, the higher the cost is; and vice versa.

For 59 out of the 70 bugs, after the critical read, failure occurs before the current function exits. This trend is consistent across different applications and different types of failures. Of course, the distance is also frequently more than one data/control dependence step. The rationale behind this observation is that there are usually operations within a few dependence steps that have the potential to cause an internal error to be visible externally. These include pointer operations, I/Os, and sanity checks inserted by programmers. This observation about concurrency bugs is consistent with previous observations about other types of software/system defects [49, 71] that have guided previous work on failure recovery [105] and software replay [101].

Finding 3: Error propagation distance is short. This finding implies the single-threaded analysis is not going to be too expensive for bug detection and re-executing a short code region may survive many failures.

3.4 Summary

3.4.1 Implication for concurrency bug detection

Traditional concurrency bug detection approaches solve the problem by looking forward along the concurrency bug's lifecycle – they focus on the complicated root cause and analyze complicated interleaving patterns to find the abnormal one. The detected abnormal interleaving may or may not be related to the critical read, thus may or may not cause a program to fail.

The characteristics study suggests that one can take a backward approach, specifically, we can start from identifying potential failure sites or error sites, whichever is easier. Thanks to our finding 1, it is easy to do, e.g., find all asserts or pointer dereferences. Then we can use sequential static analysis to locate the potential critical reads – because of finding 2 and finding 3, we do not have to analyze backwards very far in the same thread, thus this analysis is cheap. We then can use dynamic analysis to monitor potential critical reads and other related operations in the program run to check which interleaving might lead to wrong values at critical reads. In this way, the false positive is low, because when we report a bug, it is a bug that can cause program to fail. The false negative is also low, because we report bugs regardless of their root cause, on the contrary, existing bug detector is designed for one specific root cause.

3.4.2 Implication for concurrency bug failure recovery

We can first identify potential failures sites in a program, this is easy because of finding 1. Finding 2 implies that single-threaded rollback can allow programs to recover from most concurrency bug failures. Finding 3 implies that the checkpoint can be lightweight. In fact, we can leverage this observation and take the steps further so that no checkpoint is needed (details will be discussed in Chapter 5).

The following two chapters are based on the above implications and present our bug detection and failure recovery work in detail.

4 EFFECT-ORIENTED CONCURRENCY-BUG DETECTION

As discussed in Chapter 1, concurrency bugs cause severe damages in the real world. Effective approaches to detecting the concurrency bugs before the software is released are critical to concurrent software reliability. Existing techniques for detecting concurrency bugs focus on detecting the abnormal interleaving patterns (aka root causes) of concurrency bugs. Such techniques have two problems: false positives (reported bugs may not lead to any failure) and false negatives (bug detector designed for one specific interleaving pattern may not work for another).

This chapter will present ConMem and ConSeq, two new concurrency bug detection techniques, that provide a novel solution to concurrency bug detection problem. Guided by previous characteristics study of real-world concurrency bug life cycle (Chapter 3), ConMem and ConSeq share the same design philosophy and take the effect-oriented approach: they start from potential error or failure sites and detect concurrency bugs that can lead to those errors or failures. Such an approach differentiates ConMem and ConSeq from all the existing concurrency bug detection techniques.

ConMem detects memory errors caused by concurrency bugs. ConSeq (mainly) detects semantic errors caused by concurrency bugs. Together, they are able to find many real-world concurrency bugs of different root causes and severe failure symptoms. Some of the detected bugs have never been reported before.

4.1 Introduction

4.1.1 Motivation

A fundamental challenge in concurrency-bug detection is the enormous size of concurrent programs' interleaving space (exponential in the execution length for each input). Thoroughly checking this large space is crucial because concurrency bugs are only manifested under certain interleavings. Unfortunately, due to limited computational resources, software-development

teams can only afford to check a small part of this large space. Determining *which part of the interleaving space* should be checked is a critical open problem.

To address the above challenge, previous tools for concurrency bug detection and testing focus on certain interleaving patterns that are prone to concurrency bugs. Widely used patterns include data races (un-synchronized conflicting accesses to shared variables) [93, 111, 136, 41], atomicity violations (an interleaving that makes certain code regions unserializable) [40, 130, 75, 17, 42, 110].

Although great progress has been made, previous work still leaves some issues unsolved.

First, false negatives (*are there other patterns of buggy interleavings?*). Many patterns have been proposed, while common real-world bugs that cannot be covered by traditional patterns keep emerging, such as multi-variable concurrency bugs [73, 76, 60] and order violations [74, 134], as shown in Figure 4.1.

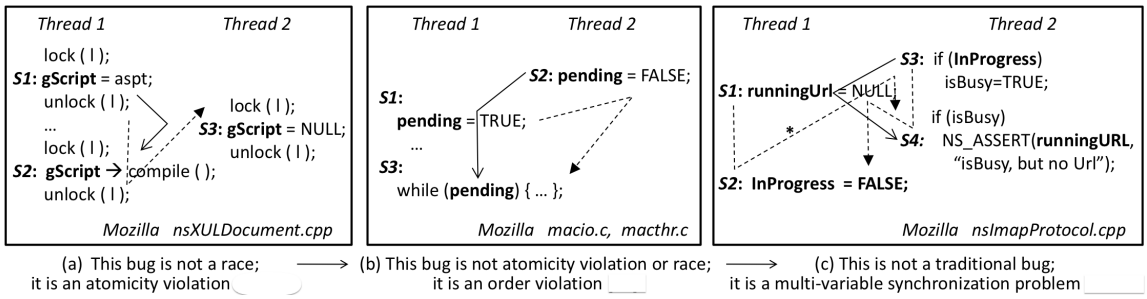
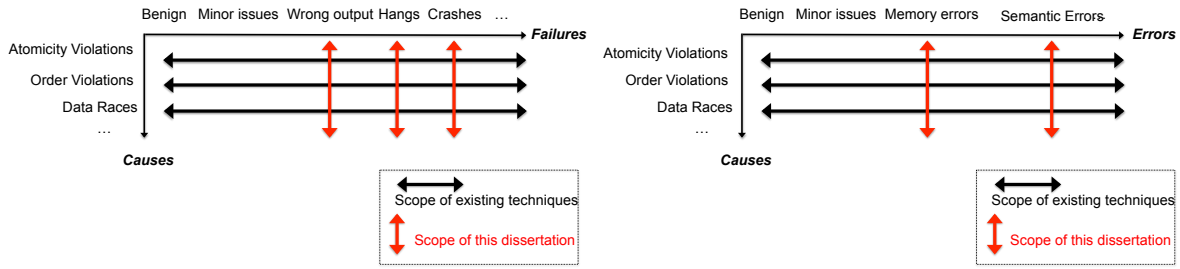


Figure 4.1: Bugs caused by various types of interleavings. (Solid and dotted arrows represent incorrect and correct interleavings, respectively. *: In (c), S1→S2→S3→S4 is a correct and feasible execution order, because `InProgress` could be set to TRUE and `runningURL` to a non-NULL URL string in between the execution of S2 and S3 by code not shown in the figure.)

Second, a high false-positive rate could cause programmers to give up on using a tool. Previous research [89, 12] observes that only approximately 2–10% of *real* data races are harmful; a similar trend is also seen among unserializable interleavings [99]. Interleavings are complicated to reason about. It is usually difficult for developers to judge whether a suspicious-looking interleaving is truly a bug, and if so, what type of failure symptoms it can cause. Some commercial bug-detection tools choose not to flag concurrency errors solely because the reported buggy interleavings are too difficult to explain to developers [9]!



(a) Cause vs Failure

(b) Cause vs Error

Figure 4.2: A conceptual two-dimensional depiction of approaches to finding flaws in concurrent programs

Finally, not all bugs represent equally harmful end effects, yet the different effects of different bugs are not considered during existing bug-detection processes. Recall that our characteristics study in Chapter 3 implies there is no correlation between bug root causes and their failure symptoms.

Figure 4.2 depicts the limitations of previous work (and our opportunities) by projecting a concurrent program's interleavings onto a two-dimensional space. The x-axis of Figure 4.2a represents the failure symptoms of concurrency bugs, the x-axis of Figure 4.2b represents the errors of concurrency bugs. The y-axis of Figure 4.2a and Figure 4.2b represent different patterns of interleavings (i.e., *faults*). Note that this is only a conceptual projection. The different categories along the y-axis can actually overlap; some horizontal stripes may have larger portions of benign effects than others.

Previous work has considered different *horizontal stripes* of the above 2-D space. These horizontal approaches inevitably suffer from the following limitations.

First, lack of good **coverage for certain type of failures**. Developers naturally want to know about all (or most) interleavings that can cause certain classes of negative effects, such as software crashes. Unfortunately, interleavings that cause certain effects span vertically in the space and are difficult to capture adequately through a horizontal approach.

Second, a large number of **false positives**. This is observed in the real world [12, 89, 99], and is reflected in Figure 4.2, where each horizontal stripe inevitably covers interleavings with benign effects.

We can deepen our understanding of the false positive and severity issues by connecting our findings of cause–effect chains in concurrent programs from our characteristics study in Chapter 3 with Figure 4.2. Interleaving patterns like data races and atomicity violations are only the start of potential error propagation chains. Some interleaving patterns do not propagate to any incorrect states (e.g., not every piece of code is intended to be atomic). For those that do cause incorrect states, their intermediate errors might be masked during further propagation (e.g., due to redundant paths [89]), or end up as a minor issue hardly visible to users. In many such cases, data races or unserializable interleavings are intentionally left there by developers for better performance (e.g., conflicting accesses to a performance counter [136]).

The false-positive issue has already caught the attention of many researchers. Various innovative approaches, such as training [75], automated testing [99, 113] and heuristics-based ranking, have been proposed to mitigate this problem. However, without changing the underlying horizontal mechanism, these proposals still require significant manual effort for specification writing and test-oracle design, as well as a large amount of computational resources to perform many rounds of testing or training.

In summary, this chapter presents a bug-detection approach that focuses on certain *vertical stripes* of the interleaving space, which cause severe failures, that spans across all kinds of (horizontal) interleaving patterns. This vertical approach will complement existing bug detectors and provide better guidance to expose severe concurrency bugs.

4.1.2 Highlights

Different from existing concurrency bug detection approaches, the approach proposed in this chapter uses potential software errors/failures to guide its search of the interleaving space. It uses an effect-oriented approach to bug detection, which starts from this observation[1]:

> *Concurrency and sequential bugs have drastically different causes but have mostly similar effects.*

[1] All the observations and characteristics mentioned here were discussed in more detail in Chapter 3.

For example, the three bugs in Figure 4.1 all start with complicated interleavings, which cannot be detected by many existing detectors [73], and end up as common errors and failures similar to sequential bugs. The bug in Figure 4.1(a) is caused by an atomicity violation and cannot be detected by race detectors [75]; it causes a NULL-pointer dereference, and crashes in thread 1. The bug in Figure 4.1(b) is caused by an order violation: S2 could unexpectedly execute before S1, in which case the `FALSE` value of `pending` would be overwritten too early. This problem cannot be fixed by locks, and cannot be correctly detected by atomicity-violation or race detectors [74]. It causes an infinite loop in thread 1. The bug in Figure 4.1(c) has a different cause from the above two bugs. Since S1→S2→S3→S4 and S3→S4→S1→S2 are both correct interleavings, the specific order between S1's and S4's accesses to `runningUrl` is not responsible for the software failure. Similarly, the specific order between S2 and S3 does not matter. It is the unsynchronized accesses to *two* correlated shared variables that lead to an assertion failure. Sophisticated multi-variable concurrency-bug detectors [73] were designed to detect this type of bug.

Recall our characteristics study summarizes the following observations on the error propagation of most concurrency bugs: After being *triggered* by an incorrect execution order across multiple threads, a concurrency bug usually *propagates* in *one* thread through a short data/control-dependence chain, similar to one for a sequential bug [49]. The erroneous internal state is propagated until an externally visible failure occurs. At the end, concurrency and sequential bugs are almost indistinguishable: no matter what the cause, a crash is often preceded by a thread touching an invalid memory location or violating an assertion; a hung thread is often caused by an infinite loop; incorrect outputs are emitted by one thread, etc. The only major class of concurrency errors that have substantially different characteristics is deadlocks, which fortunately have been thoroughly studied [59, 124].

Given the above observation and the trouble presented by the enormous space of interleavings, we naturally ask, "How can we leverage the sequential aspects of concurrency bugs? Can we reverse the bug-propagation process and start bug detection from the potential points of failure?" This approach, if doable, has the potential to leverage common error-propagation

patterns shared between concurrency and sequential bugs:

- In terms of false positives, the questions 'is this a bug?' and 'how harmful is this bug?' are easier to answer with this technique, because causing a failure is the criterion used for deciding whether or not to report a bug.
- In terms of false negatives, it can provide a nice complement to existing cause/interleaving-oriented detectors, because what this approach detects is not limited to any specific interleaving pattern.
- The bug reports are likely to be more accessible to developers who are familiar with the effects/symptoms of sequential bugs. Moreover, developers can now contribute to the bug-detection process by writing per-thread consistency checks, and by specifying which potential failures are worth more attention.

The effect-oriented approach to bug detection presents many challenges. Backward analysis might be straightforward for failure replay and diagnosis [2, 138, 29, 125, 101] when the failure has already occurred. However, it is much more difficult for bug-detection and testing, where we need to identify *potential* points of bugs and failures. The problem we face is more similar to proving whether a specific property in a concurrent program could be violated. This is known to be a hard problem in software verification, even when attempted on small programs, and explains why little work has been done in this direction.

In this dissertation, we propose a general framework that leverages the findings in the characteristics study presented in Chapter 3 and takes the effect-oriented approach. There are three major steps in this framework and work backwards along the life-cycle of a concurrency bug life-cycle. Step 1, identifying potential error/failure site. According to our characteristics study, such potential error or failure sites are similar to those in the sequential program and easy to identify. Step 2, identifying critical reads. According to our characteristics study, the distance from the critical reads to the failure sites are short, thus the search is feasible. Step 3, identifying suspicious interleavings. We use dynamic analysis to explore the interleaving space to narrow down the interleavings that cause critical reads to read wrong values. In addition, an optional bug exposing and validation step is provided to the developers to further prune

out the false positives and reliably trigger the bug. Figure 4.3 illustrates the work-flow of this framework along the error propagation process of concurrency bugs. Traditional approach takes a forward approach which starts from the faults of the concurrency bugs, whereas this dissertation takes a backward approach which starts from the potential error/failure sites.

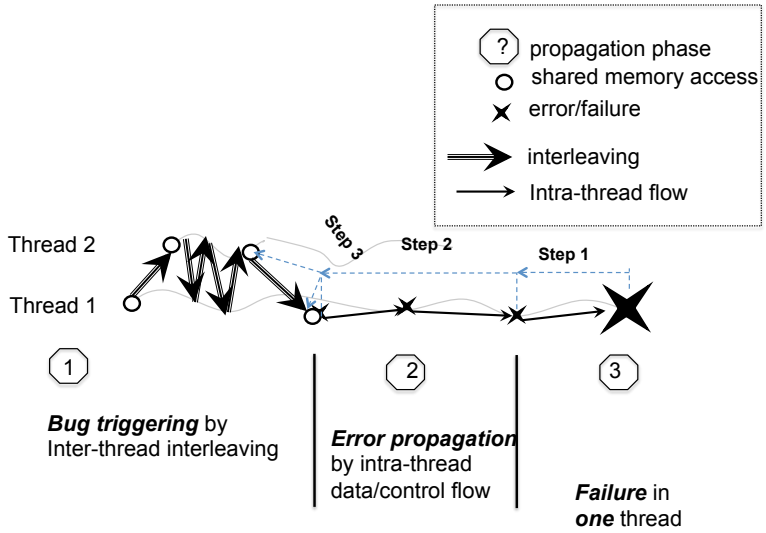


Figure 4.3: Effect-oriented concurrency bug detection framework workflow

Based on this general framework, two tools – ConMem and ConSeq – are instantiated. ConMem detects concurrency memory bugs and ConSeq (mainly) detects concurrency semantic bugs. Concurrency memory bugs and concurrency semantic bugs have different error propagation patterns, thus they require different detection techniques:

Per concurrency memory bugs' definition (Chapter 3.3.1), the critical reads and the potential memory error sites overlap and the error propagation distance is either 0 or 1[2]. To identify the critical reads, one can just identify potential memory error sites. Such sites can be structurally identified in a program, e.g., shared pointer dereference and shared buffer indexing. In comparison, concurrency semantic bugs' critical reads do not overlap with anything structurally identifiable. However, the failures caused by concurrency semantic bugs are structurally identifiable, as they are assert calls, error message display functions,

[2]For null pointer, dangling pointer and buffer overflow bugs, the error propagation distance is 1; for uninitialized read bugs, the distance is 0, as we treat the uninitialized memory reads as the potential error/failure sites.

output functions and the loop back-edges. Thanks to our characteristics study that the error propagation distance is short, it is feasible to locate concurrency semantic bugs' critical reads by conducting backward program slicing from the potential failure sites.

After the critical reads are identified, ConMem and ConSeq follow the similar steps to identify and expose the buggy interleaving. Table 4.1 summarizes what types of errors and failures each tool detects and the error propagation distance each tool allows.

Tool	Error type	Failure type	Error propagation distance
ConMem	null pointer dereferences uninitialized reads dangling pointers buffer overflows	crash	0 or 1
ConSeq	(mainly) semantic errors	assertion failures wrong outputs error messages hangs	Configurable, currently set as 4

Table 4.1: The errors, failures, and error propagation distance handled by ConMem and ConSeq

In our evaluation, we show that it is possible to apply the proposed techniques to large, real-world C/C++ programs (with millions of lines of source code and object files tens of megabytes long) and find the (unknown) bugs. In the following, Chapter 4.2 presents the details of ConMem. Chapter 4.3 presents the details of ConSeq. Chapter 4.4 concludes effect-oriented concurrency bug detection.

4.2 ConMem

4.2.1 Overview

ConMem detects concurrency bugs that cause memory errors. According to our characteristics study in Chapter 3, there are four major types of concurrency memory errors (concurrency memory errors are memory errors caused by concurrency bugs) – null pointer dereferences, uninitialized reads, dangling pointers, and buffer overflows. ConMem includes four dynamic bug-detection modules that are responsible for detecting each of those four types of memory bugs: Con-NULL, Con-UnInit, Con-Dangling, and Con-Overflow bugs, respectively. The design of ConMem is guided by the findings of our characteristics study, and follows the

effect-oriented design principle. Instead of interleaving-oriented, ConMem tries not to analyze an interleaving pattern unless it is related to concurrency-memory errors. Moreover, ConMem does not limit itself to any specific interleaving pattern.

ConMem also has following design goals: (1)Predictive bug detection. ConMem bug detection is not limited to the monitored interleaving. Instead, it aims to report concurrency bugs that could occur under future interleavings. This property is critical due to concurrent programs' non-determinism. (2) Balance between analysis accuracy and complexity. Because the optional validation step can help prune out false positives, ConMem has the luxury of trading accuracy for simplicity, when necessary.

To follow the design principle and achieve the design goals, ConMem dynamically and predicatively detects concurrency-memory errors in two steps.

First, it identifies basic ingredients of concurrency-memory errors from a monitored program execution. The basic ingredients are memory operations, such as a pointer dereference, a NULL assignment, a buffer deallocation, etc. Their existence is necessary to a concurrency-memory error and is (fortunately) usually insensitive to interleavings. They will be detected by the memory checking part of ConMem.

Second, it analyzes whether special timing conditions can be satisfied among those basic ingredients during future execution. Special timing, such as de-allocating a memory object *before* another thread accesses it, can turn a set of memory operations into a true bug. Whether a timing condition can be satisfied in future interleavings depends on the synchronization operations in the program. The synchronization-analysis part of ConMem is responsible for making this decision and reporting bugs.

A summary of the ingredient-and-timing conditions for each sub-type of concurrency-memory error is shown in Table 4.2. The following chapters will elaborate on how to detect each sub-type of concurrency-memory error.

	Error Conditions		Can synchronization avoid the error?	
	Basic Ingredients	Timing Condition	Order Synch.*	Mutual Exclusion
Con-NULL	(1) rp: from t1, reads pointer ptr (2) wp: from t2, writes NULL to ptr	(1) wp executes before rp (2) No write to ptr between rp, wp	Yes	Yes
Con-UnInit	(1) r: from t1, reads variable v (2) $\nexists w$: from t1, writes v before r (3) w: from t2, initializes v, usually before r	r executes before w	Yes	Not by itself
Con-Dangling	(1) a: from t1, accesses memory m (2) Free(M): from t2, $m \in M$	a executes after Free(M)	Yes	Not by itself
Con-Overflow	(1) v: a buffer-index/boundary var. (1) a1: from t1, accesses v (2) a2: from t2, accesses v	Data race between a1 and a2 (approximated condition)	Yes	Yes

Table 4.2: The conditions for Concurrency-Memory errors. (*: order synchronization represents barrier-style synchronizations).

4.2.2 Con-NULL Detection

What is a Con-NULL bug?

Con-NULLs are NULL-pointer dereference errors directly caused by buggy interleavings. An example of Con-NULL is shown in Figure 3.4. As we can see there, S2 from thread 1 dereferences a shared pointer variable `thd→ proc_info`, and S3 from thread 1 assigns NULL to the same variable. Under a buggy interleaving, S3 executes right between S1 and S2, immediately causing a NULL-pointer dereference and MySQL crashes. Of course, the above buggy interleaving occurs only rarely, and MySQL behaves correctly most of the time.

In general, the *basic ingredients of Con-NULL bugs* include two pointer accesses, denoted as `wp` and `rp`. `wp` writes NULL to a shared pointer variable `ptr`, and `rp` reads `ptr` from a different thread that later performs a pointer dereference. We consider each {`wp`,`rp`} pair to be a bug suspect.

The *timing condition of Con-NULL* is to execute `wp` before `rp` with no other write to `ptr` in between. A bug suspect is reported only if the timing condition can be satisfied.

Con-NULL detection algorithm

The algorithm includes two parts.

Detecting the basic ingredients Building a run-time monitoring tool to identify {`wp`, `rp`} pairs is straightforward using binary instrumentation. Specifically, for every heap/global

access, [3] ConMem collects its thread-id, program counter, memory location, and store-value information at run-time. Analyzing this information can easily reveal Con-NULL suspects. The only issue remaining is to differentiate memory locations that hold pointers from those that hold normal integer or Boolean variables. This matter will be discussed later.

Checking the timing condition After a Con-NULL error suspect (i.e., a {`wp`,`rp`} pair) is discovered, the next step is to check whether the synchronization operations in the program allow `wp` to execute before `rp` without another interfering definition in between.

Without losing generality, ConMem separately considers *mutual-exclusion* synchronization and *order synchronization*. If the timing condition explained above is not prohibited by either of them, the corresponding suspect will be reported as a Con-NULL bug.

Order-synchronization operations [93, 99], such as barriers, set up a happens-before partial order among all accesses in the concurrent execution. Under this happens-before order, two accesses are either strictly ordered or concurrent with one another.

Order synchronization could make a Con-NULL timing condition infeasible if and only if one of these two conditions are satisfied: (1) the NULL-assignment is strictly ordered after the pointer read; or (2) another write to the pointer is strictly ordered between the NULL-assignment and the read. The 'order' here is determined by the happens-before relationship.

Mutual exclusion, such as locks and transactions, prevents those code regions that are protected by the same lock or covered in transactions from interfering with one another.

Mutual exclusion could protect the {`wp`,`rp`} pair and prevent a Con-NULL error in two ways: (1) `rp` and an earlier write to `ptr` from the same thread are atomic with respect to `wp`; or (2) `wp` and a later write to `ptr` from the same thread are atomic with respect to `rp`. In the former case, `rp` always uses a definition from its own thread, instead of `wp`. In the latter case, `wp`'s assignments are always overwritten before reaching `rp`.

ConMem monitors mutual-exclusion and order synchronizations at run time. By checking against the above conditions, ConMem can identify Con-NULL suspects that are properly

[3] Data stored on the stack is not usually shared across threads and is therefore ignored in our current prototype.

protected and report the remaining suspects as Con-NULL bugs.

Note that, the above analysis is different from traditional data race checkings. A {`wp`,`rp`} pair does **not** need to be a data race in order to be a Con-NULL bug. As discussed above, a Con-NULL bug could occur between a `wp` and a strictly happened-after `rp`, which is not a data race; a Con-NULL bug could also occur between a `wp` and a `rp` that are protected by the same lock variable. The same is true for Con-UnInit and Con-Dangling. In fact, ConMem can detect many bugs that cannot be caught by race detectors, as shown in the Table 4.11.

Of course, our synchronization analysis is neither sound nor complete, because it does not consider potential control-flow changes under future interleavings. We believe it provides a good balance between analysis complexity and analysis accuracy, as shown by our experimental results in Chapter 4.2.9.

Implementation

ConMem implements the above algorithm using run-time recording (with PIN [80] binary instrumentation) and off-line trace analysis. We choose trace analysis over pure run-time detection due to the algorithm complexity.

The run-time component logs three types of information. The first type is information about accesses to a global or to heap memory, which is used to identify basic ingredients (i.e., {`wp`, `rp`}). The second type is the synchronization operations, including `barrier`, `pthread_mutex_(un)lock`, `pthread_create/join`, etc. This part is used to check suspects' timing conditions. The last type is information about all malloc/free operations. Since virtual addresses could be recycled through malloc/free, the latter information helps us to identify which memory locations are truly holding the same memory object. The recycling issue is similarly handled in the three remaining detection modules.

Con-NULL only needs to record and analyze memory accesses to pointer variables. Our current implementation differentiates pointers from non-pointer variables based on the value stored in a memory location. That is, an access to a memory location `m` is ignored by Con-NULL if the value stored in `m` is neither 0 nor within the range of the stack, the heap,

or the global data region. This scheme works well in practice.

The trace-analysis includes three major steps: (1) identify all {wp,rp} pairs; (2) analyze mutual-exclusion synchronization; and (3) analyze order synchronization.

The first step is straightforward. By checking the memory-address, thread-id, and store-value information in the trace, we can easily find all Con-NULL suspects.

The second step is to analyze mutual-exclusion synchronization. Following our earlier discussion, for every suspect {wp, rp} pair, ConMem identifies the preceding write of rp (refer to as rp-p) and the follow-up write of wp (refer to as wp-f) from the trace. It then calculates the lock-sets that protect rp, {rp-p,rp}, wp, and {wp,wp-f}. Any lock-set overlap between {rp-p,rp} and wp or overlap between {wp,wp-f} and rp indicates that this suspect is well-protected and should not be reported as a bug.

The last step is to determine whether order synchronizations can protect a {wp, rp} pair from NULL-pointer dereference. This analysis is conducted through vector timestamp comparisons.

Our run-time updates and logs the vector timestamp of each thread right after every *order-enforcing* synchronization operation, including pthread_mutex_create/join and barriers, based on the Lamport logical-timestamp algorithm [64]. During trace analysis, we can easily obtain the vector timestamp of each memory access a in thread t, which is the latest timestamp logged before a in the log of t.

With the timestamp information, we want to check (1) whether wp will always execute after rp, and (2) whether wp will always be overwritten before it reaches rp. If neither is true, a Con-NULL bug is reported. This checking could be time-consuming, because for each suspect {wp, rp} pair that accesses memory location ptr, it requires comparing their timestamps with the timestamp of *every* write access to ptr. Our implementation simplifies this checking using a heuristic: if there exists a ptr-definition that is strictly ordered between wp and rp, it usually comes from either the thread of wp or the thread of rp. Using this heuristic, we only need to check two candidates that might sit between rp and wp: the write to ptr on rp's thread right before rp and the write to ptr on wp's thread right after wp. Overall,

our implementation has a modest complexity, linear in the number of suspect {`wp`, `rp`} pairs, and works well in our bug-detection experiments, never introducing false positives.

Discussions Con-NULL predicts concurrency bugs that could occur in the future based on the observation of one program execution. This prediction inevitably has false positives and false negatives.

The *false positives* of Con-NULL detection mainly have two sources. The first is unidentified custom synchronization, an issue shared with many previous concurrency-bug-detection tools [111]. Without knowledge about some custom synchronization operations, such as spin loops and producer-consumer queues, ConMem will mistakenly consider some timing conditions as feasible and report false positives. Chapter 4.2.6 discusses how to prune some of these false positives. The second sources of false positives are due to simplifications made by our implementation. One simplification that has not yet been mentioned is that we do not check whether a pointer read is used for dereferencing. Sometimes, a pointer read is used for condition-checking, where reading a NULL-valued pointer does not cause any problem. We prune out this type of false positive by checking whether a pointer read has a NULL value during the monitored run. If it does, we do not report the bug. This pruning has been very effective, as we will see in Chapter 4.2.9.

The *false negatives* of Con-NULL detection mainly come from the code/path coverage problem. Under a fixed input and different interleavings, the predicate variable of a branch could have different values and lead to different execution paths. If an instruction is executed only under rare interleavings or if two instructions access the same memory location only under rare interleavings, ConMem may miss the basic ingredients of potential Con-NULL bugs and have false negatives. This type of false negatives exist in all ConMem detection algorithms and also previous work that tries to predict future interleavings based on one observed interleaving [111, 40, 17, 58, 131, 99]. Fortunately, it rarely occurs in practice, based on our experience. In addition, this problem can be mitigated by making ConMem observe more than one run of the program under the same input and analyze each run independently. If one of the runs reaches a path that can only be observed in a rare interleaving, then

ConMem is able to report bugs on this path. ConMem can also benefit from techniques that improve the testing code coverage in concurrent programs [114].

Finally, *trace size* is a potential concern for all trace-based analysis tools. Since Con-NULL only records heap/global memory accesses that touch (likely) pointer variables, its traces will be significantly smaller than those generated by deterministic replay tools [101]. Based on our experience, it is rarely a problem for Con-NULL, as shown in Chapter 4.2.9. One could also split the trace of a long-running program into several sub-traces and apply the Con-NULL algorithm to each sub-trace.

4.2.3 Con-UnInit Detection

What is a Con-UnInit bug?

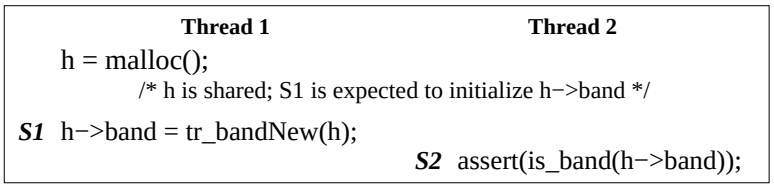


Figure 4.4: A concurrency bug that leads to an undefined read and finally causes crash (from Transmission-1.42)

Con-UnInit bugs are un-initialized memory reads directly caused by buggy interleavings. An example of a Con-UnInit bug is shown in Figure 4.4. In this example, a shared variable `h→ bandwidth` is initialized at S1 in thread 1. Read accesses to this variable are supposed to occur after S1. Unfortunately, without proper synchronization, S2 in thread 2 can execute before S1 and read an uninitialized value, which causes an assertion failure later.

The *basic ingredients* of a Con-UnInit bug typically include a read access, denoted as `r` (e.g., S2 in Figure 4.4), to a memory location that should be initialized by another thread. The *timing condition* for a Con-UnInit bug is to execute `r` before the initializations by another thread.

Note that, when we observe an `r` reading a value defined by its own thread, an un-initialized read is unlikely to happen under a different interleaving. However, there could be exceptions.

For example, future interleavings could change the execution path and make the local definition disappear. This goes beyond our definition of concurrency-memory errors and is not considered here.

Detection algorithm & implementation

Con-UnInit's detection algorithm is simpler than Con-NULL's and is implemented via run-time detection without trace analysis.

Detecting the basic ingredients This task identifies a shared-memory read, the target memory location of which is *not* defined earlier in its own thread, but in another thread. Such reads will be considered as Con-UnInit suspects.

This task is quite straight forward to implement during dynamic monitoring. Relying on the PIN instrumentation framework, we use a hash-table *Initializer* to maintain the per-thread information about which memory locations are already initialized in this thread. Specifically, *Initializer* is indexed by memory locations. Whenever a write to memory location ν occurs, *Initializer* is checked to determine whether this is the first write to ν from that thread. If it is, the information of this write is inserted into the table. Looking up *Initializer* at every read access to a heap variable will reveal all Con-UnInit suspects.

Checking the timing condition At run-time, whenever a read suspect `r` is discovered, ConMem must conduct a synchronization analysis and decide whether there exists a remote initialization that is strictly ordered before `r`. Mutual exclusion cannot help to avoid this type of bug and is not considered here.

Conducting this task at run-time requires several pieces of information. Suppose that the suspect `r` accesses memory location ν. The first piece of information we need is the vector timestamp of `r`. ConMem maintains the vector timestamp for each thread at run-time, by intercepting order synchronizations (i.e., barrier and `pthread_create/join`) and analyzing them based on the classic Lamport algorithm [64]. The timestamp of `r` can be easily retrieved from the current timestamp of its own thread.

The second piece of information is the vector timestamp of all the initializations to ν from

other threads. This information is kept in the *Initializer* table mentioned above. Specifically, when a write access is found to be the first write to v from thread t, t's current timestamp is inserted into *Initializer*.

Finally, after obtaining the above information, ConMem compares the timestamp of r with the timestamps of remote initializers. A Con-UnInit bug is reported when r is *concurrent* with all the recorded initialization timestamps.

As an optimization, we only conduct the above check for the first read from each thread to a memory location v. This is sufficient to detect Con-UnInit bugs on v, if they exist.

Discussions The sources of false negatives and false positives for Con-UnInit detection are similar to those of Con-NULL, except for one unique source of false positives. That is, some un-initialized reads may not cause negative effects, a property different from NULL-pointer dereferences, dangling pointers, and buffer-overflows. Previous sequential bug detectors, such as Valgrind [91], have considered this and choose to report bugs only when the un-initialized value is used for critical operations, including system calls, condition checking, and memory-address calculation. ConMem could borrow this idea to prune this set of false positives, but this is not included in the present implementation.

In contrast with Con-NULL, Con-UnInit does not dump traces and does not have the trace-size issue. However, since Con-UnInit conducts all its analysis on-line, its run-time analysis will consume more memory than Con-NULL. The memory consumption of Con-UnInit is mainly for storing the initialization timestamp for each active heap/global memory location. It is linear in the heap/global memory footprint of a program, like many previous dynamic bug detectors [75]. It will *not* increase with longer executions, as long as the program's active memory consumption does not change.

4.2.4 Con-Dangling Detection

What is a Con-Dangling bug?

A Con-Dangling bug occurs when buggy interleavings directly cause dangling pointer accesses. Figure 4.5 demonstrates a bug from PBZIP2. In this example, pointer q (a local variable in

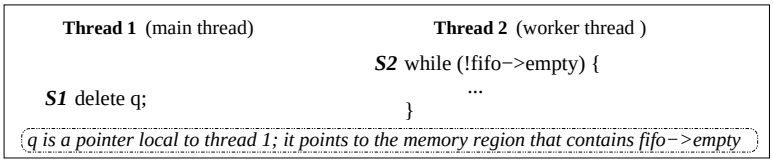


Figure 4.5: A concurrency bug that leads to a dangling pointer and finally causes crash (from PBZIP2-0.9.4)

thread 1) points to a heap object shared by thread 1 and thread 2 (`fifo` in thread 2 points to the same object). Due to lack of synchronization, thread 2 can access the shared object at S2 when it is already deleted by thread 1 at S1, which can cause PBZIP2 to crash.

As we can see, the *basic ingredients* of a Con-Dangling bug is a memory access whose target memory location is de-allocated by a different thread. The *timing condition* of Con-Dangling is to conduct the memory access after the de-allocation.

Detection algorithm & implementation

Similar to Con-UnInit detection, Con-Dangling is implemented in PIN as a pure run-time bug detector with no trace analysis.

The algorithms of **detecting basic ingredients** and **checking timing conditions** are straightforward here. For the first task, we must identify all memory accesses whose target memory locations are de-allocated by a different thread. For the second task, we must analyze order synchronizations to determine whether the accesses are concurrent with the de-allocation operation. Just like with Con-UnInit, mutual exclusion itself cannot avoid Con-Dangling bugs and is not considered in the following.

In our PIN-based implementation, every `malloc` and `free` invocation is intercepted, in addition to every order synchronization and heap access. A map `Malloc_Map` is used to maintain a list of currently active heap memory regions, ordered by their starting addresses. A new entry is inserted in `Malloc_Map` at every `malloc`. At every heap access, ConMem looks up `Malloc_Map` with the accessed heap address to find the corresponding entry, and then updates the entry to record the latest access from each thread to each memory region. Whenever a `free` is invoked, the timestamp of this `free` will be compared with the timestamps of the

latest accesses to this to-be de-allocated memory region from each thread. A Con-Dangling bug is reported when we find a concurrent access (based on timestamps) from a different thread.

4.2.5 Con-Overflow Detection

What is a Con-Overflow bug?

Buffer overflow occurs when a buffer access goes beyond the buffer boundary. In concurrent programs, interleavings can cause additional buffer-overflow problems when buffer-index or buffer-boundary variables are shared among different threads.

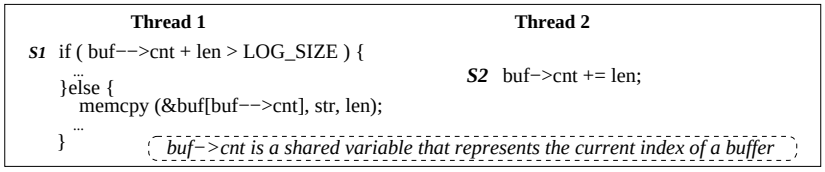


Figure 4.6: A concurrency bug that can lead to a buffer overflow and subsequent crash (from Apache-2.0.45)

Figure 4.6 shows an example of a typical Con-Overflow bug. Thread 1 conducts a sanity check at S1 on buffer index variable `buf→cnt` to ensure that the later `memcpy` will not overflow the buffer `buf`. Unfortunately, the index variable is shared with thread 2. Due to lack of synchronization, thread 2 can change the buffer index between the sanity check and the real buffer access, thus causing a buffer overflow.

Accurately reporting Con-Overflow bugs is difficult because exposing buffer-overflow bugs requires not only a certain order of memory operations, but also certain variable values. Even when an index variable is unexpectedly corrupted by a different thread, buffer overflow may not occur, depending on the new value stored into the index. In the future, symbolic-execution and constraint-solving techniques [13] can potentially address the issue of identifying whether problematic values can arise.

In our current prototype, we only consider a common subset of Con-Overflow bugs: conflicting accesses to shared buffer-index variables cause buffer overflows. Specifically, we report all data races on shared buffer-index variables as potential Con-Overflow bugs, and

we rely on our ConMem-validator (Chapter 4.2.7) to prune out false positives. We leave the more general Con-overflow detection problem to future work.

Detection algorithm & implementation

Con-Overflow detection includes two steps. The first step detects data races in the execution. The second step attempts to identify accesses to buffer-index variables among those data races.

The first step is conducted through an existing lock-set algorithm [111]. The second step can be conducted in different ways. Our solution is based on the heuristic that an index variable should be used to generate buffer-access addresses sooner or later. Currently, we implement this step as an additional run of dynamic data-dependence analysis. That is, after we have information about data races in hand, the program is executed a second time. Whenever a memory location involved in a race is read, the dependence analysis starts, tracking the data flow to determine whether the read value would be used to generate a global/heap address within a threshold number of steps. In addition, we also make sure the read value itself is not already a global/heap address. Full dependence-analysis has large overhead, since we need to keep track of both local and shared memory accesses. Fortunately, we only need to track those accesses and memory locations related to races and currently we set the number of steps to track as 3. Therefore, the overhead is acceptable.

Our current implementation of Con-Overflow requires two runs of the program – one to find races and one to perform dependence analysis. We expect that the second run is not always necessary. After one variable or one instruction is marked as accessing (or not accessing) a buffer index, this information can be kept for future use. Static analysis can also help identify instructions that access buffer-index variables and potentially remove the second run of the program.

In summary, ConMem bug detection includes four sub-tools. Con-UnInit and Con-Dangling bugs are detected and reported at run-time. Con-NULLs and Con-Overflow bugs are reported after a post-mortem analysis. It is also conceivable to combine all these four modules into one

big run-time bug-detection tool in the future.

4.2.6 Handling Spin-Loop Synchronizations

As discussed in Chapters 4.2.2 and 4.2.3, a major source of false positives in ConMem is custom-synchronization operations, as demonstrated by Figure 4.7(a). We here discuss how to handle one common type of custom synchronization, synchronization loops (also called spin loops). The algorithm presented below is an *optional* step in ConMem. It is neither sound nor complete. Its usage in practice will be evaluated in Chapter 4.2.9.

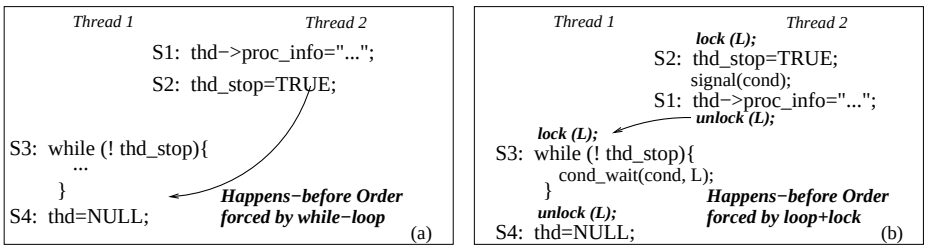


Figure 4.7: Examples of spin-loop synchronization (thd_stop is a volatile variable). (a) A NULL-pointer dereference can never occur between S4 and S1, because thread 1 cannot execute S4 until its S3-loop is terminated by S2 in thread 2. (b) Synchronization is achieved by a spin loop **and** locks. Without locks, the execution order between S1 and S4 is not fixed; with locks, S1 will always be executed before S4 just as that in (a). Note, *cond_wait* implicitly releases the lock L, thus there is no potential deadlock.

Spin-loop identification

This analysis algorithm is inspired by SyncFinder [129], and involves two steps.

First, *identifying loops.* This step is conducted through CodeSurfer/x86 [5], a static-analysis framework for x86 executables. CodeSurfer/x86 identifies every loop in the program's control-flow graph. To identify nested loops, it implements *Bourdoncles*'s algorithm [10], which recursively decomposes an SCC into sub-SCCs, etc. For each loop, we use CodeSurfer/x86 to identify all (conditional) jump instructions that jump out of the loop, referred to as *loop-exit jumps*. We then use static slicing, also a functionality supported by CodeSurfer/x86, to find all read instructions in the loop for which there is a path of control-dependence or

data-dependence edges from the read to a loop-exit jump. We refer to these read instructions as *potential loop-exit reads*.

Second, *identifying synchronization loops.* This step is conducted through run-time analysis — a loop that is always terminated by reading a value defined by a different thread is considered to be a synchronization loop.

To conduct this analysis, we record a trace of three types of instructions at run-time: (1) all potential loop-exit reads; (2) all loop-exit jumps; (3) all instructions in the program that write global or heap variables.

In trace analysis, we first identify *the* loop-exit read $\mathtt{r}$ for each loop $\mathtt{L}$ — a potential loop-exit read that obtains the same value from a variable v in all but the last iteration of $\mathtt{L}$. We then identify the write w that defines the value read by $\mathtt{r}$ in the last loop iteration. $\mathtt{L}$ is considered to be a synchronization loop if w always comes from a different thread than $\mathtt{r}$. In that case, w, such as S2 in Figure 4.7(a), is marked as a synchronization write, and the loop-exit read, such as S3 in Figure 4.7(a), is marked as a synchronization read. We can execute the program several times to prune false positives. If a loop is ever observed to be terminated by a definition from the same thread, it will never be considered to be a synchronization loop. If the value of v changes from non-loop-exiting to loop-exiting for more than once in one run, the corresponding loop will never be considered to be a synchronization loop. Actually, this type of loop likely belongs to a custom lock implementation, which our current implementation does not handle.

Note that, how to accurately identify all custom-synchronization operations is an open problem in concurrency-bug detection [118, 17, 129]. Our approach is inspired by SyncFinder [129]. SyncFinder identifies synchronization loops purely based on static analysis. We use dynamic analysis at the second step, which suits the dynamic nature of ConMem. Dynamic analysis also gets us around the challenges of pointer alias analysis and statically figuring out which code regions could execute concurrently.

Like previous work that tries to identify custom-synchronization operations [118, 17, 129], our analysis is neither sound nor complete, because it makes decisions based only on the runs

that are observed in the run-time analysis. A loop that can be terminated by a write from its own thread may never be observed to exit in that manner, and thus will be mistaken for a synchronization loop. A loop that is sometimes used for synchronization and sometimes not is always considered to be a non-synchronization loop by us.

Integrating synchronization-loops into ConMem

A synchronization loop is one type of 'order synchronization' discussed in Table 4.2 — it forces a happens-before order between operations before the synchronization write in one thread and operations after the synchronization loop in another thread. Because the synchronization analysis in ConMem already covers order-synchronization operations, here we only discuss how to adjust the logical time-stamps given synchronization-loop information. After properly adjusting the time-stamps, ConMem can easily prune the false positives that would otherwise be reported for the examples in Figure 4.7.

When ConMem monitors a test run, we instrument not only normal synchronization operations, such as `pthread_mutex_(un)lock` and `pthread_join`, but also every synchronization read/write and exit jump of each synchronization loop. At run-time, we maintain a hash-table indexed by memory locations. Whenever a synchronization write w is executed by thread t on memory location m, the m entry in the hash-table is updated with {t, t's current time-stamp}. Whenever a synchronization read in thread t' is executed, we look up the information about its definition write in the hash-table. This information will be used to update t''s time-stamp, whenever it exits a synchronization loop.

Sometimes, locks can be used together with spin-loops to achieve synchronization, as demonstrated in Figure 4.7(b). ConMem considers this interaction between mutual-exclusion synchronization and order synchronization, and adjusts the time-stamp update accordingly.

We provide the above analysis as an option to ConMem users. We evaluate its effect in Chapter 4.2.9.

4.2.7 Bug Exposing and Validation

ConMem-v is the bug validator component of ConMem. The design of ConMem-v is inspired by previous tools that validate data-race [98] and atomicity-violation bug reports [99]. ConMem-v takes every bug report from ConMem as its input. It tries to trigger the buggy interleavings predicted in ConMem's bug reports by carefully perturbing the concurrent execution. The whole process is automated.

ConMem-v serves two purposes. The first is to prune false positives that are caused by customized synchronization and by some of the approximations made by ConMem's detection algorithms. The second is to provide developers with a reliable way to repeat the true bugs reported by ConMem.

In the following, we discuss the design and implementation of ConMem-v, explaining what is the interleaving enforcement target and how to provoke a specific timing condition. ConMem-v is implemented using PIN [80] binary instrumentation. For the sake of brevity, some implementation details are omitted.

Validating Con-NULL reports From a {`wp`,`rp`} pair of a Con-NULL bug report, ConMem-v aims to execute `wp` before `rp`, with minimized timing distance in between.

To enforce such a timing condition, ConMem-v instruments the binary code right before and after `wp` and `rp`. At run-time, whenever `wp` or `rp` is to be executed, ConMem-v checks whether the other instruction has already 'arrived'. If so, `wp` will be arranged to execute first, immediately followed by `rp`. If not, an artificial delay (several iterations of `usleep`) is added to the current thread, in the hope that the other instruction will arrive from a different thread. This process is illustrated in Figure 4.8 (consider A as `wp`, B as `rp`).

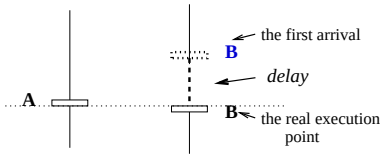


Figure 4.8: Illustration of how ConMem-v perturbs execution

Note that, as a general principle in ConMem-v, ConMem-v only improves the chances of a

bug to occur and does not provide any guarantee. *All* the delays inserted by ConMem-v have time-outs, so that the program will not hang.

Validating Con-UnInit reports The input to Con-UnInit validation is a list of instruction pairs {w,r} from the Con-UnInit bug report. w is an instruction that initializes a memory location that is later read by r from a different thread.

ConMem-v's target here is to execute w after r. To achieve this target, ConMem-v instruments the binary code to postpone the execution of w in an attempt to wait for r to execute first (consider r as A and w as B in Figure 4.8). ConMem-v can keep track of all heap/global writes to know whether an uninitialized read has truly occurred. In practice, just observing whether r is executed before w very likely already tells users whether the Con-UnInit bug report is a true bug.

Validating Con-Dangling reports The input to Con-Dangling validation is a list of instruction pairs {F,a}. F is a `call` instruction that invokes a de-allocation operation on a memory region that contains the memory location accessed by a from a different thread.

ConMem-v's target here is to postpone the execution of a in an attempt to have the F occur first, as illustrated in Figure 4.8 (F is A, a is B). To know whether a dangling pointer has been produced, ConMem-v records and compares the memory address accessed by a and the range of the memory region freed by F.

Validating Con-Overflow reports The input to Con-Overflow validation is a list of data-race pairs {i1,i2}. i1 and i2 race upon a shared buffer-index variable. The target of ConMem-v is to make the race truly occur (i.e., first execute i1 *right before* i2 without any other instruction in the middle and then i2 *right before* i1) and observe what happens after the race.

ConMem-v's perturbation strategy for Con-Overflow bugs is similar to those for the three discussed above. The unique complexity of Con-Overflows is that even if a buffer index is corrupted to an incorrect value through a data race, overflow may not happen. In our current validator, we look for fail-stop symptoms (crash or assertion failure) to tell whether buffer overflow has happened, which can be improved by more accurate buffer-overflow detection techniques designed for sequential programs [91, 51].

In the end, a ConMem bug report is generated. It includes the conflicting instruction pair, their corresponding call stacks, and the bug category (Con-NULL/Con-UnInit/Con-Dangling/Con-Overflow). When ConMem-v successfully exposes the bug, the bug report also includes the corresponding failure-triggering thread-scheduling, i.e. where and how long are the injected delays.

Discussion Two types of interleaving-enforcement approaches were proposed before. One is to execute programs on single-core machines and control the scheduling [85, 113]; the other is to insert artificial delays [99, 31]. ConMem-v chooses the latter for more effective use of the existing multi-core machines.

In summary, ConMem-v does not report false positives. In addition, benefiting from the clear error-pattern of memory bugs, ConMem-v does not need manually written oracles to judge whether a bug has occurred. ConMem-v could have false negatives: it may miss some bugs whose manifestation requires very sophisticated interleaving manipulation.

4.2.8 Evaluation Methodology

Applications ConMem is evaluated using 7 widely-used C/C++ applications, including 3 server (Apache HTTP server, MySQL data base server, and Cherokee HTTP server), 3 desktop (Mozilla web browser, PBZIP2 parallel decompressor, and Transmission bittorrent client) and 1 scientific application from SPLASH2 (FFT) [128].

Apart from these 7 applications, ConMem is also evaluated on the latest version of a multi-threaded software system, Click [21], for which no concurrency bug was previously known. ConMem uses the standard test inputs released by Click developers and is able to find previously unknown concurrency bugs. The detailed set-up and results are presented in Chapter 4.2.9.

Bugs in evaluation For evaluation, we use 10 real-world concurrency bugs[4] that were introduced by the original developers of the above 7 applications. 9 out of these 10 bugs can cause client and server crashes. We carefully set up this bug set to make sure it is

[4]One of these 10 bugs, PBZIP2-2, was not reported in previous documents. It was first detected in our ConMem experiments. It can be fixed by the same patch that fixes PBZIP2-1.

representative, covering different types of faults and error-propagation patterns, as shown in Table 4.10. One of these 10 bugs does not lead to software crash. It was introduced by external library developers of FFT. This FFT bug will help measure the false-positive rate and overhead of ConMem on scientific applications.

Bug-ID	**Causes**	**Effect Description**	**Software version**
MySQL-1	Atom.	Server crash at NULL-ptr dereference	MySQL-4.0.19
MySQL-2	Atom.	Server crash at NULL-ptr dereference	MySQL-5.1.28
PBZIP2-1	Order/Atom.	Crash at NULL-ptr dereference	Pbzip2-0.94
Apache-1	Multi-Atom.	Crash due to dangling ptr	Apache-2.0.46
Mozilla	Multi-Atom.	Crash due to dangling ptr	Mozilla-JS1.5
PBZIP2-2	Order	Crash due to dangling ptr	Pbzip2-0.94
Apache-2	Atom.	Crash/corrupted-log due to overflow	Apache-2.0.46
Cherokee	Atom.	Crash/wrong-message due to overflow	Cherokee-0.9.2
Transmission	Order	Crash due to uninitialized read	Transmission-1.42
FFT	Order/Atom.	Wrong output due to uninitialized read	N/A

Table 4.3: 10 bugs in evaluation (Atom.: single-variable atomicity violation; Order: order violation; Multi-Atom.: multi-variable involved atomicity violation.)

Experiment setup The experiments are conducted on dual quad-core Intel Xeon (2.67GHz) machines, with Linux, version 2.6.18. We use the PIN [80] binary instrumentation framework for all our tools. We use Valgrind–Helgrind [91] as the race-detection front-end for Con-Overflow.

Our experiments use bug-triggering inputs reported by the user, like previous dynamic concurrency-bug detectors [130, 75]. Note that the bugs **never** manifest during our bug-detection runs. Actually, many concurrency bugs do not manifest even after multiple days' worth of execution with bug-triggering inputs [99, 85], which is exactly why ConMem's predictive detection will be useful.

Our evaluation executes each bug-triggering input (or a set of bug-triggering client requests) to the end in order to measure both false positives and performance. The reported performance numbers are the averages across multiple runs. By default, the special algorithm for custom-synchronization (Chapter 4.2.6) is **not** applied. We evaluate how that algorithm further improves the accuracy of ConMem in Chapter 4.2.9.

ConMem includes four sub-tools for four types of concurrency-memory errors. Each application was executed with the bug-triggering input once for each sub-tool. We present

the bug-detection results for each sub-tool. When ConMem is compared with other detection tools, the true bugs as well as the false positives from all four sub-tools are put together. The artificial delay used by ConMem-v is 1 millisecond at a time.

We also compare ConMem with two state-of-the-art interleaving checking approaches: race-based (denoted by *Race*) and atomicity-violation-based (denoted by *Atom*). *Race* is a lock-set–happens-before hybrid race detector [30, 94], originally implemented in the widely-used Valgrind-Helgrind detector [91] and slightly modified by us for better race coverage. *Atom* was implemented by us based on an algorithm described in previous work [99]. It predictively identifies each static memory instruction that can be unserializably interleaved with its preceding access to the same memory location from the same thread (the most common type of atomicity bug [74, 121, 75]). There are other race and atomicity bug detectors, such as happens-before race detectors [93] and training-based atomicity detectors [75]. We did not choose them, because their training requirement or interleaving-sensitive design makes for an apples-to-oranges comparison.

4.2.9 Experimental Results

Overall Results

Overall, as shown in Table 4.11, ConMem can detect 9 out of 10 tested concurrency bugs, showing a good coverage on this set of concurrency memory bugs. In comparison, *Race* and *Atom* detect 4 and 6 out of the 10 bugs, respectively[5].

ConMem shows a good bug-detection capability on these evaluated bugs, because it effectively captures the most common pattern among concurrency bugs with crash-effects. Specifically, three bugs (MySQL-1, MySQL-2, and PBZIP2) are detected by Con-NULL; Apache-1 and PBZIP2-2 are detected by Con-Dangling; Apache-2 and Cherokee are detected by Con-Overflow; Transmission and FFT are detected by Con-UnInit.

[5]We treat these 10 known bugs as the ground truth in our experiment. Admittedly, there could be some unknown bugs lurking and hence some missed false negative problems, which unfortunately has no conceivable way to accurately measure.

Bug-ID	ConMem	Race	Atom
MySQL-1	✓	✓	✓
MySQL-2	✓	×	✓
PBZIP2	✓	✓	✓
Apache-1	✓	×	×
Mozilla	×	×	×
PBZIP2-2	✓	×	×
Apache-2	✓	✓	✓
Cherokee	✓	✓	✓
Transmission	✓	×	×
FFT	✓	×	✓

Table 4.4: Bug-detection results (Key: ✓– bug was detected; × – bug not detected.)

App.	# ShrMem Inst		Races	Atom.	Null	Dangling	UnInit	Overflow	ConMem Total
	Static	Dynamic	#F:#B	#F:#B	#F:#B	#F:#B	#F:#B	#F:#B	#F:#B
Apache	297	76540	14 : 1	157 : 2	4 : 0	6 : 3	0 : 0	0 : 1	10 : 4
MySQL	1086	17379	267 : 2	155 : 2	4 : 2	1 : 0	11 : 0	0 : 0	16 : 2
Transm.	507	978	42 : 0	33 : 0	2 : 0	3 : 0	3 : 1	0 : 0	8 : 1
PBZIP2	93	1744	17 : 6	21 : 4	6 : 6	0 : 2	3 : 0	0 : 0	9 : 8
FFT	205	182532	8 : 0	16 : 5	0 : 0	0 : 0	0 : 4	0 : 0	0 : 4
Cherokee	598	48502	8 : 2	28 : 2	0 : 0	0 : 0	0 : 0	0 : 1	0 : 1
Mozilla	76	18330	13 : 0	48 : 0	0 : 0	0 : 0	2 : 0	0 : 0	2 : 0
False Positive Rates			**3**69:11	**4**58:15	16:8	10:5	19:5	0:2	**4**5:20

Table 4.5: Bug reports and false positives before ConMem-v pruning (Note: 1. the bug report number here is larger than that in Table 4.11, because some bug reports share one root cause. There are 9 distinct root causes of these 20 bug reports. 2. #F: # of false positives; #B: # of bugs; #ShrMem Inst: instructions that access variables truly shared among threads. 3. The special ConMem algorithm to handle custom synchronization is **not** applied here. It will be discussed in connection with Table 4.6)

ConMem still misses one bug in Mozilla. This is a complicated concurrency bug that requires more than one rare timing condition to manifest it. Specifically, a rare atomicity violation among accesses to a shared pointer first causes two threads to mistakenly read from the same heap object, which does not lead to any visible software failure. Later on, another rare timing could cause one thread to delete this heap object while the other thread is still using it, which finally causes the program to crash. This complicated bug is not detected by ConMem, because the buggy interleaving does not *directly* lead to memory errors. It cannot be detected by *Race* or *Atom* either, because it is a multi-variable bug. Note that, Apache-1 bug is also a multi-variable atomicity-violation bug. It can be detected by ConMem, because its manifestation only requires one rare timing between a deletion and a heap-object read access.

Atom and *Race* failed to detect 3 and 4 concurrency bugs, respectively, are detected by ConMem, mainly because these bugs are not caused by data races or simple atomicity violations. For example, Apache-1 is caused by conflicting accesses to multiple variables. Therefore, it is not detected by either *Race* or *Atom*. PBZIP2-2 and Transmission are both caused by order-violation problems and are missed by Atom. In addition, the heuristics used in the Valgrind-Helgrind algorithm to prune false positives also lead to some false negatives in *Race*.

Overall, ConMem has good coverage on the evaluated real-world concurrency bugs that can cause crashes, and is not limited to any specific interleaving pattern. Its algorithms complement existing race and atomicity-violation bug-detection tools.

False-Positive Results

Before automated pruning

Table 4.5 shows the number of false positives (vs. true bugs) of all the tools on the 7 evaluated applications. Every report of *Race* is a pair of static race instructions; every report of *Atom* is a static instruction that can be unserializably interleaved with its preceding access; every report of ConMem is a static instruction that, under certain interleavings, can dereference a NULL-pointer, access a freed memory region, etc. These reports are obtained *before* applying automatic bug exposing. Automatic bug exposing could help prune out most false positives for *Race*, *Atom* [99], and ConMem, at the cost of testing time. Each bug report is judged to be a false positive or a true bug report based on our manual inspection and comparison against all known concurrency bugs in the bug database of the corresponding software. [6] Since some bug reports in Table 4.5 share the same root cause, the total number of true bug reports there is larger than that in Table 4.11.

In general, ConMem's false-positive rate is much lower than *Race* and *Atom* – about one tenth of their false-positive-rates – befitting its effect-oriented approach. ConMem's false-

[6] Code regions that are problematic only under weak memory consistency models are not considered as bugs here, similar to previous work [99, 111]

positive rate (about 2.5 false positives per true bug) is reasonably low considering ConMem's predictive detection capability on concurrency bugs.

All these tools, including *Race* and *Atom*, have done a good job in identifying bug-prone interleavings from the huge interleaving space. As we can see in Table 4.5 (the ShrMem-Inst column), the number of dynamic memory accesses to memory locations that are truly shared among threads ranges from 978 to 182532. The interleaving space size grows exponentially in that number. In contrast, many fewer interleavings are singled out by *Race*, *Atom*, and ConMem.

ConMem has much smaller false-positive rates than *Race* and *Atom*, mainly because of its effect-oriented approach (i.e., taking vertical stripes in the feature space of Figure 4.2a and Figure 4.2b). As discussed in Chapter 4.1, races and unserializable interleavings do not always end up as bugs. Although the algorithms in *Race* and *Atom* already use good heuristics to prune false positives, the false-positive problem is still there.

Table 4.6 provides a further breakdown of the false positives reported by ConMem. As we can see, 43 of the 45 false positives are caused by unidentified custom synchronizations. These 43 bug reports involve infeasible interleavings and can never actually occur. ConMem mistakenly reported these 43 bugs because it did not consider while/if-flags and producer-consumer queue synchronizations in the program. The remaining 2 false positives come from harmless uninitialized reads, as discussed in Chapter 4.2.3.

Note that, according to Table 4.6, *almost all buggy interleavings reported by ConMem are true bugs that cause program to crash, as long as they are feasible.* This is a big accuracy improvement over race detectors and atomicity violation detectors: many races and atomicity violations are intentionally introduced by developers for performance or semantic reasons [88, 12, 99].

Pruning false positives via custom-synchronization analysis

We also evaluated the synchronization-loop analysis discussed in Chapter 4.2.6. As shown in Table 4.6, this analysis can further prune out 16 ConMem false positives, which is more than one third of all ConMem false positives. During this process, no true bug is pruned.

App.	Benign UnInit	# of F.P. caused by custom synchronization		# of F.P. pruned by Chapter 4.2.6 syn-loop analysis
		Producer-Consumer Queue	If/While-flag	
Apache	0	5	5	0
MySQL	0	3	13	5
Transm.	2	0	6	5
PBZIP2	0	3	6	6
FFT	0	0	0	0
Cherokee	0	0	0	0
Mozilla	0	0	2	0
Total	2	11	32	16

Table 4.6: Causes of ConMem false positives

The false-positive rate of ConMem is thus decreased to 1.45 false positives per true bug. The run-time overhead of custom-synchronization identification is similar to that of ConMem bug detection, because it records similar amount of memory-access information as ConMem bug detection.

Automated false positive pruning of ConMem-v

All the 75 bugs reported by ConMem in Table 4.5 are sent to ConMem-v for validation. As a result, ConMem-v automatically prunes out *all* false positives, without introducing any false negatives for the bugs shown in Table 4.11 and Table 4.10.

Specifically, among the 20 true bug reports from ConMem, ConMem-v successfully makes 15 bug reports manifest through its systematic perturbation. Each of these 15 can be reliably (almost deterministically) exposed under ConMem-v, which will help developers diagnose and fix the root causes. There are still 5 bug reports that are actually true bugs. However, the manifestation condition is complicated, requiring artificial delays at multiple places, and is not handled by our current prototype of ConMem-v. Recall that some of these 20 bugs share the same root cause. The 15 bugs successfully exposed by ConMem-v have already covered all the root causes. Therefore, failing to expose the rest 5 bug reports did not cause ConMem-v to miss any root cause.

The ConMem-v validation phase is fast, because of the small number of ConMem bug reports. For example, validating the 17 bug reports of PBZIP2 only takes 20.02 seconds, roughly equal to executing PBZIP2 without any instrumentation 30 times.

Discussion One question the above evaluation does not directly answer is how false positives would change under longer executions with more inputs or more runs of one input.

As discussed in Chapter 4.2.2, the bug-detection ability of ConMem is sensitive to the code/path coverage, like all dynamic bug detectors [111, 75, 91], and is mostly insensitive to small differences in timing (given the same input). Therefore, we expect ConMem to report more true bugs and more false positives when it observes more program runs that touch previously unobserved code/paths. We also expect ConMem's false-positive rate to remain low for most applications and most inputs, because of its effect-oriented design philosophy. For example, if a program performs few NULL-pointer assignments, there will be few bug reports, no matter how long the execution is.

Time and Space Overhead

Table 4.14 shows the run-time overhead of ConMem. Con-NULL also needs trace analysis. Therefore, the off-line analysis time for Con-NULL is also listed. Overall, ConMem's analyses have reasonable run-time overhead: around 16X slow down for memory intensive FFT and 3–29% latency overhead for I/O-intensive server applications. This overhead is comparable to previous concurrency bug-detection tools [75, 130, 111] and is suitable for developers' use.

Con-Overflow's major overhead comes from Valgrind-Helgrind race detector. The overhead of its dependence-analysis ranges from 5% overhead (server applications) to 13X slow down (for FFT).

Currently, Con-NULL, Con-UnInit, Con-Dangling, and Con-Overflow are implemented as separate tools. Since many tasks conducted by them overlap with each other, we expect the overhead of the combined tool to be smaller than running each of them one by one.

In terms of space overhead, Con-NULL is the only tool in ConMem that generates traces. In our experiments, the traces are reasonably small under the bug-triggering inputs, ranging from 50KB to 30 MB. The fact that Con-NULL only analyzes memory accesses to pointer variables greatly mitigates the trace-size problem that is encountered by all trace-based analysis tools. Because the disk sizes keep increasing, we believe that trace size will not be an issue for the usage of Con-NULL.

Bug-ID	Base* Line	Con-NULL		Con-Dangling Run-time	Con-UnInit Run-time
		Run-time	Off-line Analysis		
Apache	0.154s	19%	0.118s	28%	28%
MySQL	0.034s	29%	0.029s	24%	13%
Cherokee	0.072s	7.6%	0.012s	2.7%	6.6%
Mozilla	1.010s	505%	0.030s	185%	196%
PBZIP2	0.662s	116%	0.019s	76%	78%
Transmission	1.362s	82%	0.005s	79%	80%
FFT	0.001s	1113%	0.000s	1285%	1556%

Table 4.7: ConMem Run-time overhead (%) and off-line analysis time (*: BaseLine is to execute the application's test input from the beginning to the end without any instrumentation. Sever applications, like Apache and Cherokee, each serves a set of requests from multiple clients.)

Synchronization Analysis in ConMem

When detecting Con-NULL, Con-UnInit, and Con-Dangling bugs, ConMem conducts synchronization analysis to check whether the timing condition of bug suspects can be satisfied in the future or not. ConMem prunes out those suspects that are well-protected by mutual exclusion or order synchronization. Table 4.8 shows the number of bug suspects that are pruned out by this analysis. As we can see, the pruning is effective. The remaining false positives mainly come from two types of unidentified custom synchronizations. One type is imposed by non-loop control dependency. As illustrated in Figure 4.9(a), the reported Con-Dangling bug *S2, S3* can never happen due to the control dependency imposed by *S1* and *S4*. The second type is imposed by producer-consumer queues. As illustrated in Figure 4.9(b), the assignment in *S1* can never affect *S5*, because *S5* can only access objects from the queue *trxlist* and the update made in *S1* is already overwritten by *S2* when *S3* puts the shared object pointed by *thd* into the queue *trxlist*.

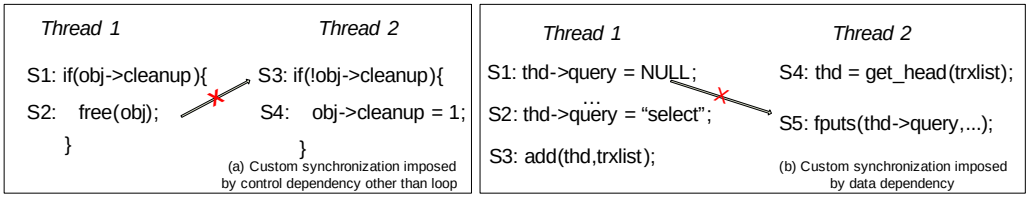


Figure 4.9: Two false positive examples caused by unidentified custom synchronization

Bug-ID	Con-UnInit	Con-Dangling	Con-NULL
Apache	0	0	4
Mozilla	10	0	0
MySQL	62	2	74
PBZIP2	18	0	8
Cherokee	109	21	64
Transmission	25	0	18
FFT	28	0	0

Table 4.8: Bug suspects pruned by synchronization analysis

Testing experience with *Click*

To better evaluate the in-house testing capability of ConMem, we applied ConMem to the latest version of an open-source software system, *Click* [21], for which no concurrency bugs had been previously reported.

Experimental setup *Click* is a popular open-source software router, originally developed by a research group at MIT. *Click* contains around 220K lines of source code. It uses multi-threading experimentally to speed up processing network packets.

The latest version (v-1.8.0) of *Click* contains an input suite designed by *Click* developers to test the basic functionality of *Click*. This suite includes 22 test cases in total. We applied ConMem to all 7 of the test cases that do not require modification of the operating system (i.e., building modules into the kernel).

The testing process is straightforward. We executed each test input once with one ConMem-tool attached to it.[7] No modification was needed to either *Click* or ConMem.

Bug detection results ConMem reports 4–9 buggy interleavings for each test input, as shown in Table 4.9. Since some code regions, such as the start-up code and shut-down code, are covered by most or all test inputs, there are many overlapping bug reports among the 7 test inputs. After manual inspection, we found that the false-positive-vs-true-bug ratio ranges from 3:1 to 2:4 for each test input. Altogether, ConMem reports 6 distinct buggy interleavings that can lead to severe software failures, such as program crashes. These 6 buggy interleavings

[7]Currently, the Con-NULL, Con-Dangling, Con-UnInit, and Con-Ovfl are implemented as four separate Pin tools. Therefore, we executed each test input four times, with each tool attached to one run. We could combine these four into one Pin tool, and each test input would only need to be executed once.

are caused by **2** different root causes in the program. One buggy interleaving reported by Con-Dangling is demonstrated in Figure 4.10. As we can see, the master thread in *Click* maintains a meta-data object, `router_thread`, for each router thread. Because the code does not perform any synchronization, the master thread could delete that object prematurely while it is still being used by the router thread. This bug can lead to a crash in *Click*.

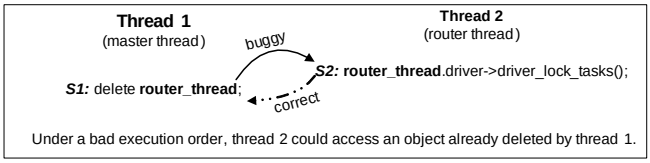


Figure 4.10: A concurrency bug that leads to a dangling pointer and finally a crash (from *Click*-1.8.0)

ConMem has about a 1:1 false-positive-vs-true-bug rate for *Click*, which is consistent with the earlier experiments shown in Table 4.5. The false positives here are mainly caused by a complicated if-condition control-flow synchronization. This custom synchronization forces the dereferences to certain shared pointers to either happen before the pointer deletion or to get by-passed. This type of custom synchronization is not handled by ConMem.

As shown in Table 4.9, we also tried *Races* and *Atom* on these 7 test cases. The results follow a similar trend to that in Table 4.5. *Races* and *Atom* cannot detect the bugs that ConMem detected. For example, the bug depicted in Figure 4.10 is neither a race nor an atomicity violation. Race bugs and atomicity-violation bugs should involve several accesses to the same memory location with at least one write. This is not true for the bug in Figure 4.10 that involves a call to a C++ library function in Thread 1 and some reads in Thread 2. Currently, neither *Races* nor *Atom* instruments the library code. Even if they do, there is a large chance that no write to the conflicting memory location exists, depending on how delete is implemented in the library.

Performance *Click* has two execution modes. The normal execution mode is IO-intensive, where *Click* listens to the network. Under this mode, the overhead of ConMem depends on the network traffic and is usually negligible. The other execution mode ("simulation mode") is CPU- and memory-intensive, where *Click* reads packages from a trace. Under the

App.	Races	Atom.	Con-Null	Con-Dangling	Con-UnInit	Con-Ovfl	ConMem Total
	#FP:#Bug	#FP:#Bug	#FP:#Bug	#FP:#Bug	#FP:#Bug	#FP:#Bug	#FP:#Bug
Test 1	13 : 0	20 : 0	1 : 0	1 : 4	0 : 0	0 : 0	2 : 4
Test 2	18 : 0	22 : 0	3 : 0	1 : 4	1 : 0	0 : 0	5 : 4
Test 3	18 : 0	18 : 0	1 : 0	1 : 4	0 : 0	0 : 0	2 : 4
Test 4	19 : 0	41 : 0	2 : 0	1 : 2	0 : 0	0 : 0	3 : 2
Test 5	10 : 0	17 : 0	1 : 0	2 : 3	0 : 0	0 : 0	3 : 3
Test 6	28 : 0	25 : 0	1 : 0	2 : 1	0 : 0	0 : 0	3 : 1
Test 7	8 : 0	41 : 0	1 : 0	2 : 1	0 : 0	0 : 0	3 : 1

Table 4.9: *Click*'s ConMem testing reports. The false-positive numbers are collected before ConMem-v pruning (Notes: 1. The bugs detected by ConMem have not been reported before. 2. There is overlap among the bugs reported for the 7 inputs.).

memory-intensive mode, each ConMem testing run introduces about a 20-times slow-down. Without ConMem, the original 7 test cases take 0.259 seconds to finish. ConMem testing takes 22.108 seconds in total, including 0.028 seconds for off-line analysis, and 22.08 seconds for Con-Null, Con-Dangling, Con-UnInit, and Con-Overflow testing runs. The trace size of ConMem-NULL is 16K bytes on average for the 7 test cases.

Summary Our experience of applying ConMem to *Click* is summarized as follows:

- ConMem is easy to use, straight out of box. The user needs to provide nothing other than the standard test suite.
- ConMem is effective, it can detect previously unknown concurrency bugs.
- ConMem is accurate, compared to many traditional tools. Its false-positive rate was low enough to allow us to manually inspect every bug report.
- ConMem imposes low-enough overhead for use during in-house testing. For CPU and memory intensive applications, such as *Click* in simulation mode, ConMem imposes about an 80-fold run-time overhead (= 4 tools, each with about 20x slowdown) and requires about 500KB/sec for storing traces. We also see two approaches that can significantly decrease ConMem's overheads in the future: (1) Combining all four ConMem tools into one, because each ConMem tool has only about 20 times overhead on *Click* and there is significant redundancy among the four tools. (2) Saving redundant interleaving testing among inputs that have overlapped code coverage. This is obviously more challenging,

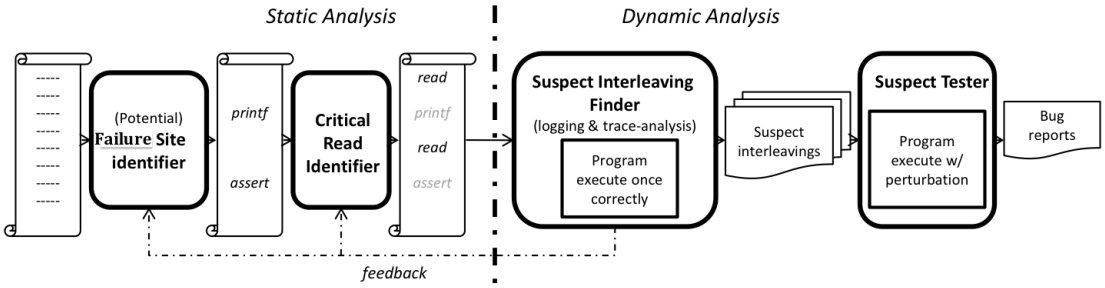


Figure 4.11: An overview of the ConSeq architecture.

but is also promising. As discussed in connection with Table 4.9, there is an overlap among the concurrency bugs revealed by different inputs.

- While we have had a fairly positive experience with applying ConMem to Click, one additional feature can make ConMem easier to use in the future: providing information about why a bug suspect is not exposed by ConMem-v.

4.3 ConSeq

4.3.1 Overview of ConSeq

ConSeq mainly detects concurrency bugs that cause semantic errors. According to our characteristics study in Chapter 3, there are four major types of failures caused by concurrency semantic bugs – assertion failures, error messages, wrong outputs and hangs. ConSeq detects all of them in a unified way. ConSeq is divided to two working stages: sequential stage, which starts from the potential failure sites in the software and conducts the backward search to locate the critical reads in the same thread; and concurrent stage, which identifies a buggy interleaving that can feed critical reads with bad values to fail the program. Specifically, as shown in Figure 4.11, ConSeq uses a combination of static and dynamic analyses. It uses the following modules to create an analyzer that works backwards along error-propagation chains.

Failure-site identifier: this static-analysis component processes a program binary and identifies instructions where certain failures might occur. For example, a call to `__assert_fail` is a potential assertion-violation failure site. Currently, ConSeq identifies potential failure

sites for four types of errors (Chapter 4.3.2). Developers can adjust the bug-detection coverage and performance of ConSeq by specifying specific types of failure sites on which to focus.

Critical-read identifier: this component uses static slicing to find out potential critical reads. Note that static analysis is usually not scalable for multi-threaded C/C++ programs. By leveraging the short-propagation characteristic of concurrency bugs and the staged design of ConSeq, our module is scalable to large C/C++ programs (Chapter 4.3.3).

Suspicious-interleaving finder: this dynamic-analysis module monitors one run of the concurrent program, which is usually a correct run, and analyzes what alternative interleavings could cause a critical read to acquire a different and dangerous value. By leveraging the characteristics of concurrency bugs' root causes, this module is effective for large applications. Via this module, ConSeq generates a bug report, which provides a list of potential critical reads that can potentially read dangerous writes and lead to software failures. Potential critical reads, dangerous writes, and the potential failure sites are represented by their respective program counters in the bug report. Additionally, the stack contents are provided to facilitate programmers' understanding of the bug report.

Suspicious-interleaving tester: this module tries out the detected suspicious interleavings by perturbing the program's re-execution (Chapter 4.3.5). It helps expose concurrency bugs and thereby improves programmers' confidence in their program. Via this module, ConSeq prunes false positives from the bug report, and extends the report of each true bug with how to perturb the execution and manifest the bugs.

Note that the boundaries of ConSeq's static and dynamic analysis are not fixed. Making the bug-detection technique scalable and applicable to large C/C++ applications is a principle in ConSeq's design. ConSeq uses dynamic analysis to refine static-analysis results, and static analysis also takes feedback from run-time information.

Before diving into the technical details of ConSeq, we use the multi-variable concurrency bug shown in Figure 4.1(c) as an example to demonstrate the work flow of ConSeq. When we apply ConSeq to the binary of the Mozilla mail client, ConSeq's *failure-site identifier* identifies 200 assertions. One of them is the instruction 0x4f81d (i.e., the assertion-failure call

site corresponding to S4 in Figure 4.1(c)). Next, ConSeq's *critical-read identifier* statically analyzes the control/data dependences leading to each assertion identified above. In particular, instruction 0x4f7f2 (i.e., the read of `runningUrl` in statement S4 in Figure 4.1(c)) is identified for the assertion site 0x4f81d. The application is then executed. Not surprisingly, no error occurs during the execution. ConSeq analyzes the 31 executed potential critical reads one by one. It identifies an alternative interleaving that might cause instruction 0x4f7f2 to read an assertion-violating value, `NULL`, defined by instruction 0x8062e5 (S1 in Figure 4.1(c)). Finally, ConSeq's *suspicious-interleaving tester* executes the program again and triggers a failure. In terms of users' involvement, ConSeq only requires a user to provide one thing: a test suite. Users are also allowed to provide a list of function names of interest (such as the customized error-message function). ConSeq then will automate the whole bug-finding process described above.

4.3.2 Identify potential failure sites

The failure-site identification module has three goals: (i) to identify potential failure sites automatically, (ii) to identify them before a failure occurs, and (iii) to accomplish (i) and (ii) with good accuracy and coverage. This module provides the starting points for ConSeq's backward concurrency-bug detection strategy and directly affects the false-positive and false-negative rates of ConSeq. To achieve its goals, ConSeq follows two design principles:

(1) Use static analysis instead of dynamic analysis. Errors rarely occur during monitored runs of concurrent programs. Static analysis can go beyond what occurs during a single execution.

(2) Exploit the failure patterns of software bugs. Concurrency bugs, fortunately, have similar failure patterns as sequential bugs, which are well-studied and well-understood.

Identifying explicit failure sites

According to the characteristics study in Chapter 3, failures of non-deadlock concurrency bugs can be covered by four patterns that they share with sequential bugs. ConSeq identifies

each pattern as follows.

Infinite Loop: For non-deadlock bugs, infinite loops in one thread are the main causes of hangs (an example is shown in Figure 4.1(b)). Every back-edge in a loop is a potential site for this type of failure. ConSeq identifies strongly connected components (SCCs) that are potential failure sites for infinite-loop hangs by checking whether any shared-memory read is included in the backward slice of each back-edge in an SCC. To identify nested loops, CodeSurfer/x86 implements *Bourdoncle*'s algorithm [10], which recursively decomposes an SCC into sub-SCCs, etc.

Assertion Violations: Assertion violations (Figure 4.1 (c)) are a major source of program crashes. Fortunately, it is a common practice of developers to place assertions in their code. Moreover, assertions are able to specify certain other types of errors. In C/C++ programs, a call to gcc's `assert` library function is translated to an if statement whose else-branch contains a call to `__assert_fail`. The call sites on `__assert_fail` are considered to be potential failure sites. Some applications use customized assertions, such as `nsDebug::Assertion` in Mozilla. ConSeq also considers those call sites to be potential failure sites.

Incorrect Outputs: Most non-fail-stop software failures occur when the software generates incorrect outputs or totally misses an output. ConSeq considers a call to an output function, such as `printf` and `fprintf`, as a potential incorrect-output failure site. Some applications have special output functions, such as MySQL's `BinLog::Write`. ConSeq allows developers to specify application-specific output functions in a text file. ConSeq reads the text file and identifies call sites on the specified functions.

Error Messages (consistency-check sites): Consistency checks have an interesting role in concurrency bugs. They are usually **not** designed for catching synchronization bugs, and simply reflect a developer's wish to enforce some important correctness property. Luckily, however, for many complicated concurrency bugs, there are warning signs long before the ultimate failure arises. As a result, the error-propagation distance is greatly shortened and backward bug-detection becomes much easier due to error messages. *Writing such consistency checks has been a common practice of programmers for a long time [68], and the presence of*

consistency checks can greatly help the approach taken by ConSeq.

ConSeq identifies calls to functions that print error messages as potential failure sites. These include both library functions, such as `fprintf(stderr,...)`, and application-specific routines, such as the `NS_WARNING` in Mozilla and `tr_err` in Transmission (a BitTorrent client). ConSeq allows developers to specify these error-reporting functions in a file. ConSeq reads this file and identifies call sites on all these functions. In our experience, most applications only have a few (usually just one or two) error-reporting routines. Therefore, we believe it will not be a big burden for developers to write down these functions.

In the case of assertion failures and error messages, a condition that indicates whether the value acquired at a given site is correct or not is obtained as a by-product. This condition is used to improve the accuracy of ConSeq's bug-detection capabilities (Chapter 4.3.4).

Inferring implicit error sites

As discussed above, consistency checks added by developers are very helpful in ConSeq's method for bug detection. What if developers did not provide any consistency checks?

Interestingly, research on sequential programs has faced this problem before, and some solutions have been proposed. For instance, Daikon [35] is a tool that infers likely program invariants based on evidence provided by (correct) training runs. Daikon's most advanced features allow for inference among derived variables, as well as set relations and arithmetic relations between array elements. In this respect, Daikon can automatically provide information that is similar to the consistency checks manually added by developers. We can treat those places where Daikon identifies program invariants to be potential error sites.

Specifically, we first apply Daikon to the target software. Daikon's frontend logs run-time variable values at program points selected by Daikon. Daikon's backend processes the log and outputs a list of {program-point, invariant} pairs.

ConSeq checks every global read instruction I that reads global variable v. If Daikon has identified an invariant involving v right before I, ConSeq identifies I as a potential invariant-violation site.

One implementation challenge we encountered is that the default frontend, kvasir-dtrace, of Daikon's academic version only collects information at function entries and exits. As a result, we cannot obtain invariants at the granularity of individual instructions. With the help of the Daikon developers, we tried two ways to get around this problem. For small applications in our experiments, we manually inserted dummy functions before every global-variable read. For large applications in our experiments, we replaced Daikon's default front-end with our own PIN tool. This PIN tool collects run-time information before every global-variable read, and outputs this information in the input format used by kvasir-dtrace. By this means, the Daikon backend can process the Pin tool's output and generate invariants.

For large applications, one potential concern is that Daikon could identify a huge number of invariants, which could impose a large burden on ConSeq's critical-read identifier, suspicious-interleaving finder, and suspicious-interleaving tester. Fortunately, Daikon provides ranking schemes [36] to identify important invariants. ConSeq leverages the ranking mechanism to focus on the most important invariants.

In summary, ConSeq currently focuses on five types of potential failure/error sites. Except for the potential error sites inferred by Daikon, all sites are identified by statically analyzing the program.

4.3.3 Identifying potential critical reads

The goal of the critical-read identification module is to identify potential *critical-read* instructions that are likely to impact potential failure sites through data/control dependences. It uses static slicing to approximate (in reverse) the second propagation phase of a concurrency bug, as shown in Figure 3.3. There are two major design principles for this module:

1. Use *static* analysis rather than dynamic analysis to identify which instructions *may* affect a failure site. ConSeq is different from failure-diagnosis tools. It aims to expose concurrency bugs **without any knowledge of how they may arise or even if they exist**, so its analysis cannot be limited to any specific monitored run. Specifically, ConSeq uses *static slicing* for this purpose.

2. Only report instructions with short propagation distances as potential critical reads. Computing the complete program slice, e.g., all the way back to an input, is complicated and also unnecessary for ConSeq. ConSeq leverages the short-propagation characteristic of concurrency bugs (Chapter 3) to improve bug-detection efficiency and accuracy.

General issues

We had to make several design decisions that are general to all types of failure sites:

Must critical-read instructions access shared memory? Instructions that read thread-local variables could be of interest for sequential bug detection, but not for concurrency bug detection, because their values cannot be directly influenced by interleavings. To get rid of these instructions, ConSeq first uses static analysis to filter out as many stack accesses as possible. ConSeq's run-time monitoring will proceed to prune out the rest of the stack accesses. Of course, it is possible for threads to share values using the stack, although it is rare in practice. Escape analysis would be able to identify these special stack accesses, and make ConSeq more accurate. We leave this as future work.

Shall we consider inter-thread control/data dependences? Multi-thread static slicing is much more difficult than single-thread slicing. Fortunately, because ConSeq's design separates the propagation steps in a concurrency bug into inter-thread and intra-thread phases, here only single-thread dependence analysis is needed to identify potential critical reads. All analyses involving multi-thread interleavings will be conducted in the suspicious-interleaving finder (Chapter 4.3.4).

How to set the propagation-distance threshold? In accordance with the short-propagation heuristic, ConSeq only reports read instructions whose return values can affect the failure sites through a short sequence of data/control dependences. Our static-slicing tool provides the slice, together with the value of the shortest distance to the starting point of the slice, for each instruction of the slice. An example is shown in Figure 4.12. ConSeq provides a tunable threshold *MaxDistance* for users to control the balance between false negatives and false positives. By default, ConSeq uses 4 as *MaxDistance*. A detailed evaluation is presented in Chapter 4.3.7. We will explore other metrics for propagation distance in the future.

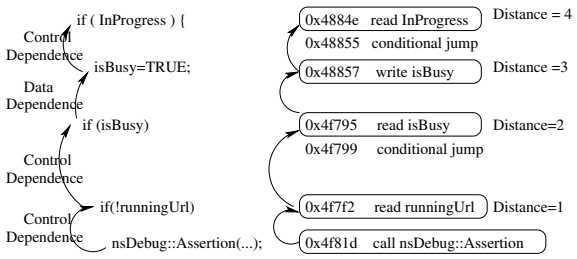


Figure 4.12: Static slicing of machine code (right) and the distance calculation.

How to reuse the analysis results across inputs? Because ConSeq uses static instead of dynamic analyses, the results from this module, as well as those from the failure-site identifier, can be reused for different inputs. Our current static slicer analyzes one object file at a time. To speed up the analysis when there are only a few inputs, we first process those object files that these inputs would touch.

Customization for different types of failure sites

ConSeq customizes the analysis for each type of failure site:

- Each consistency-check failure site is a 'call' instruction that calls a standard or custom error-reporting routine. ConSeq directly applies slicing for that instruction. For a 'call' instruction, the first step backward is always through a control dependence, followed by a sequence of control and data dependences.
- Each assertion-failure site is a 'call' to the `__assert_fail` library routine. We handle it in the same way as a consistency-check failure site.
- Each invariant-violation failure site is an instruction that reads heap or global variables. No customization is needed. ConSeq directly applies control-and-data slicing for each of these instructions.
- Each incorrect-output site is a '`call`' instruction to output functions. Before applying static slicing, we first use a simple static analysis to identify all instructions that push argument values onto the stack (sometimes via `push` and sometimes via `mov`).[8] We add

[8]Figuring out the parameters to a call in the binary is easy for the CDECL calling convention. If we need to process GCC ___fastcalls, we will need to go back to analyzing the source code. The current implementation of ConSeq does not handle ___fastcalls.

those instructions into the slice (at distance 1) and apply slicing to these instructions and the original `call`.

- Each infinite-loop site involves a jump instruction that conditionally jumps out of a loop. Among instructions that are on the slice, we only keep those that are repeatedly executed inside the loop body, because only those instructions could lead to the loop executing repeatedly.

Static slicing details

Program slicing is an operation that identifies semantically meaningful decompositions of programs, where the decompositions may consist of elements that are not textually contiguous [127]. A *backward slice* of a program with respect to a set of program elements S consists of all program elements that might affect (either directly or transitively) the values of the variables used at members of S. Slicing is typically carried out using *program dependence graphs* [53].

CodeSurfer/x86. ConSeq uses backward slicing to identify shared memory reads that might impact each potential failure site. To obtain the backward slice for each potential failure site, it uses CodeSurfer/x86 [5], which is a static-analysis framework for analyzing the properties of x86 executables. Various analysis techniques are incorporated in CodeSurfer/x86, including ones to recover a *sound approximation* to an executable's variables and dynamically allocated memory objects [6]. CodeSurfer/x86 tracks the flow of values through these objects, which allows it to provide information about control/data dependences transmitted via memory loads and stores.

Side-Stepping Scalability Problems. To avoid the possible scalability problems that can occur with CodeSurfer/x86 due to the size of the applications used in evaluating ConSeq, we set the starting point of each analysis in CodeSurfer/x86 to the entry point of the function to which a given potential failure site belongs, instead of the main entry point of the program. By doing so, CodeSurfer/x86 only needs to analyze the functions of interest and their transitive calls rather than the whole executable. Thus the static analyses time grows roughly linearly in the number of functions that contain failure sites. This makes ConSeq much more scalable,

as will be illustrated in Chapter 4.3.7.

This approach is applicable in ConSeq because—based on the observation that the error-propagation distance is usually short, as discussed in Chapter 3—ConSeq only requires a *short* backward slice that can be covered in one procedure. The backward-slicing and other analysis operations in CodeSurfer/x86 are, however, still context-sensitive and *interprocedural* [53]. Moreover, to obtain better precision from slices, each of the analyses used by CodeSurfer/x86 is also performed interprocedurally: calls to a sub-procedure are analyzed with the (abstract) arguments that arise at the call-site; calls are not treated as setting all the program elements to $\top$.

Analysis Accuracy. To obtain static-analysis results that over-approximate what can occur in any execution run, all the program elements (memory, registers, and flags) in the initial state with which each analysis starts are initialized to $\top$, which represents any value. Such an approximation makes sure that no critical read will be missed by ConSeq at run time. Of course, some instructions could be mistakenly included in the backward slice and be wrongly treated as critical reads. Fortunately, our short-propagation-distance heuristic minimizes the negative impact of over-approximation. In practice, we seldom observe the inaccuracy caused by this over-approximation.

Finally, the CodeSurfer/x86 framework has information about every direct calls' call-sites. Therefore, if needed, it can also support backward slicing that starts at the entry of a procedure and backs up into the callers.

4.3.4 Identifying suspicious interleavings

The module for finding suspicious interleavings focuses on the first phase of concurrency-bug propagation. ConSeq monitors a program's (correct) execution, collects a trace using binary instrumentation, and analyzes the trace to decide whether a different interleaving could change the dynamic control/data dependence graph and generate a potentially incorrect value at a critical read.

Because it is impractical to check all potential interleavings and all potential dynamic

control/data dependence graphs [56], ConSeq leverages the short-propagation characteristic (Chapter 3) and the widely used shallow-depth heuristic (i.e., the manifestation of most concurrency bugs involves only two or three shared-memory accesses) [84, 85, 11, 99]. It examines writes that are *one data-dependence step backward* from each potential critical read $\mathtt{r}$, and looks for suspicious interleavings that could make $\mathtt{r}$ obtain a potentially incorrect value written by a write access that is different from the one that occurred in the monitored run. The algorithm in ConSeq is neither sound nor complete. Rather, ConSeq tries to balance generality, simplicity, and accuracy.

The core analysis

We formalize the key question this portion of ConSeq has to answer as follows: in a concurrent program's execution trace T, a read instruction $\mathtt{r}$ gets a value defined by a write access w; we ask whether $\mathtt{r}$ can read a value defined by a different write w' in an alternative interleaving.

To realize a w'–$\mathtt{r}$ data-dependence, three conditions have to be satisfied. First, w' and $\mathtt{r}$ need to access the same memory location $\mathtt{m}$. This condition is fairly easy to check, as long as ConSeq records the addresses of the locations touched by memory accesses.

Second, w' needs to execute before $\mathtt{r}$. This condition can be prohibited by barrier-style synchronizations. Therefore, ConSeq monitors `pthread_create/join`, `pipe`, and `barrier` at run-time to maintain vector-clock time-stamps for each thread and hence each access. A w'–$\mathtt{r}$ dependence is infeasible if $\mathtt{r}$ has a smaller time-stamp than w'. ConSeq computes the vector-clock time-stamps in a similar way as traditional happens-before race detectors [93]. ConSeq does not update time-stamps according to `lock`/`unlock` operations, because these operations do not provide any execution-order guarantees.

Third, the value written by w' to $\mathtt{m}$ is not overwritten before it reaches $\mathtt{r}$. There are three situations in which an overwrite always happens, as demonstrated in Figure 4.13. The first is due to intra-thread program logic, when there is another write w to $\mathtt{m}$ between w' and $\mathtt{r}$ in the same thread as shown in Figure 4.13(a). The second is due to barrier-style synchronization, as shown in Figure 4.13(b). That is, synchronization operations, such as

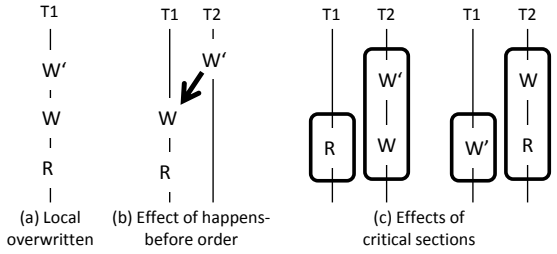


Figure 4.13: A value written by W' may never reach R

`barrier` or `pthread_create/join`, force w' to always execute before another write w and w to always execute before r. The third is due to mutual exclusion, as shown in Figure 4.13(c). When w' is followed by w in a critical section from which r excluded, the result of w' can never reach r. The situation is similar when r is preceded by w in a critical section from which w' is excluded. As long as the trace includes sufficient information about lock operations, ConSeq can analyze all of these situations.

Pseudo-code for the method described above is given as Algorithm 1.

The complete algorithm and extensions

ConSeq uses binary instrumentation to monitor three types of operations at run-time: critical-read instructions, instructions that write global and heap variables, and synchronization operations. For each memory-access instruction, ConSeq records the program counter, the address of the accessed memory, and the value of the accessed memory location before the read or after the write. For each lock operation (`pthread_mutex_(un)lock`), ConSeq records the address of the lock variable. For each barrier-style synchronization (`pthread_create/join`, `pipe`, `barrier`,etc.), ConSeq updates the vector time-stamps of every thread. ConSeq uses one trace file for each thread to avoid slow global synchronization. Given these pieces of information, ConSeq can easily analyze the trace and find out all feasible w'–r dependences.

ConSeq also extends the basic algorithm in three ways.

First, ConSeq records the values read by r and written by w' during the correct run, denoted by v' and v, respectively. If the two values are the same, ConSeq does not report a

Algorithm 1 ConSeq identify suspicious interleavings

Require: write access w
Require: write access w'
Require: read access r
Require: w,w',r access the same shared memory address
Ensure: return true if r can read a value from w', false if not

```
1: /*
   Time-stamp comparison is based on the happens-before relationship and vector-clock
   time-stamps
   */
2: if r.time-stamp < w'.time-stamp then
3:    /*r happens before w' */
4:    return false
5: end if
6: if  w'.time-stamp < w.time-stamp < r.time-stamp then
7:    /*w' is overwritten by w */
8:    return false
9: end if
10: if w is executed before r in a critical section CS1, w' is in critical section CS2, CS1 and
    CS2 are from different threads, and CS1 is mutually exclusive from CS2 then
11:   return false
12: end if
13: if w' is executed before w in a critical section CS1, r is in critical section CS2, CS1 and
    CS2 are from different threads, and CS1 is mutually exclusive from CS2 then
14:   /*w' is overwritten by w */
15:   return false
16: end if
17: /*Report feasible in all the other cases */
18: return  true
```

suspicious interleaving. To further prune false positives, ConSeq also evaluates v' against the assertion/error-condition before reporting a suspicious interleaving, using a symbolic-execution module inside ConSeq.

Second, the basic algorithm cannot be directly applied for detecting infinite loops. Suppose that r is a potential critical read that is associated with a potential infinite-loop site. During the monitored run, ConSeq records the write w and its value v that are read by the last dynamic instance of r right before the loop terminates. Now suppose that the basic algorithm identifies an alternative interleaving in which this specific instance of r can receive a different value from an alternative write w'. This condition is insufficient to conclude that this interleaving

is suspicious. If w is executed after w', another instance of $\mathtt{r}$ in a later iteration of the loop can still receive v from w and terminate the loop. Therefore, for each alternative write w' identified by the basic algorithm, ConSeq further compares the happens-before time-stamps between w and w'. An infinite-loop suspect is reported when w' is strictly ordered after w and when w' is concurrent with w.

Third, interleavings could make a critical read $\mathtt{r}$ execute too early and receive an uninitialized value. ConSeq also reports these cases as suspicious interleavings.

Discussion. There are several sources of inaccuracy in our analysis that can cause false positives and negatives. One is that the value written by a write w' might vary in different runs. Another is that interleavings could change the control flow and cause inaccuracy of our analysis. Finally, ad-hoc synchronization has been a problem for almost all predictive concurrency-bug-detection tools. We leverage our static analysis component, which identifies loops, back-edge jumps, and backward slices of back-edge jumps, to identify one type of common ad-hoc synchronization (one thread spins on a while-flag to wait for another thread). The identification algorithm is the same as the one presented in Chapter 4.2.6. After identifying this type of ad-hoc synchronization, ConSeq treats occurrences as traditional barrier-style synchronizations.

4.3.5 Bug Exposing and Validation

The input to ConSeq-tester, the module for testing suspicious interleavings, is a list of data dependences, represented as write/read pairs ($w_{\mathtt{bad}}$–$\mathtt{r}$). The goal is to exercise suspicious interleavings that can realize these suspicious data dependences, so that we can either reliably trigger the bugs or prune them as false positives.

To achieve this goal, ConSeq uses a testing technique that has been used in several previous bug-detection tools [113, 99]. Specifically, ConSeq instruments the program binary and inserts conditional delays with time-outs before every $\mathtt{r}$ and $w_{\mathtt{bad}}$ instructions. ConSeq then re-executes the program with the original input. Because ConSeq is used during in-house testing, the input is available. At run time, the instrumented code either suspends for a while

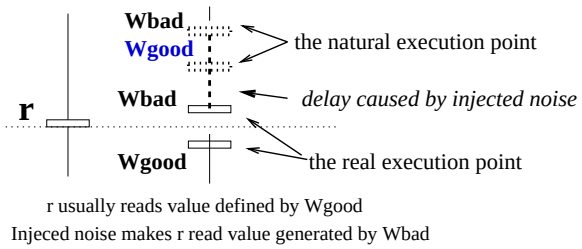


Figure 4.14: Exercising a suspicious interleaving.

the thread that is going to execute w_{bad}, to wait for the arrival of r in another thread, or suspends for a while the thread that is going to execute r, to wait for the arrival of w_{bad}. When both w_{bad} and r are ready to execute, the instrumented code will force the program to execute w_{bad} immediately followed by r. Therefore, the probability that those w_{bad}–r dependences occur is significantly improved. An example of how ConSeq-tester exercises a suspicious interleaving is shown in Figure 4.14.

We have encountered two interesting issues in ConSeq.

First, w_{bad} and r might be from the same thread. The basic scheme shown in Figure 4.14 does not work for this case, because ConSeq will not see r coming when it blocks a w_{bad} operation that is from the same thread as r. ConSeq's suspicious-interleaving identification module marks these cases during trace analysis. During testing, instead of blocking w_{bad}, ConSeq will let it proceed and block any following writes that touch the same memory location that w_{bad} accesses, until r is executed. Second, sometimes w_{bad} and r are protected by the same lock. In those cases, ConSeq inserts a delay before the thread enters the corresponding critical section.

Like many previous concurrency-bug validation tools [89, 99], ConSeq can significantly increase the probability that a concurrency bug manifests, but it cannot provide a 100% guarantee to provoke every bug. In Chapter 4.3.7, however, we will see that ConSeq performs well in practice.

4.3.6 Experimental Methodology

ConSeq's dynamic modules are implemented using the PIN [80] binary-instrumentation

Bug-ID	Symptoms	Application	LOC
Aget1*	Wrong output	Aget-0.4.1	1.1K
FFT	Wrong output	FFT	1.2K
MySQL1	Miss log	MySQL-4.0.12	681K
Moz1	Assertion	Mozilla-1.7	1.2M
MySQL2	Assertion	MySQL-4.0.16	654K
Trans	Assertion	Transmission	95K
Moz2	Error message	Mozilla JS-1.5	87K
Moz3	Error message	Mozilla	N/A
MySQL3	Error message	MySQL-5.0.16	1.6M
MySQL4	infinite-loop	MySQL-5.0.41	1.6M
OO	infinite-loop	OpenOffice	N/A
Cherokee-0.99.48*, web server			96K
Click-1.8.0*, modular router			290K

Table 4.10: Applications and Bugs (Mozilla-JS is the Mozilla Javascript Engine; Cherokee-0.99.48 and Click-1.8.0 are both the latest versions and previously had no known buggy inputs; Moz3 and OO are extracted from old versions of Mozilla and OpenOffice that can no longer compile. *:ConSeq detected new bugs in Aget, Cherokee, and Click.)

framework. The experiments are carried out on an 8-core Intel Xeon machine running Linux version 2.6.18.

We evaluated ConSeq on 8 widely used C/C++ applications. This includes two server applications (the MySQL database and the Cherokee web server), two client applications (Transmission BitTorrent client and Mozilla), two desktop applications (Aget file downloader and OpenOffice), one router (Click [21]), and one scientific application kernel (FFT [128]).

Input design is usually out of the scope of dynamic bug detection [75, 111] and interleaving testing [85, 99, 113], and ConSeq is no different. The intended usage scenario is that ConSeq will be applied to a test suite during in-house testing to expose hidden interleaving errors from (apparently) non-buggy runs on inputs provided by developers or testers. Our experiments were designed to provide insight on the following two questions:

(1) Can ConSeq handle a wide range of types of concurrency bugs? To address this question, and to evaluate ConSeq's bug-detection capability in comparison with traditional bug-detection tools, we used a large set of concurrency bugs that cover different failure symptoms from different applications. In particular, we took 11 concurrency bugs—which

cover assertion failures, hangs, wrong outputs, and error-message problems—from the change logs and bug databases of 6 applications (the first 11 lines of Table 2). In these experiments, to drive ConSeq's bug-detection process we used inputs that were known to have the *potential* of triggering the bug. Our experiments did not leverage any information about the bugs, other than the known inputs. In fact, **none** of the bugs *ever* manifested themselves during the runs that ConSeq performs to generate execution traces for subsequent bug-detection analysis. This methodology is consistent with that used in many previous studies [99, 113]. We will see that ConSeq was able to handle a wide range of types of concurrency bugs (detecting 10 of the 11 bugs).

(2) Can ConSeq find new bugs in the setting of in-house testing (i.e., bugs are not previously known, and inputs are supplied by knowledgeable users)? To mimic the setting of in-house testing, we applied ConSeq to the latest versions of the Cherokee web server [18] and the Click [21] modular router, using test inputs provided by their developers. We were not aware of any concurrency bugs in these two programs. We will see that ConSeq found concurrency bugs in them. Note that these experiments were not started until ConSeq's design and implementation were completely finished. The ability of ConSeq to detect such unknown concurrency bugs also demonstrates the effectiveness of heuristics like the short-propagation heuristic.

Our evaluation of false positives and performance overhead completely executes each input (or set of client requests) from the beginning to the end. The reported performance numbers are the average across 5 runs. The reported false-positive numbers are stable across the multiple runs that we tried. By default, we set *MaxDistance* to 4. We also evaluate false-positive and false-negative results under different *MaxDistance* settings. ConSeq-Daikon demands special setup, and is discussed separately in Chapter 4.3.7.

For comparison, we also evaluated two state-of-the-art cause-oriented approaches to detecting concurrency bugs under the same setting. *Race* is a lock-set–happens-before hybrid race detector, commonly known as Helgrind, implemented as part of the open-source bug-detection tool Valgrind [91]. *Atom* [99] detects the most common type of atomicity bug (two

accesses in one thread unserializably interleaved by another thread [74, 121, 75]). Similar to ConSeq, these two detectors aim to detect bugs from correct runs.

4.3.7 Experimental Results

Overall bug-detection results

Table 4.11 shows the overall bug-detection results. As we can see, ConSeq has good coverage in bug detection. It detected 10 out of the 11 bugs. *Race* and *Atom* only correctly detected 3 and 4 bugs, respectively.

Aside from the bug in Aget listed in the Table 4.10, ConSeq detected two new bugs in Aget that have never been reported before (one by tracing back from a `printf` call site and one by finding a violation of a (candidate) invariant identified by ConSeq-Daikon). In MySQL-5.0.16, ConSeq detected an infinite-loop concurrency bug initially reported in MySQL-5.0.41, which shows that the bug actually existed in the older version and can be triggered using a different input. Our analysis of Cherokee-0.99.48 and Click-1.8.0 used the basic inputs provided in the applications' test suites, and ConSeq discovered bugs in them as we will see in Chapter 4.3.7.

	ConSeq	*Race*	*Atom*
Bug-ID	Detected	Detected	Detected
Aget1			
FFT	✓		✓
MySQL1	✓		✓
Moz1	✓		
MySQL2	✓		
Trans	✓		
Moz2	✓	✓	✓
Moz3	✓	✓	
MySQL3	✓	✓	✓
MySQL4	✓		
OO	✓		

Table 4.11: Bug detection results (✓: detected; Blank: not).

Atom targets single-variable atomicity violations that involve three accesses, and cannot detect concurrency bugs caused by other interleaving patterns, such as the bugs in Moz-1 (a multi-variable atomicity violation), MySQL-2 and OO (anti-atomicity violations where

the software behaves correctly only when a certain code region in a thread is not atomic), Moz-3 (an atomicity violation involving more than three accesses), MySQL-4 and Trans (order violation), etc. *Race* suffers from a similar source of false negatives as *Atom*: the root cause of many of these bugs has nothing to do with locks, and many buggy code fragments did use locks correctly (e.g., OO, MySQL-4). In addition, *Race* uses some heuristics to lower the false-positive rate (e.g., not reporting a race when earlier races are already reported on that variable), which leads to some false negatives.

ConSeq's effect-oriented approach means that its bug-detection capabilities are not limited to any specific interleaving pattern, and thus ConSeq can detect bugs that *Race* and *Atom* cannot. Chapter 4.3.1 has already discussed how ConSeq detects Moz-1, the **multi-variable bug** illustrated in Figure 4.1(c). Figure 4.15 shows an **anti-atomicity** example (MySQL2). S3 from the slave thread wants to use the value of `pos` defined by the master thread (S2) to read the log. Unfortunately, S3 could non-deterministically execute before S2 and mistakenly read a value defined by its own thread, leading to the MySQL failure. The bug is obviously not a race, because all accesses are well-protected. The bug is also not an atomicity-violation bug, because MySQL executes *correctly* when the atomicity between S1 and S3 is *violated*! Furthermore, it is not a simple order-violation bug, because there are many dynamic instances of S1, S2, and S3. No order between S2–S1 or S2–S3 can guarantee failure. With a cause-oriented approach [76, 116], more sophisticated interleaving patterns and a large number of training runs are needed to detect this bug. In contrast, with ConSeq's effect-oriented approach, this bug presents no special challenges. MySQL developers already put a sanity check before each log read: `assert(pos_in_file + pos == req_pos)`. Analyzing backwards from that check, ConSeq easily discovers the bug.

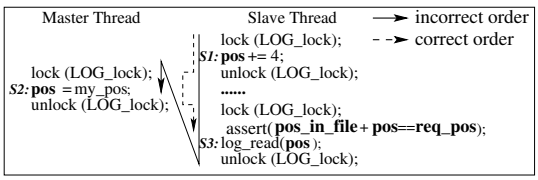


Figure 4.15: An example showing that ConSeq can detect a non-race, non-atomicity-violation bug (simplified for purposes of illustration).

New bugs detected by ConSeq: Apart from detecting the bugs described above, ConSeq also detected two more concurrency bugs that we were unaware of in Aget, and a known infinite-loop bug in a different version of MySQL than originally reported.

More interestingly, ConSeq found an output non-determinism in Cherokee and two bugs in Click. For example, one bug in Click can cause locks to be destroyed when they are still in use. ConSeq can detect this bug using any input provided in Click's test suite. Specifically, after a correct run of Click, ConSeq reported that an error message `"Spinlock:: Spinlock(): assertion '_depth == 0' failed"` could be triggered under a different interleaving. The report accurately points us to the bug. In terms of the root cause, this bug is a non-datarace order-violation bug.

False negatives: There is one bug in Aget that evaded detection by all three tools. The bug is an atomicity violation that involves 11 threads and 21 shared variables. ConSeq failed to detect it because the bug involves a long propagation distance. However, with the support of Daikon, ConSeq can successfully detect it, see (Chapter 4.3.7).

ConSeq and traditional tools look at concurrency bugs from different perspectives and can miss bugs in different ways. *Race* and *Atom* have false negatives in the examples discussed above because they cannot cover certain interleaving patterns. ConSeq will inevitably miss some bugs due to missing certain types of failure sites or due to error-propagation distances that exceed the threshold used in the short-propagation heuristic. Of course, the coverage of ConSeq could be further improved in the future by adding more failure templates or tuning the *MaxDistance* threshold. It could also be helped by the use of additional invariant-inference techniques and by developers who are comfortable with adding consistency checks. In summary, ConSeq can well complement existing bug-detection approaches.

False positives

Before suspicious-interleaving testing. False positives have always been a problem in concurrency bug detection, especially for predictive bug detectors that need to analyze a huge number of potential interleavings, such as ConSeq, *Race*, and *Atom*.

	ConSeq				*Race*	*Atom*	Base
Bug-ID	OUT	ASS	ERR	LOOP			
Aget1	0	0	0	0	2	4	0
FFT	0	0	0	0	8	16	20
MySQL1	0	0	0	0	127	51	77
Moz1	0	0	0	0	26	n/a	OM
MySQL2	0	4	2	5	163	402	OM
Trans	0	2	0	0	42	33	136
Moz2	1	4	0	0	20	279	244
Moz3	0	0	0	0	0	0	0
MySQL3	0	4	3	7	714	1026	OM
MySQL4	0	1	3	0	180	552	197
OO	0	0	0	0	0	0	0
Cherokee	0	0	1	3	40	296	OM
Click	0	1	0	0	13	20	37

Table 4.12: False positives in bug detection. (OM means analysis runs out of memory before finish.)

As shown in Table 4.12, ConSeq has much better accuracy than *Race* and *Atom*,[9] exhibiting only about one-tenth the false-positive rate of the latter two. For 11 out of the 13 cases, ConSeq only has 0–5 false positives. Compared to traditional predictive bug-detection tools, ConSeq can save significant testing resources and manual effort by developers.

The main reason that ConSeq reports fewer false positives than traditional approaches is that its effect-oriented approach has made it much more focused. To validate this, we also measured the false-positive rate for ConSeq with identification of critical reads turned off. The numbers are roughly comparable to the ones for *Atom*, as shown by the last column ('Base') in Table 4.12. Actually, if not guided by potential failure sites and critical reads, the analysis runs out of memory before finishing for several MySQL and Mozilla workloads, because the interleaving space is huge. This is exactly why ConSeq identifies critical read instructions based on potential failure patterns.

The false positives of ConSeq are of two types: (1) Unidentified customized synchronization operations make a suspicious interleaving infeasible. This reason is responsible for all but 3 cases. (2) A different but still correct value is read at a critical read `r`. This is responsible for

[9]We conducted manual validation for randomly sampled *Race* and *Atom* bug reports.

3 false positives.

After suspicious-interleaving testing. ConSeq-tester prunes out all the false positives discussed above with one false negative. Specifically, ConSeq-tester successfully makes the 9 bugs detected by ConSeq in Table 4.11 manifest themselves. Unfortunately, of the two new bugs detected by ConSeq in Click, one cannot be automatically exposed by ConSeq-tester. Its manifestation requires a complicated sequence of branches to be followed, involving multiple branch points in Click.

For those bugs that can be automatically exposed by ConSeq-tester, their manifestation can all be reliably repeated by inserting delays at the same places recorded by ConSeq-tester, which can help programmers perform further diagnoses.

Detailed bug-detection results

Sensitivity of MaxDistance: In ConSeq, the *MaxDistance* threshold affects how many read instructions are considered potentially 'critical'. We measured the false positives and false negatives of ConSeq under different *MaxDistance* settings. The total number of false positives gradually increases, as does the bug-detection capability. Adding all 13 bug-detection runs together, ConSeq reports 25, 33, 37, 41, 61 false positives in total, with *MaxDistance* set to 1, 2, 3, 4, and 5, respectively. ConSeq detects 6, 8, 9, 10, and 10 out of the tested 11 bugs, with *MaxDistance* set to 1, 2, 3, 4, and 5, respectively. These numbers demonstrate the usefulness of static slicing in ConSeq: if ConSeq only looks at shared variables right at the failure site, almost half of the bugs will be missed.

Potential failure sites and potential critical reads. Table 4.13 shows the size of the object files processed by our static-slicing tool and the number of potential failure sites and potential critical reads identified. As we can see, our static-analysis component can handle large applications whose object files are tens of mega-bytes.

Performance results

ConSeq is designed for in-house testing and goes through three phases. The first phase of

	Object File Size	# of Failure Sites	# of Potential Critical Reads
Aget1	56K	86	49
FFT	33K	52	111
MySQL1	15M	2137	8562
Moz1	4.9M	142	397
MySQL2	14M	487	1369
Trans	1.2M	158	232
Moz2	1.4M	856	929
Moz3	5.7K	1	1
MySQL3	25M	1349	2020
MySQL4	27M	867	2341
OO	13K	26	75
Cherokee	2.6M	424	1261
Click	24M	386	1365

Table 4.13: Potential failure sites and potential critical reads.

ConSeq uses static analysis to identify potential failure sites and potential critical reads. This step is *not* performance-critical, because it is conducted only *once* for each piece of code. It can be re-used across different testing runs and different inputs. Even after a code modification, ConSeq only needs to re-analyze those object files that have been changed. In our experiments, the static analysis is scalable. It can finish within a couple of hours for most applications. Processing MySQL3 takes the longest time — 18392 seconds or about 5 hours.

The second phase of ConSeq takes a test input, runs it once to collect a trace, and analyzes the trace to report suspicious interleavings. At the end of this step, concurrency bugs are reported as shown in Table 4.11 and 4.12. This step would be repeated many times during in-house testing and is the most important for ConSeq's performance. Table 4.14 shows the results for this phase. ConSeq's run-time overheads for the four types of failure patterns are similar, so for each application in the table we only present the worst-case performance and largest trace size among these four cases. Adding the run-time and the time for off-line analysis of the trace together, ConSeq introduces execution overhead of 1.26X — 38.5X for each test input, which is suitable for in-house use. ConSeq's trace sizes are reasonably small. The biggest trace size is about 115MB in our experiments, as shown in Table 4.14. The peak memory consumption of ConSeq at run-time is less than 100 MB for all applications.

At the end, ConSeq also has an optional step of suspicious-interleaving testing. This step imposes less than 10–55% execution overhead for validating each bug report. Given the small false-positive rate of ConSeq, this step does not take a long time and can be omitted by developers. In our experiments, MySQL3 had the largest accumulated overhead at this step, because it has the largest number of false positives. The baseline run of MySQL3 finishes in 0.46 seconds. In total, ConSeq took 10.6 seconds for the validation step for all 15 reported suspicious interleavings and pruned out 14 false positives.

	Base Line	Run-Time Overhead (%)	Trace Analysis	Trace Size
Aget1	12.45s	26%	0.01s	24.7K
FFT	0.05s	2724%	0.01s	1.2M
MySQL1	0.18s	21.1%	0.27s	2.9M
Moz1	0.18s	444%	1.57s	36M
MySQL2	0.13s	157%	0.27s	18M
Trans	1.17s	210%	0.01s	132K
Moz2	12.0s	1065%	0.01s	490K
MySQL3	0.46s	130.2%	6.77s	67M
MySQL4	0.10s	135.5%	0.13s	17M
Cherokee	11.26s	21.28%	11.45s	115M
Click	0.02s	3846%	0.01s	80K

Table 4.14: Performance of trace collection and analysis (Base Line is the time for the original test run w/o any instrumentation.)

Experience with ConSeq-Daikon

As discussed in Chapter 4.3.2, we played some tricks to get around the granularity limitations of Daikon's frontend. For MySQL, we replaced the default frontend with our own PIN tool. Daikon's backend generates invariants from the log dumped by our PIN tool. For Aget, a relatively small program, we manually inserted dummy functions before *every* global-variable read and then instructed Daikon's frontend kvasir-dtrace to record all global variables at the exit of these dummy functions. Note that this is not a fundamental limitation of Daikon. In fact, the commercial version of Daikon can provide invariants at instruction-level granularity, and could have been used straight out of the box.

MySQL1: After a training phase with a mix of 50 INSERT, 25 SELECT, and 25 DELETE queries to MySQL server, Daikon produced a total of 338 equality invariants, each associated with one instruction that reads a global variable. ConSeq considers these instructions as potential critical reads and detects that 13 out of these 338 invariants could be violated by pure interleaving changes. Among these 13, one of them points to a read of `binlog::log_type` inside function `MySQL::insert`. Daikon observes that this variable's value is always 3 (i.e., `LOG_OPEN`), while ConSeq finds that the value could become 0 (i.e., `LOG_CLOSED`) under an alternative interleaving. This turns out to be exactly the MySQL-1 bug. The abnormal `LOG_CLOSED` value would cause MySQL to miss some logging entries.

Among the other 12 possible violations reported by ConSeq, ten of them are false positives that cannot actually occur due to custom synchronization. The other two can truly occur and violate the candidate invariants proposed by Daikon. However, they are not bugs and do not lead to software failures.

Aget: Aget contains a concurrency bug that involves 11 threads and 21 shared variables (1 scalar and 20 entries in an array of structs), as shown in Figure 4.16. For purposes of illustration, we only show 11 involved variables here.

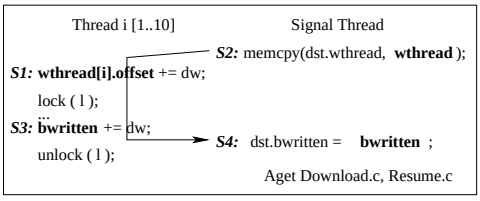


Figure 4.16: A multi-variable atomicity-violation bug that involves 11 threads and many shared variables.

After 50 training runs, Daikon generates equality invariants at 20 program locations. Based on that information, ConSeq reports 2 bugs. **The first bug is Aget1, which cannot be accurately detected by any previous tool. The second is a new bug in Aget that has never been reported.** Specifically, Daikon showed the following invariants just before S4 in Figure 4.16:

```
..dummy_bwritten_005():::EXIT
```

```
::bwritten == sum(::wthread[].bwritten)
```

It means the sum of each thread's `wthread[i].bwritten` fields (across all 10 worker threads) should be equal to `bwritten`, a global variable representing the total number of bytes that have been written by all threads. ConSeq then reports a suspicious interleaving (Figure 4.16) that could violate this invariant.

One point to note about the new Aget bug that ConSeq found is that it could be triggered by fewer than 11 threads (e.g., two worker threads and one signal thread would suffice). However, when the user does not specify the number of worker threads on the command line, the default is 10. We used the default in the Aget experiment to simulate an in-house tester who wishes to test a system's default configuration. Note that if a (user-supplied) test provokes a complicated situation in the initial run, that is what ConSeq must work with. It has no automatic way of "reducing" the run down to a minimal-size example. Fortunately, Daikon will work with any number of threads and does not have a major scaling problem; Daikon's work remains roughly the same, because each thread does less work when there are more threads. In summary, the Aget experiment illustrates the usage scenario of an in-house tester wishing to test a system's default configuration, and shows that ConSeq has the capability to detect bugs even when the input used generates a complicated interleaving scenario.

4.4 Conclusions

This chapter proposes an effect-oriented approach to detecting concurrency bugs that cause program failures. By focusing on the concurrency bug error-propagation pattern revealed by our characteristics study, this approach effectively and predictively detects concurrency bugs of severe failure symptoms. A general effect-oriented concurrency bug detection framework is proposed. The framework works in the following way: first, it identifies potential error/failure sites in a program; second, it uses static analysis to identify potential critical reads that might affect the potential error/failure sites; third, it uses dynamic analysis to identify suspicious interleavings; finally, it has an optional component that inserts sleeps in the program to

exercise the suspicious interleaving. Two tools – ConMem and ConSeq – are instantiated using the proposed framework. ConMem starts from the errors sites of concurrency memory bugs, which are easier to identify, to detect concurrency memory bugs. ConMem does not need to implement the second step of this framework because the potential critical reads and the potential memory error sites overlap. ConSeq starts from the potential failure sites of concurrency semantic bugs, which are easier to identify, to detect concurrency semantics bugs. ConSeq implements all the steps of this framework.

In our evaluation with over a dozen of real-world severe concurrency bugs, both tools detect more bugs with significantly fewer false positives than data-race and atomicity-violation detectors. In addition, the optional bug exposing component prunes out all false positives and provides a reliable way to expose all the true bugs reported by ConMem and ConSeq. Both tools are able to find previously un-reported bugs. In addition, it also demonstrates that we can leverage tools designed for sequential programs and sequential bugs, such as Daikon, to detect complicated concurrency bugs. Application developers can easily extend and adjust our framework by inserting sequential-style assertions and error messages in their code.

In general, the proposed approach has several nice features to help developers: predictive bug detection, no training requirement, easy-to-validate bug results, high accuracy, and high coverage on concurrency bugs that cause program failures. By looking at the interleaving space from a different perspective, the proposed approach complements existing concurrency bug-detection tools.

Of course, both tools have limitations. First, similar to other dynamic bug-detection tools, they depend on test inputs for code coverage. Second, they both benefit from the short-propagation-distance heuristic; while this heuristic has proven to be effective in our rather inclusive benchmarks, it will inevitably cause some bugs of long error-propagation distance to be missed.

In the future, ConMem and ConSeq can be extended in the following ways. First, for ConMem, we could use static analysis to improve their ability to identify potential memory error sites (e.g., shared pointer deference, shared buffer indexing, etc.) and identify more

types of failure sites (e.g., security vulnerabilities, silent data corruption, etc.). Second, we could try to identify more kinds of customized synchronization and further decrease the remaining false positives. Finally, we could also apply the effect-oriented idea to detecting other types of severe bugs in applications that are written in languages other than C/C++ and use concurrency programming constructs other than the pthread library.

While effect-oriented approach is demonstrated to be a very effective approach to detecting concurrency bugs, it remains to be seen how this approach can be applied to failure recovery. We present our effect-oriented failure recovery tool in the next chapter.

5 EFFECT-ORIENTED CONCURRENCY FAILURE RECOVERY

The last chapter focuses on concurrency-bug detection. Despite much effort developed into bug detection, concurrency bugs do slip into production runs because there is not enough resources to conduct thorough testing. Thus, serving as the last resort to fending off concurrency bugs, failure recovery is much needed.

Guided by the characteristics study presented in Chapter 3, a novel concurrency failure recovery technique ConAir is presented in this chapter. ConAir is a static code transformation tool that transparently inserts feather-weight failure recovery code into software to help it recover from a wide variety of known and hidden concurrency bugs. At the high-level, ConAir is also an effect-oriented approach – it starts from the potential failures sites and analyzes the code backwards to find a large-enough re-execution region within the failure thread. After the re-execution region is identified, ConAir then automatically inserts necessary code so that when a failure occurs, the failure thread is rolled back and re-executed. This chapter first presents the insights and techniques behind ConAir, then demonstrates ConAir's effectiveness by evaluating it with 10 real-world concurrency bugs of different root causes and failure symptoms.

5.1 Introduction

5.1.1 Motivation

Many concurrency bugs are hidden in production-run software, causing severe failures in the field with huge financial losses [102, 112, 70]. When they finally get noticed by developers, correctly fixing them takes substantial manual effort. Developers often need weeks, or even months, to design a concurrency-bug patch [47, 74] yet about 40% of released concurency-bug patches are incorrect, which are the most error-prone among all types of bug patches [132]. Therefore, it is critical to help *end-users* enable production-run software to *survive* failures caused by hidden concurrency bugs and help *developers fix* known concurrency bugs.

An ideal bug fixing and survival technique should have several key properties: *compatibility*, i.e. no OS/hardware modification; *performance*, i.e. small run-time overhead and fast failure recovery; *generality*, i.e. helping bugs with a wide variety of root-cause interleaving patterns without reliance on accurate bug detection; *correctness*, i.e. not generating results infeasible for original software.

Table 5.1 summarizes three solutions to this problem and outlines their differences for all these four properties. As shown in columns 2, 3, and 4, no existing technique can achieve all four properties at the same time. We now elaborate on these techniques.

	Auto. Fixing	Prohibiting Interleaving	Rollback Recovery	ConAir
Compatibility	✓	*	*	✓
Correctness	✓	✓	✓	✓
Generality	-	*	✓	✓
Performance	✓	*	*	✓

Table 5.1: A comparison among concurrency-bug fixing and survival techniques (✓: yes; -: no; *: cannot all be yes at the same time.)

The *automatic fixing* approach statically or dynamically adds synchronization into programs to eliminate *known* bug-triggering interleavings [57, 83, 59]. Although promising, a tool with this approach only fixes bugs with a specific root cause (e.g., atomicity violations [57]), because it requires different types of synchronization to eliminate buggy interleavings of different root-causes. Furthermore, by design, this approach does not help software survive hidden bugs. It only fixes known bugs based on accurate bug root-cause detection.

Proactively prohibiting certain types of interleavings at run time is a common approach to surviving hidden concurrency bugs [119, 79, 108, 19, 106, 104, 69, 134, 135, 126]. Techniques

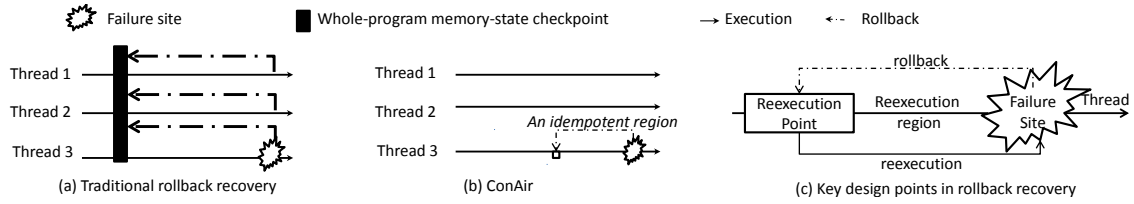


Figure 5.1: An overview of ConAir

based on this approach only survive failures caused by certain types of interleavings, and tend to impose unnecessary serialization and performance loss on existing hardware. In rare cases, some techniques belonging to this approach may need programmer annotations to eliminate the effect of certain interleavings to maintain correctness.

The *rollback recovery* approach leverages the non-determinism of multi-threaded software to survive hidden concurrency bugs [105, 122, 117]. Without prohibiting any interleaving, it uses system checkpoint and rollback techniques to recover from failures, which is not limited to particular types of root causes. Unfortunately, existing techniques based on this approach require periodic *whole-program* checkpoint at run time and *whole-program* rollback for failure recovery. As a result, they require OS/hardware modifications to achieve good performance.

5.1.2 Highlights

This chapter presents ConAir, a static program analysis tool that automatically inserts rollback-recovery code into multi-threaded software and allows software to recover from a wide variety of known and hidden concurrency bugs with little run-time overhead, as shown in Figure 5.1.

ConAir distinguishes itself from existing rollback-recovery systems by the following features:

1. **No multi-threaded rollback**. We observe that failures caused by most concurrency bugs can be survivied through rolling back just *one* thread, instead of all threads.

2. **No memory-state checkpointing**. We observe that failures caused by many concurrency bugs can be survived by re-executing an *idempotent* region surrounding the failure site. A code region is idempotent if it can be reexecuted for any number of times without changing the program semantics. More formal definition of idempotent regions is in Chapter 5.2.

The above observations help ConAir achieve the four properties listed in table 5.1. The reexecution of single-threaded idempotent regions guarantees no change to program semantics

(*correctness*).[1] The rollback-recovery approach, by design, allows software hardened by ConAir to recover from concurrency bugs caused by a wide variety of root causes (*generality*). By avoiding the need for checkpointing memory state and by avoiding coordination across threads, ConAir limits its run-time overhead (*performance*) and requires no modification to OS/hardware (*compatibility*).

ConAir can be used in two modes. In survival mode, ConAir can be applied to harden a multi-threaded program against hidden concurrency bugs. In fix mode, it can generate safe temporary patches for concurrency bugs whose root causes are unknown. This is helpful to developers who often know the failure symptom of a reported bug long before they understand the root cause of the bug.

We have evaluated ConAir by using 10 real-world concurrency bugs in open-source server and client software. These 10 bugs represent bugs of common root causes, including atomicity violations, order violations, and deadlocks, and bugs of common failure symptoms, including assertion violations, wrong outputs, segmentation faults, and hangs. Without any knowledge of these bugs, ConAir automatically hardens the software at 7 – 19185 statically identified potential failure sites per program. The hardened software runs almost as fast as the original software, with only 0 – 0.2% run-time overhead. Failures caused by 8 out of 10 bugs can always be successfully survived. The other 2 bugs lead to wrong-output failures. If the output-correctness conditions are known to ConAir, failures caused by these 2 bugs can also be successfully survived. The time taken for failure recovery varies and is between 8 microseconds and 17 milliseconds. ConAir also has its limitations, which will be discussed in Chapter 5.6.5.

5.2 ConAir overview

The design of a rollback-recovery system for multi-threaded software includes three key components, as shown in Figure 5.1c: (1) how many threads participate in the rollback recovery; (2) where the failure site is in each participant thread; (3) what is the reexecution

[1] ConAir does not violate memory consistency model (see Chapter 5.2).

```
y=x+1;
z=x+y;
```

(a) An idempotent region

```
x=x+1;
z=x+y;
```

(b) Not an idempotent region

Figure 5.2: Idempotency

point in each participant thread. We take a novel approach to these components by leveraging our key observations obtained from our characteristics study in Chapter 3.

To address component (1), *unlike* previous work, only one thread participates in ConAir rollback recovery. This is supported by our characterisitics study's finding 2 that rolling back a single thread (i.e., the failing thread) is effective to recover from most concurrency bug failures.

Components (2) and (3) are synergistically handled by forming idempotent code-regions whose end-point is the failure site, and start-point serves as a natural reexecution point for the participant thread. Identifying the potential failure sites can be easy thanks to our characteristics study's finding 1. It is feasible to use the idempotent code region as the re-execution region is because of our characteristics study's finding 3. The reasoning follows:

In general, an idempotent region is a code region that can be reexecuted for any number of times without changing the program semantics. Figure 5.2 shows a code-snippet that is idempotent contrasted with one that is not. In general, such regions can have arbitrary start and end-points in the program, so long as the code in that region adheres to idempotency semantics.

We narrow the definition to make the regions amenable for bug recovery. In this work, an idempotent region always ends at a potential failure site. It does not contain any writes to shared variables, so that its single-threaded reexecution does not violate the memory-consistency model of hardware and system. It does not contain any I/O operations. It also does not contain any writes to local variables that could cause incorrect reexecution.

Using idempotent regions as reexecution regions can easily achieve *correctness* and good *performance*, because they can be correctly reexecuted without any checkpointing or logging.

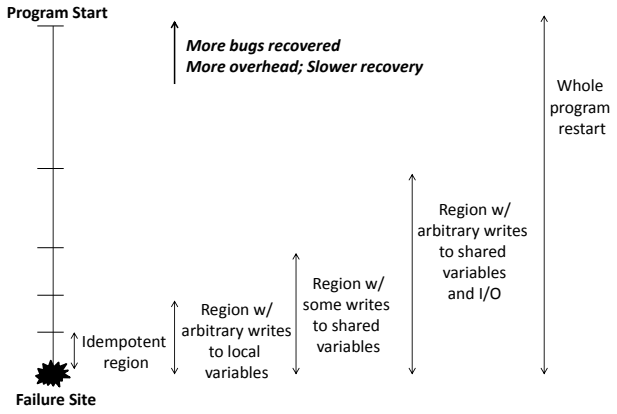


Figure 5.3: The tradeoff of reexecution-region design

The only concern is that they may be too short to help program recover from concurrency bugs. On the other hand, longer reexecution regions can help recover from more bugs, but also hurt *performance* and/or *compatibility* more, because they require more complicated rollback-reexecution techniques, such as sandboxing I/O events, buffering shared-variable modifications, and checkpointing local-variable values. Figure 5.3 sketches this trade-off and design space.

Recall the finding 3 of characteristics study conducted in Chapter 3 is that error propagation distance is short. This finding implies that constraining reexecution regions to be free of idempotency-destroying operations (i.e., I/O operations[2] and shared memory writes) does not significantly eliminate the chance of recovering from concurrency-bug failures.

To know the exact percentage of real-world failures that require re-execute I/O operation or shared memory writes are difficult – most of the real-world concurrency bugs examined in previous work [74] have never been reproduced in a research environment. Therefore, we studied all the 31 bugs repeated and presented by 7 recently published works on concurrency-bug detection and prevention [59, 116, 140, 139, 57, 123, 134].

Among these 31 bugs, 24 can be survived through single-threaded reexecution. Among the reexecution regions of these 24 bugs, only 5 contain I/O operations, and 3 contain shared memory writes but no I/Os. From this study, we can see that many concurrency bugs'

[2]A Recent study of real-world concurrency bugs shows that only about 15% of concurrency bugs' recovery involves I/O operations[123].

reexecution region can fit inside an idempotent region surrounding the failure site.

Traditional techniques mainly trend toward the right end of the reexecution-region design spectrum in Figure 5.3. Their focus of the failure-recovery universality inevitably leads to large run-time overhead and complicated/non-existing platform support. This chapter will explore the leftmost end of the design spectrum. We use idempotent regions as reexecution regions, and identify a reexecution point as the starting point of the idempotent region surrounding each failure site. Our design does not aim the universality of failure recovery. Instead, it aims to survive a significant portion of concurrency-bug failures with a wide variety of root causes at negligible overhead on existing platforms, which will allow easy adoption in production systems.

ConAir framework includes three components:

1. A static analysis component that identifies potential failure sites in software (Chapter 5.3.1).

2. A static analysis component that identifies reexecution points for every potential failure site (Chapter 5.3.2).

3. A static code-transformation component that enables a multi-threaded program to survive concurrency bugs at the failure sites identified above through single-threaded rollback (Chapter 5.3.3).

In the following, Chapter 5.3 presents a basic design and implementation of ConAir. Chapter 5.4 discusses further extensions and optimizations of ConAir, such as how to avoid useless recovery attempts and how to conduct inter-procedural recovery. Chapter 5.5 and Chapter 5.6 present the evaluation of ConAir.

```
//assert(e)

  if(e){
  }else{
Failure:

    __assert_fail(...);

  }
```

(a) Assertion Failures

```
//printf("..",e,..);

  if(Assert(e)){
  }else{
Failure:
//developers specify
//Assert(...)
  }
  printf("..",e,..);
```

(b) Wrong Outputs

```
//tmp=*G_ptr;
  l_ptr=G_ptr;
  if(l_ptr>LowerBound){
  }else{
Failure:


  }
  tmp=*l_ptr;
```

(c) Segmentation Faults

```
//pthread_mutex_lock(..);
  int ret = pthread_mutex_timedlock(..);
  if (ret!=ETIMEOUT ){
  }else{
Failure:


  }
```

(d) Deadlock Failures

Figure 5.4: Failure sites for different types of failures (Some of them involve ConAir code transformation; LowerBound is 10,000 by default.)

5.3 ConAir design and implementation

5.3.1 Failure site identification

Failure sites are where failures occur. Some failures may occur due to hidden bugs and some failures may have already been manifested with their symptoms known to users/developers. To handle these two types of failures, ConAir operates in two modes: survival mode and fix mode. These two modes only differ in how the failure sites are identified.

Identifying failure sites in survival mode

Without any knowledge of hidden concurrency bugs, ConAir uses static analysis to identify program locations where common failures could occur. The following four types of failures are the most common among the real-world concurrency bugs according to our characteristics study.

Assertion failures. The assert macro is widely used by developers to specify critical program properties. In Linux systems, an assertion failure will cause the execution of

___assert_fail(...). ConAir identifies the invocation of ___assert_fail(...) as a (potential) failure site, as shown in Figure 5.4a.

Wrong outputs. Wrong output failures occur when software produces an incorrect output or fails to produce any output when an output is desired. Judging a wrong-output failure requires oracles specified by developers or users. The current prototype of ConAir can help recover from wrong-output failures, if developers can provide output oracles in the format of assert as shown in Figure 5.4b.

Segmentation-fault failures. In ConMem, we show that most segmentation faults caused by concurrency bugs occur during the dereference of a heap/global pointer variable. Therefore, ConAir identifies every dereference of a heap/global pointer variable as a potential segmentation fault failure site, as shown in Figure 5.4c.

Deadlock failures. There are different ways to detect a deadlock failure. Some previous work [59] instruments Pthread library functions and reports deadlocks by catching cycles in the run-time resource-acquisition graph. Many real-world multi-threaded systems, such as MySQL [86], simply maintain a timer for each lock acquisition function and report a deadlock once the lock-acquisition times out. ConAir can work with any deadlock-detection mechanism: the detection code that reports a deadlock is treated as a (potential) failure site. Our current prototype assumes the time-out based deadlock detection. ConAir transforms every pthread_mutex_lock function into pthread_mutex_timedlock, and identifies failure sites accordingly as shown in Figure 5.4d. ConAir can handle customized lock functions, as long as the developers specify the prototypes of their lock, unlock, and timeout-lock functions.

ConAir does not require its failure-site identification to be sound or complete. Inevitably, many sites identified above never fail. Treating them as potential failure sites only causes negligible run-time overhead, as we will see in Chapter 5.6.2, benefiting from ConAir's low-overhead design. The above analysis can be easily customized to cover more types of failures or to focus on a smaller set of severe failures.

Identifying failure sites in fix mode

Fix mode can be used when users or developers encounter a non-deterministic failure with an *unknown* root cause. In this case, users or developers inform ConAir of the failure location. For example, when the bug shown in Figure 3.5(b) manifests itself, users or developers will observe a segmentation fault at the statement tmp=∗ptr, which ConAir treats as the failure site.

5.3.2 Reexecution point identification

As discussed in Chapter 5.2, the placement of reexecution points and reexecution regions largely determines the system performance. ConAir uses idempotent regions as its reexecution regions during failure recovery. Each reexecution point is the starting point of an idempotent region, which ends at a potential failure site. This design makes ConAir lightweight and able to help recover from many, although not all, concurrency-bug failures.

Principle of identifying idempotent regions

Identifying idempotent code regions is not trivial. A code region that is *not* idempotent in source code, such as x=x+1, could become idempotent in bitcode, such as x_1=x_0+1, due to variable renaming conducted by a compiler. A code region that is idempotent in bitcode could later become *not* idempotent in binary code due to physical-register allocation. Due to these challenges, there are usually two approaches to identifying idempotent code regions in the binary code. One is to rely on binary code analysis alone. Unfortunately, this could be very complicated for x86 code. The second approach, which is used by recent work [27], is to use a combination of bitcode/binary-code analysis and bitcode/binary-code transformation.

ConAir takes the second approach using the LLVM static analysis and code generation framework [66]. As discussed in Chapter 5.2, an idempotent region does not contain shared-variable writes, non-idempotent local-variable writes, or I/O operations. Following this, ConAir identifies an idempotent region as an LLVM bitcode region that contains none of the following *idempotency-destroying instructions*: (1) writes to global or heap variables; (2) writes

to local variables that are not allocated in virtual registers[3]; (3) function-call instructions. This code region is guaranteed to be idempotent at bitcode level.

To guarantee the region is also idempotent in the binary code, ConAir performs two transformations. First, ConAir uses the −no−stack−slot−sharing flag for LLVM to generate the binary code. This flag guarantees that different virtual registers, when not allocated in physical registers, are allocated in different stack slots. Under this configuration, the code regions identified above will always conduct idempotent operations on memory states. The only concern is that these regions may modify the value of a physical register and cause the reexecution to read a different register value from the original execution. Therefore, ConAir saves the register image at the beginning of the code region and restores the register image right before a rollback. The register save and restore are conducted by setjmp and longjmp. They are both very lightweight, taking only a few nanoseconds.

Alternative methods to identify idempotent code regions Some code regions that contain *idempotent-destroying operations* are still idempotent in binary code. For example, writing a stack variable v that is not allocated in virtual registers does not necessarily hurt the idempotency of a code region R, unless this write is preceded by a read of v that is not preceded by another write to v. As another example, some function calls do not hurt the idempotency. With more complicated analysis, we could identify more and longer idempotent regions in the future.

An alternative implementation decision is to modify the register allocator. A recent work [27] first identifies the boundaries of idempotent regions in LLVM bitcode, it then modifies the compiler back-end code generator to guarantee that idempotent bitcode is translated to idempotent binary code [27]. For our work, we took the setjmp/longjmp approach because it is easier to implement and is ISA independent. A production use of ConAir could employ either approach. The previous work [27] also splits the whole program into idempotent code regions, covering every instruction by idempotent regions. In contrast, our work only identifies

[3]In LLVM, a virtual register is a variable in static single assignment form (SSA) [23]. It is statically assigned only once.

idempotent regions that end at potential failure sites. This allows us to achieve negligible overhead ($< 1\%$) in our experiment (Chapter 5.6). On the contrary, previous work [27] could have more than 10% run-time overhead.

Algorithm of identifying idempotent regions

When a program does not contain any branch instruction, identifying reexecution points is straightforward. For every failure site f, we simply need to analyze statements one by one backwards until we find the first statement s that is an *idempotency-destroying* instruction. The reexecution point is right after s.

Unfortunately, real programs always contain branch instructions and there could be multiple execution paths leading to a failure site f. Therefore, we have to identify an appropriate reexecution point along every path leading to f.

ConAir conducts a backward depth-first search from f. This static analysis starts with pushing the predecessors of f in the control-flow graph (CFG) into a work-list stack, and keeps processing the top statement in this stack as follows. (1) When the analysis encounters an idempotency-destroying operation, ConAir identifies a reexecution point right after this operation. ConAir then removes this statement from its work list. (2) When encountering the entrance of function containing f, ConAir identifies it as a reexecution point and removes it from its work list. This decision means that ConAir reexecution does not touch the caller of f. We will revisit this decision and discuss inter-procedural recovery in Chapter 5.4. (3) When encountering other statements, ConAir checks how many predecessors of this statement have not been visited. If there is none, ConAir removes this statement from the work list. Otherwise, ConAir pushes an unvisited predecessor of this statement to the top of its work list. ConAir stops its analysis when its work list is empty. At that point, all reexecution points for f are identified. The complexity of this analysis is linear to the static function size.

ConAir repeats the above algorithm for every failure site. Note that the reexecution points of different failure sites do not conflict with each other. That is, the reexecution region of a failure site f_1 will never get shortened by the reexecution points of another failure site f_2. The

```
1
2
3
4
5
6
7
8  if(e){
9  }else{
10
11
12
13
14    __assert_fail(..);
15 }
```

(a) Original code

```
1    __thread jmp_buf c;
2    __thread int RetryCnt=0;
3    ...
4  Reexecution:
5    setjmp(c);
6    ...
7    //reexecution region
8    if(e){
9    }else
10 Failure:
11     while(RetryCnt++<maxRetryNum){
12       longjmp(c,0);
13     }
14     __assert_fail(..);
15   }
```

(b) Transformed code

Figure 5.5: ConAir code transformation for assert(e)

reason is that a reexecution point is always right after an idempotency-destroying operation or at the entrance of a function, which is the same for all failure sites.

5.3.3 Transformation at failure sites and reexecution points

After identifying failure sites and reexecution points, ConAir performs the following code-transformations that enable the multi-threaded software to recover from concurrency-bug failures.

At every reexecution point, ConAir inserts a setjmp (line 5 in Figure 5.5(b)) to make sure our reexecution region is idempotent. Sometimes, multiple failure sites may share a common reexecution point. In these cases, ConAir makes sure to insert just one, instead of multiple, setjmp at the common reexecution point.

At every failure site, ConAir inserts a longjmp to rollback the execution to the reexecution point (line 12 in Figure 5.5(b)). While restoring the register image c, longjmp automatically changes the program counter to the reexecution point where c was taken. This naturally accomplishes the control-flow rollback. ConAir supports multiple reexecutions through the loop on line 11, because some failures may require several rounds of reexecution to recover. The loop-condition variable is a configurable threshold that prevents endless recovery attempts. Its default value is one million.

```
Reexecution:
lock(&L);//blocked
```
(a) Can**not** be survived

```
Reexecution:
lock(&L0);
lock(&L); //blocked
```
(b) Could be survived

```
Reexecution:
tmp=tmp+1;
assert(tmp);//violated
```
(c) Can**not** be survived

```
Reexecution:
tmp=global_x;
assert(tmp);//violated
```
(d) Could be survived

Figure 5.6: Some failure sites cannot be survived by ConAir (The last line in each figure is a potential failure site)

ConAir uses a *thread-local* variable to save the register image at every reexecution point (line 1 in Figure 5.5). At run time, this variable always keeps the register image taken at the most recent reexecution point in a thread. This guarantees that the program will roll back to the right reexecution point.

Deadlock recovery can potentially lead to a livelock problem. This occurs when multiple threads involved in the deadlock try to rollback at exactly the same time. This issue can be solved by putting a small random sleep at the failure site.

5.3.4 Discussion

Future work can extend ConAir by extending its failure-site identification. Some potential failure sites could be pruned, if we can statically prove that failures can never occur there. For example, analysis could know that NULL-pointer dereference may never occur at some places [20]. We can also use dynamic technique like ConSeq to prune well tested potential failure sites. We can also enlarge the set of potential failure sites based on developers' annotations or automatically inferred specifications. For example, ConAir currently inserts an assertion before every fputs function call to check whether the parameter of fputs is NULL or not.

Future work can also explore other designs of the reexecution regions. For example, some regions that write shared variables can be correctly reexecuted with more sophisticated rollback or checkpoint techniques.

5.4 Optimizations and Extensions

This chapter discusses how we extend the basic design of ConAir to recover from more concurrency bugs and optimize the basic design to achieve better performance.

5.4.1 Extending reexecution regions for library functions

The basic design of ConAir reexecution regions is very stringent: it cannot contain any function calls. In the following, we extend the basic reexecution regions to include some library function calls.

Why do we need to reexecute library functions? Some failures cannot be survived unless some library-function calls are reexecuted. For example, if we do not allow a reexecution region to call pthread_mutex_timedlock, ConAir can never help recover from a deadlock failure shown in Figure 5.4d. In fact, a reexecution region has to include a call of pthread_mutex_lock to recover from deadlock failures, which we will discuss later.

Which library functions can be correctly reexecuted? Some library functions can be correctly reexecuted by executing *compensation functions* at the failure site. For example, if a code region executes a malloc, we need to call a free at the failure site; if a code region executes a pthread_mutex_lock, we need to call a pthread_mutex_unlock at the failure site.

Some library functions cannot be easily reexecuted. For example, output functions in general are difficult to reexecute without system support. Reexecuting free or pthread_mutex_unlock could also be dangerous. Imagine a code region that frees an object that is allocated before this region starts or releases a lock that is acquired before this region starts. It is almost impossible to correctly reexecute this type of code regions.

Implementing library-function extension in ConAir Following the above observations, we allow ConAir reexecution regions to contain memory-allocation functions and lock functions. Other functions, such as free, unlock, and output functions, are still considered idempotency-destroying and cannot be included in any reexecution region.

To support this extension, three changes are made. First, ConAir instruments every call site of memory-allocation and lock functions to record which region is allocated and which lock is acquired. ConAir checks the return value of pthread_mutex_timedlock to know whether a lock is acquired. Second, ConAir needs to know which memory-regions/locks are acquired in

the *current* reexecution region at a failure site. To support this, ConAir maintains an integer counter for each thread, which is increased by one at every reexecution point. At the return of every memory-allocation/lock function, ConAir stores the starting address of the newly allocated region or the address of the newly acquired lock, as well as the current counter value of this thread, into a per-thread vector maintained by ConAir. Before storing the new record, ConAir cleans the vector, if the current counter value is different from what is stored in the vector. Third, at each failure site, before the longjmp, ConAir inserts code to iterate through the vector, identify every region/lock that is allocated/acquired under the current counter value, and deallocate/free it.

The above extension guarantees the recovery correctness. Note that deallocating a memory region R or releasing a lock L at the failure site in thread t_1 does *not* affect the correctness of other threads. Since a reexecution region cannot contain writes to shared variables, other threads could not have obtained any pointer pointing to R. Furthermore, no other thread could have acquired L before t_1 releases it. Also note that reexecution regions do not contain any free or pthread_mutex_unlock functions. Therefore, we do not need to worry about an object/lock that is allocated/acquired and then freed/released during one reexecution region.

5.4.2 Optimizations to remove unnecessary rollbacks

ConAir cannot help recover from some failures, such as those shown in Figure 5.6a and Figure5.6c. We identify failure sites that are statically proven to be unrecoverable and remove any unnecessary rollback-reexecution code inserted by the basic ConAir algorithm described in Chapter 5.3.

Deadlock failure optimizations To recover from a deadlock failure, ConAir should at least release a lock originally held by the thread at the failure site, as shown in Figure 5.6b. Otherwise, other threads involved in this deadlock cannot possibly make progress during the recovery attempt, and hence ConAir has no chance to help recover from the deadlock. ConAir optimization follows this intuition. Each deadlock failure site f could correspond to different reexecution regions along different execution paths. ConAir checks whether there is a

lock-acquisition operation inside at least one reexecution region of f. If there is none, no lock will be released at f and there is no chance for deadlock recovery. Therefore, ConAir removes the failure-recovery code at f. The current prototype of ConAir turns pthread_mutex_lock functions into pthread_mutex_timedlock functions when it identifies potential deadlock sites, as discussed in Chapter 5.3.1. Once ConAir identifies a failure site to be unrecoverable, the corresponding pthread_mutex_timedlock is turned back to pthread_mutex_lock.

Non-deadlock failure optimizations To recover from a non-deadlock failure, the reexecution conducted by ConAir should include at least one shared-variable read that can affect the evaluation outcome at the failure site, as shown in Figure 5.6d. Otherwise, the reexecution is guaranteed to fail again. Following this intuition, ConAir checks every non-deadlock failure site f. ConAir first uses intra-procedural static backward slicing to identify global/heap memory-read instructions that can affect f through data dependence and/or control dependence. ConAir then checks whether there is at least one such read instruction that is inside a reexecution region of f. If not, f is not recoverable and no failure-recovery code is inserted for it.

Our intra-procedural backward-slicing analysis is implemented in LLVM to analyze LLVM bitcode. Interestingly, our analysis is much simpler than general backward-slicing algorithms.

A major source of complexity in general slicing analysis is tracking data dependence through memory accesses, which requires pointers-alias analysis, as shown by the dotted line in Figure 5.7a.

ConAir does *not* have this concern. Recall that write instructions in a ConAir reexecution region only write to virtual registers (Chapter 5.3.2), such as the write to %0, %1, and %2 in Figure 5.7b. Therefore, when ConAir backward slicing encounters a read instruction r that does not read from a virtual register, such as line 2 in Figure 5.7b, ConAir simply *stops* tracking its data dependence. The reason is that the instruction that provides value for r must write to non-virtual-register locations (e.g., line 1 in Figure 5.7b) and do not belong to ConAir idempotent regions. Slicing outside an idempotent region, and hence a reexecution region, is useless for ConAir.

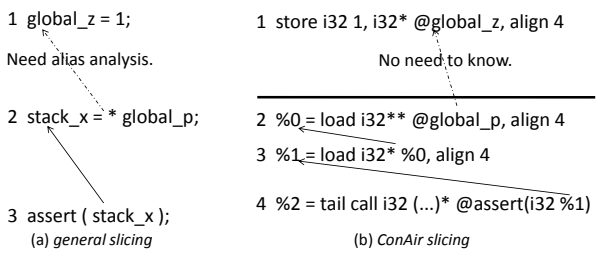


Figure 5.7: The difference between general slicing analysis and the slicing in ConAir (the stack_ prefix denotes stack variables; the global_ prefix denotes global variables; a solid arrow shows an easy-to-get dependence; a dotted arrow shows a difficult-to-get dependence; the thick line in (b) shows the reexecution-region boundary.)

After removing all unrecoverable failure sites, ConAir also removes reexecution points that do not correspond to any failure site and finishes the optimization.

5.4.3 Inter-procedural reexecution

The basic ConAir algorithm presented in Chapter 5.3 only attempts intra-procedural recovery. That is, the reexecution point for a failure site `f` is always in the function that contains `f`, referred to as foo. In this chapter, we discuss pushing reexecution points into the callers of `f`. We refer to this as *inter-procedural recovery*. Inter-procedural recovery can help recover from more failures, but can also hurt performance.

When is inter-procedural recovery correct? We should not attempt an inter-procedural recovery, if doing so would generate results infeasible for the original program. Following the discussion in Chapter 5.3.2, when there is no idempotency-destroying operation on a path between `f` and the entrance of foo, it is safe to extend the reexecution region into the caller of foo.

When can inter-procedural recovery help? Since a ConAir reexecution region cannot contain any modification to shared variables, parameters are the only ways for a caller to affect the execution outcome at `f`. Therefore, inter-procedural reexecution can potentially help the recovery of a non-deadlock failure, only when a parameter of foo is on the backward slice of `f`.

When do we need inter-procedural recovery the most? It is difficult to accurately predict when inter-procedural recovery will be needed without knowing the hidden bugs. Intuitively, imagine a path p between the entrance of foo and f. As discussed in Chapter 5.4.2, reexecuting p cannot help recover from a non-deadlock failure at f, if p contains no shared-variable read that can affect the outcome of f. Similarly, reexecuting p cannot help recover from a deadlock failure at f, if p contains no lock-acquisition functions. Therefore, we hypothesize that an inter-procedural recovery is most needed when such an *unrecoverable* path p exists.

Conditions of ConAir inter-procedural recovery Inter-procedural recovery could help recover from more failures. However, it significantly slows down ConAir static analysis, which we will see in Chapter 5.6; it will also identify more reexecution points with more setjmp executed at run time, which incurs more overhead.

Based on these considerations, ConAir selects a failure site f for inter-procedural recovery when f satisfies all the following three conditions: (1) There is no idempotent-destroying operation on any path between the entrance of foo and f. This way, once selected for inter-procedural recovery, the recovery attempt of f is always conducted interprocedurally, no matter which path is followed in foo during the failure run. (2) At least one argument of foo is on the backward slice of f, when f is a non-deadlock failure site. This way, the inter-procedural reexecution can potentially help recover from the failure. This parameter is referred to as a *critical parameter*. (3) At least one path between the entrance of foo and f is unrecoverable.

How to identify inter-procedural reexecution point? When a failure site f is identified for inter-procedural recovery, ConAir uses static analysis to find every function foo_1 that calls foo. Inside foo_1, we use the analysis described in Chapter 5.3.2 to look for reexecution points. This analysis starts from the instruction that pushes the critical parameter onto the stack of foo, when f is a non-deadlock site; it starts from the invocation of foo, when f is a deadlock site.

We then identify reexecution points just as during intra-procedural recovery (Chapter 5.3.2).

Note that the setjmp and longjmp inserted at reexecution points and failure sites handle the program-counter register and the stack-frame registers. Therefore, no extra effort is needed for inter-procedural rollback.

While analyzing the caller of foo, we could decide to try inter-procedural recovery again. In our current prototype, we set a threshold of how many levels of inter-procedural recovery we would attempt for one initial failure site f. The default setting is 3. That is, to help recover from f inside a function foo, ConAir could at most rollback the execution to the callers' callers' caller of foo, referred to as foo_3. This threshold is configurable. It balances the recovery capability and run-time performance.

Theoretically, there could be a path between the entrance of foo_3 and f that does not contain any idempotent-destroying operations. Since we decide not to go further into the caller of foo_3, we could choose to set the reexecution point at the entrance of foo_3. However, this scheme could prevent failure sites inside foo_3 to attempt inter-procedural recovery. Therefore, in our current prototype, ConAir simply gives up the inter-procedural recovery attempt of f in that case and puts the reexecution point back to the entrance of foo. Note that, this case is extremely rare and has *never* occurred in any of the applications evaluated by us.

Other issues Our inter-procedural recovery analysis needs to work together with the intra-procedural recovery analysis discussed in Chapter 5.3 and the optimization analysis discussed in Chapter 5.4.2. ConAir first conducts intra-procedural analysis. This analysis could identify the entrance of foo as a reexecution point for a failure site f, referred to as $\mathsf{RE}_{\text{intra}}$. ConAir then conducts inter-procedural recovery analysis. Once f is identified for inter-procedural recovery, ConAir safely removes the reexecution point $\mathsf{RE}_{\text{intra}}$[4]. Finally, ConAir conducts its optimization analysis discussed in Chapter 5.4.2. This optimization is *only* applied to failure sites that conduct intra-procedural recovery. Failure sites that are selected for inter-procedural recovery usually have long reexecution regions. It is much harder to statically prove them to be unrecoverable.

[4]This could cause some other failure sites in foo to conduct interprocedural recovery too, which is fine.

App.	App. Type	LOC	Failures	Causes
FFT	Scientific computing	1.2K	w. output	A/O Vio.#
HawkNL	Network library	10K	hang	deadlock
HTTrack	Web crawler	55K	seg. fault	O Vio.
MozillaXP	XPCOM: cross platform component object model	112K	seg. fault	O Vio.
MozillaJS	JavaScript engine	120K	hang	deadlock
MySQL1	Database server	681K	w. output	A Vio.
MySQL2	Database server	693K	assertion	A Vio.
Transmission	BitTorrent client	95K	assertion	O Vio.
SQLite	Database engine	67K	hang	deadlock
ZSNES	Game simulator	37K	assertion	O vio.

Table 5.2: Applications and Bugs (w. output: wrong output failures; A Vio.: atomicity violations; O Vio.: order violations; #: There are both order violations and atomicity violations in FFT.)

5.5 Experimental Methodology

Our work aims to allow programs to recover from a significant portion of real-world concurrency-bug failures of a wide variety of root causes, while incurring low overhead on existing platforms. To empirically evaluate whether ConAir has achieved this goal, our experiments look at 10 real-world concurrency-bug failures in 10 open-source applications that have been widely used in previous bug detection and avoidance research [59, 116, 140, 139, 57, 123]. They represent a wide variety of failure symptoms and root causes, as shown in table 5.2. We will quantitatively evaluate whether ConAir can indeed help recover from failures with different root causes; what is the run-time overhead introduced by ConAir; how long it takes to recover from a failure under ConAir; and the static-analysis complexity of ConAir.

We apply ConAir to analyze and transform each application twice, representing fix mode and survival mode respectively.

While applying ConAir in survival mode, ConAir needs **no** knowledge of failures or bugs. It automatically identifies potential failure sites as discussed in section 5.3.1 and transforms software.

While applying ConAir in fix mode, ConAir assumes the knowledge of failure sites provided by developers or users who want to fix a particular failure they observed. This could be a specific assert that is violated; a particular pthread_mutex_lock that blocks the program; a particular memory-access instruction that causes a segmentation fault; or an output function

that generates incorrect results. Note that ConAir needs **no** information about bug-triggering inputs, bug root-causes, or bug-detection results.

To evaluate whether the software can survive the manifestation of a bug, we insert sleeps into each program's buggy code regions to force the occurrence of the failure-inducing interleaving. Executed under this setting and failure-inducing inputs, the software in our benchmark set fails with almost 100% probability, if ConAir is not applied. After ConAir is applied, we execute the software under the same setting for 1000 times. We claim ConAir to have successfully helped recover from the failure if the hardened software executes correctly in all 1000 runs. To evaluate the run-time overhead, we execute the original program and the transformed programs under the same input (i.e., the bug-triggering input) for 20 times each, and calculate the average overhead. No sleep is inserted and software never fails during the run-time overhead measurement.

Among all types of failures, wrong-output failures cannot be recovered unless the users or developers annotate the correctness condition of an output. This condition is easily available in fix mode, but is not necessarily available in survival mode. To better understand the *worst*-case overhead of ConAir in the survival mode, ConAir treats every output function as a potential failure site, even though the correctness condition may be unavailable.

All the experiments are conducted on an 8-core Intel Xeon machine running Linux version 2.6.18 and using the LLVM 2.8 compiler.

5.6 Experimental Results

As shown in Table 5.3, ConAir helps programs recover from all the evaluated bugs. ConAir incurs no overhead in fix mode, and negligible overhead ($< 1\%$) in survival mode. In this chapter, we will explain the following experimental results in detail: (1) how ConAir effectively fixes bugs with known failure symptoms; (2) how ConAir transparently hardens multi-threaded software to survive hidden bugs; (3) how ConAir achieves negligible run-time overhead; (4) the fast failure recovery under ConAir; (5) the static analysis time of ConAir.

App.	Failure Recovered?		Overhead	
	fix	survival	fix	survival
FFT	✓	$✓_c$	0%	0.0%
HawkNL	✓	✓	0%	0.0%
HTTrack	✓	✓	0%	0.0%
MozillaXP	✓	✓	0%	0.0%
MozillaJS	✓	✓	0%	0.0%
MYSQL1	✓	$✓_c$	0%	0.1%
MYSQL2	✓	✓	0%	0.0%
SQLite	✓	✓	0%	0.0%
Transmission	✓	✓	0%	0.2%
ZSNES	✓	✓	0%	0.0%

Table 5.3: Overall failure recovery results (✓: recovered; $✓_c$: conditionally recovered; recovering from these wrong-output failures requires annotations.)

5.6.1 Failure recovery

Fix-mode failure recovery

In fix mode, ConAir is aware of the failure sites and failure symptoms. It inserts rollback-recovery code accordingly.

Among the non-deadlock bugs that are evaluated, five of them (FFT, HTTrack, MozillaXP, Transmission, and ZSNES) cause failures in a thread that reads a shared variable too early; FFT[5] and MySQL2 cause failures due to RAR atomicity violations; MySQL1 causes failures due to a WAW atomicity violation. ConAir can successfully help recover from failures caused by all of them.

Some failure recoveries only roll back a few instructions. For example, Figure 5.8 shows a bug in FFT. In this program, thread 1 could unexpectedly read End (line 3 in Figure 5.8) before thread 2 updates it, causing either an order violation or an atomicity violation and a wrong-output failure. ConAir inserts a setjmp right before the assert, which helps FFT to recover from this failure.

Three of these 10 bugs (Transmission, MozillaXP, and HTTrack) require inter-procedural reexecution for the failure recovery. For example, Figure 5.9 depicts the MozillaXP bug. In MozillaXP, thread 1 could unexpectedly read mThd−>state in function GetState before

[5]FFT contains both order violations and atomicity violations.

```
1 //Thread 1
2 fprintf("Start %d",Init);
3 tmp=End;
4 assert(tmp>0);
5 fprintf("Stop %d, Total %d", tmp, tmp-Init);
```

```
1 //Thread 2
2 //End is 0 until below
3
4 End=time(NULL);
```

Figure 5.8: An atomicity/order violation in FFT that causes a wrong-output failure. If developers specify the output-correctness condition (e.g., the assert above), ConAir can help recover from the failure.

the global pointer mThd is initialized by thread 2. This could cause a segmentation-fault failure. ConAir inserts a pointer sanity check right before line 9 in GetState; it also identifies a reexecution point inside function Get and inserts setjmp there. Once ConAir sees an invalid pointer at line 9 in thread 1, the program will automatically jump back to before the invocation of GetState in Get. Eventually, thread 2 will initialize mThd and the program will succeed.

Deadlock recovery is slightly different from the recovery of non-deadlock bugs. Figure 5.10 shows a real-world deadlock bug in HawkNL. As we can see, thread 1 and thread 2 could acquire nlock and slock in reversed orders and lead to a deadlock. ConAir analyzes both threads. When ConAir considers Lock(&slock) (line 8) in thread 1 as a potential failure site, the reexecution region is very short due to the idempotency-destroying operation, driver−>Close(). Since this region does not contain another lock acquisition function, ConAir considers it as unrecoverable and does not attempt any failure recovery in thread 1 (Chapter 5.4.2). When ConAir considers Lock(&nlock) (line 8) in thread 2 as a potential failure site, its reexecution region can go all the way back to before the invocation of Lock(&slock) (line 4) in thread 2. Since this region contains another lock-acquisition function, ConAir considers Lock(&nlock) in thread 2 as a recoverable failure site. ConAir turns it into a lock with timeout and inserts setjmp to the beginning of Shutdown function. At run time, once thread 2 times out at its attempt to acquire nlock, thread 2 will release slock and reexecute a large chunk of Shutdown. This effectively resolves the deadlock problem in HawkNL.

Summary ConAir can effectively fix concurrency bugs with a variety of root causes once the failure sites and symptoms are known.

```
1  //Thread 1
2  Get(){
3    ...
4    tmp=GetState(mThd);
5  }
6
7  GetState(THD *thd)
8  {
9    return(thd->state &
10       THREAD_DETACHED);
11 }
```

```
1  //Thread 2
2  //mThd is shared
3  //between two threads;
4  //it is 0 before
5  //initialized below.
6
7  InitThd(){
8
9    mThd =
10     CreateThd(..);
11 }
```

Figure 5.9: An order violation in Mozilla XPCOM.

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

```
1  //Thread 2
2  Shutdown(){
3    ...
4    Lock(&slock);
5    if(nSockets!=NULL){
6      int i=0;
7      if(nSockets[i]){
8        Lock(&nlock);
9        ...
10     }
11   }
12 }
```

Figure 5.10: A deadlock in HawkNL.

Survival mode

In survival mode, ConAir is **not** aware of any bug. It automatically and systematically identifies potential failure sites and transforms the program accordingly.

As shown in table 5.4, ConAir has identified and hardened 7 – 19185 static failure sites in each benchmark program. Naturally, ConAir identifies the fewest failure sites in the smallest programs (FFT and HawkNL) and the most failure sites in the largest programs (MySQL1 and MySQL2). In general, potential segmentation-fault sites dominate all types of potential failure sites, because ConAir identifies every heap/global pointer dereference as a potential segmentation-fault site. Potential deadlock sites are the fewest among all four types of failure sites, because only a lock operation that is enclosed by another lock operation with no write to shared variables in between is identified as a potential deadlock site that is recoverable by

App.	Assertion Violation	Wrong Output	Seg. Fault	Dead-lock	Total
FFT	5	34	14	0	53
HawkNL	0	0	5	2	7
HTTrack	657	504	3146	0	4307
MozillaXP	1	117	6791	0	6909
MozillaJS	0	5	134	6	146
MYSQL1	119	3256	15791	19	19185
MYSQL2	518	2853	15498	21	18890
SQLite	0	25	47	1	73
Transmission	430	190	2151	0	2771
ZSNES	1	50	331	0	382

Table 5.4: Static failure sites hardened by ConAir

ConAir. HTTrack developers left many assertions in the program, leading to a large number of potential assertion-violation sites.

These automatically identified potential failure sites include the failure sites of all the 10 bugs that are evaluated. Therefore, ConAir can help software successfully recover from these hidden bugs.

Note that survival-mode ConAir identifies every output functions, including fprintf, printf, application-specific functions, such as my_printf in MySQL and js_printf in Mozilla, and others as a potential site of wrong output. The current prototype of ConAir needs developers' specification to help recover from a wrong-output failure, as shown in Figure 5.8. We believe this effort is worthwhile for hardening critical outputs. Future work can also use likely-invariant inference tools [35] to infer such specifications for an output function, and automate the wrong-output failure recovery process.

Summary The above evaluation shows that ConAir is effective to help software survive failures caused by hidden bugs.

5.6.2 Runtime overhead

The run-time overhead of ConAir comes from four sources: (1) code inserted at every reexecution point; (2) extra condition-checking at the failure sites, such as sanity checking

App.	Survival Mode		Fix Mode	
	Static	Dynamic	Static	Dynamic
FFT	56	24	5	5
HawkNL	7	7	1	1
HTTrack	3570	12995	3	4
MozillaXP	3647	2170	1	23
MozillaJS	144	6	1	1
MYSQL1	12494	215218	1	20
MYSQL2	13031	82394	1	30
SQLite	142	7	1	1
Transmission	2568	4425	3	8
ZSNES	321	32	1	2

Table 5.5: The number of reexecution points inserted by ConAir

for pointers at potential segmentation-fault sites; (3) code inserted at call site of memory-allocation and lock functions. (4) using the −no−stack−slot−sharing LLVM linking flag. Among these four, the first one is the dominant source.

To understand the runtime overhead of ConAir, we have counted the number of reexecution points in the hardened programs.

As shown in Table 5.5, ConAir introduces 6 – 215218 dynamic reexecution points in survival mode. Considering that each reexecution point only takes a few nanoseconds to execute (a setjmp and a local counter increment), the low overhead of survival-mode ConAir is understandable. Naturally, the fix-mode ConAir introduces only a few reexecution points, as shown in table 5.5. Its overhead is not perceivable.

There are mainly two reasons that ConAir only requires a relatively small numbers of reexecution points. First, the reexecution points are identified according to potential failure sites. Different from previous work [27], ConAir does not aim to find a reexecution point for every instruction in the program. Instead, it targets on common failures of concurrency bugs. Second, ConAir optimization discussed in Chapter 5.4.2 has helped to remove failure sites that are not recoverable under ConAir and corresponding reexecution points.

To quantitatively demonstrate the optimization effect, we have tried to harden each program by survival-mode ConAir with and without ConAir optimization. As we can see in table 5.6, the optimization effect is significant for deadlock reexecution points: 30–91% of static reexecution

App.	Non-Deadlock		Deadlock	
	Static	Dynamic	Static	Dynamic
FFT	2.0%	5.0%	N/A	N/A
HawkNL	50%	50%	33%	83%
HTTrack	42%	5.4%	N/A	N/A
MozillaXP	2.4%	1.7%	N/A	N/A
MozillaJS	0.0%	0.0%	50%	50%
MYSQL1	1.1%	8.2%	88%	99%
MYSQL2	0.46%	14.6%	91%	100%
SQLite	3.4%	0.0%	30%	71%
Transmission	4.5%	1.76%	N/A	N/A
ZSNES	6.8%	36.4%	N/A	N/A

Table 5.6: The percentage of reexecution points that are optimized (N/A: the non-optimized version has 0 reexecution point).

points can be optimized away. Many lock operations are not enclosed by another lock operation in its reexecution region, and hence are considered as not recoverable. In comparison, the optimization effect for non-deadlock reexecution points is not as significant. Fewer than 10% of static or dynamic reexecution points are optimized away for most benchmarks. The reason is that the optimization cannot eliminate any segmentation-fault reexecution points. In the current prototype of ConAir, the potential site of a segmentation fault is the dereference of a global/heap pointer variable. Since the reexecution regions of this type of failure sites always contain a read of global/heap variable (i.e., the pointer) that can affect the failure outcome, ConAir considers them un-optimizable. HTTrack has a large number of reexecution points that are not related to segmentation faults. Therefore, a significant number of its reexecution points are optimized away.

Summary Benefiting from its single-threaded idempotent reexecution design, its failure-oriented idempotent region identification, and its optimization analysis, ConAir can effectively improve the reliability of production-run software almost for free.

5.6.3 Recovery time

Recovery time affects the availability of production-run software. We quantitatively measure the failure-recovery time under ConAir, and compare it with the time of restarting the whole

Application	ConAir Recovery		Restart
	Time (µs)	# Retries	Time (µs)
FFT	907	97	3189072
HawkNL	59	1	943
HTTrack	4237	474	10776
MozillaXP	17388	8432	207041
MozillaJS	44	1	472
MYSQL1	6014	575	26308
MYSQL2	8	1	836177
SQLite	86	1	1443
Transmission	6476	761	553109
ZSNES	1022	123	8643

Table 5.7: Failure recovery time (The experiments are conducted with small amount of noise inserted to help trigger the concurrency-bug failures).

program.

Note that software restart almost always changes the program semantics perceived by users, unless it can log all the inputs and external signals, and sandbox I/O operations. In addition, the recovery time of software restart becomes worse with the workload getting larger. Instead, the recovery time of ConAir is largely oblivious of the workload. Therefore, the advantage of ConAir recovery in practice would be much more significant than the quantitative results presented below.

As shown in table 5.7, the failure recovery in ConAir ranges between 8 microseconds and 17 milliseconds. In contrast, program restart could take as long as several seconds when the failure occurs at the end of a scientific computation (FFT). The recovery-time speedup of ConAir ranges from 8 times to over 100,000 times.

The ConAir recovery speed is mainly determined by the root cause of the failure. Failures caused by RAR atomicity violations Figure 3.5(c) are always fast to recover. The failing thread does not need to wait for any other thread. Once it reexecutes the read-after-read, the atomicity violation is immediately eliminated and the software immediately recovers. That is why MySQL2 takes only 8 microseconds to recover. Deadlock bugs (HawkNL, SQLite and MozillaJS) also require relatively short recovery time. After one thread t_1 involved in the deadlock releases a lock at the failure site, another thread t_2 can almost immediately jumps

out of the deadlock situation. The recovery time for t_1 will be determined by the critical region length of t_2. Failures caused by order violations usually require a relatively long time to recover. Take the MozillaXP bug shown in Figure 5.9 as an example. At run time, thread 1 reads mThd too early and has to rollback due to an invalid value in mThd. Rolling back thread 1 once may not help recover from the failure, because thread 1 has to wait for thread 2's progress. In our experiment, this rollback is conducted more than 8000 times until thread 2 initializes mThd. This is the main reason of the relatively long recovery time of HTTrack, MozillaXP, Transmission, and ZSNES.

Summary Our evaluation shows that ConAir supports fast failure recovery. It can help software survive failures with little impact on latency and availability.

5.6.4 Static analysis time

The static analysis and code transformation time of ConAir ranges from less than a second (FFT) to around 4 hours (MySQL). The majority of the time is spent in attempting inter-procedural failure recovery. In fact, the basic intra-procedural static analysis discussed in Chapter 5.3 and the optimization analysis discussed in Chapter 5.4.2 together take only 50 seconds for MySQL and fewer than 10 seconds for other benchmarks. The default level of inter-procedural analysis is currently set as 3. For the three bugs that require inter-procedural analysis to recover, two of them (HTTrack and Transmission) require two-level-inter-procedural analysis and one (MozillaXP) requires one-level-inter-procedural analysis.

Summary The static analysis of ConAir is fast enough to process large real-world multi-threaded software. If the time budget is tight, ConAir users can disable the inter-procedural recovery analysis.

5.6.5 Limitations of ConAir

ConAir does not aim to make programs recover from all concurrency-bug failures, which inevitably requires much higher run-time overhead and/or complicated platform support.

Bug ID	Causes	Error	Failure	ConMem	ConSeq	ConAir
FFT	O Vio	mem	wop	✓	✓	✓$_c$
MySQL1	A Vio	sem	wop		✓	✓$_c$
Mozilla1	M.A. Vio	sem	assert		✓	✓
MySQL2	M.A. Vio	sem	assert		✓	
Transmission	O Vio	mem	assert	✓	✓	✓
Mozilla2	O Vio	sem	error msg		✓	✓
Mozilla3	A Vio	sem	error msg		✓	✓
MySQL3	A Vio	sem	error msg		✓	✓
MySQL4	O Vio	sem	hang		✓	
OpenOffice	O Vio	sem	hang		✓	
MySQL5	A Vio	mem	crash	✓	✓	✓
Pbzip	O Vio	mem	crash	✓		
Apache1	M.A. Vio	mem	crash	✓		
Apache2	A Vio	mem	crash	✓		✓
Cherokee	A Vio	mem	crash	✓		✓
HawkNL	Deadlock	N/A	hang			✓
HTTrack	O Vio	mem	crash	✓		✓
Mozilla4	O Vio	mem	crash	✓		✓
Mozilla5	Deadlock	N/A	hang			✓
SQLite	Deadlock	N/A	hang			✓
ZSNES	O Vio	mem	assert	✓	✓	✓
Aget	A Vio	sem	wop			
Mozilla6	A Vio	sem	crash			

Table 5.8: Ability of ConMem, ConSeq and ConAir to handle concurrency bugs. (✓: recovered; ✓$_c$: conditionally recovered. O Vio is order violation, A Vio is single variable atomicity violation, M.A. Vio is multi-variable atomicity violation, sem is semantic error, mem is memory error, error msg is error message, assert is assertion failure, wop is wrong output.)

Specifically, ConAir cannot help recover from failures that require multi-threaded reexecution or very long reexecution regions, as discussed in Chapter 5.2. Fortunately, as also discussed in Chapter 5.2, many real-world concurrency bug failure recoveries do not require multi-threaded reexecution or long reexecution, and hence can benefit from ConAir. Finally, ConAir cannot help a software to recover from a wrong-output failure, if developers do not provide output-correctness conditions.

5.6.6 ConAir complements ConMem/ConSeq

Table 5.8 compares the coverage of ConMem/ConSeq with the coverage of ConAir. For 23 bugs that we evaluated, only 2 bugs cannot be handled by either ConMem/ConSeq or ConAir, because they both have a long error propagation distance that is beyond the capability of ConMem/ConSeq/ConAir. ConMem/Seq can detect 18 bugs, ConAir can help recover from 15 bugs. 3 deadlock-bug-failures can be survived by ConAir but these 3 bugs cannot be detected by ConMem/ConSeq, because ConMem/ConSeq cannot handle deadlock bugs. 6 bugs cannot

be handled by ConAir but can be detected by ConMem/ConSeq. That is because they either belong to the subtype of order violation bugs that require multiple-thread-reexecution or the reexecution region contains operations such like I/O calls or shared memory writes. As we can see, ConMem/ConSeq and ConAir complements each other to handle a great majority of concurrency bugs of different root cause.

In addition, two points worth nothing are: (1) In order for ConMem/ConSeq to detect the bugs, proper testing inputs are required. Without such, ConMem/ConSeq will fail to detect the bugs. In comparison, ConAir is a pure static analysis tool, thus it does not suffer from such a limitation. As a result, ConAir naturally complements ConMem/ConSeq when testing inputs of good quality are not present. (2) For the bugs detected by ConMem/ConSeq, ConAir can still help either fix them or survive the failures caused by them. On the other hand, ConMem/ConSeq can help identify the root cause of the bug, thus provide a better understanding of the bug.

5.7 Conclusions

This chapter presents ConAir, a static analysis and code transformation tool that helps fix and survive concurrency-bug failures through single-threaded recovery and idempotent processing. The evaluation using 10 real-world concurrency bugs shows that ConAir successfully helps software quickly recover from failures that cover a variety of symptoms and root causes. ConAir works well even when it has no knowledge about a bug.

ConAir is not designed to help recover from all failures, but is effective for a large number of common concurrency-bug failures. It only introduces negligible run-time overhead, less than 1% in our experiments. This good performance is achieved without any change to hardware or operating systems and is suitable for production-run deployment. ConAir's effectiveness is a result of a seemingly serendipitous property: short recovery regions are naturally idempotent. In future work, we hope to investigate whether automatic or programmer-aided transformations can help increase its coverage.

ConAir provides a novel use of existing assertions and error checking in programs. With ConAir, assertions and error-checking code no longer just passively observe system failures and errors. Instead, they actively help ConAir to allow software recover from failures and correct software internal errors. ConAir's creative use of assertions opens up the possibility of sanity checks in multi-threaded programs being useful in deployment and as a recovery tool beyond just a debugging tool. For the future work, we would like to investigate how well this works in the field on widely deployed and used code-bases. Also, we would like to understand developer issues in using such a paradigm.

ConAir introduces a perspective that many points in the design space of rollback/recovery are meaningful, with reexecution regions spanning from tens of instructions to the whole program. Future work can extend ConAir to explore other design points in this large design space.

ConAir well complements ConMem and ConSeq to help improve concurrent program reliability. Guided by the same principle as ConMem and ConSeq, ConAir starts from the potential failure site and conducts recovery work in the single thread. It is demonstrated by the real world applications that such effect-oriented approach is an effective approach that can help both find unknown concurrency bugs (as demonstrated in ConMem and ConSeq) and recover from failures caused by unknown concurrency bugs (as demonstrated in ConAir).

6 CONCLUSIONS

6.1 Contribution

This dissertation makes following three contributions.

- Conducts the first characteristic study that studies the whole life-cycle of concurrency bugs based on 70 real world concurrency bugs. Three key observations are made: (1) The patterns of concurrency errors resemble those of sequential bugs' errors, including both memory errors and semantic errors. (2) Error propagates within one thread. (3) Error propagation distance is short. Based on these findings, novel and effective concurrency bug detection and failure recovery can be achieved.

- Guided by the above findings, effect-oriented concurrency bug detection is proposed. It starts from potential memory error sites and potential semantic failure sites, statically analyzes the software to find the potential critical reads. In the following dynamic program analysis phase, runtime trace is collected online. Offline trace analysis is then conducted to find the bugs. The result shows that such an approach achieves much lower false positive rate while detecting bugs that were never reported or missed by state of the art concurrency bug detection tools.

- Guided by the same findings, effect-oriented concurrency failure recovery is proposed. It starts from the potential failure sites and analyzes the code backwards to find the idempotent regions that surround the failure sites. Then it inserts code so that when the failure occurs the failure thread is rolled-back and re-executed. The results, demonstrated by the real-world applications, show that ConAir only incurs at most 0.2% run overhead while allowing programs to recover from failures caused by a wide variety of root causes and have different symptoms. ConAir also guarantees never introduce any new bug. ConAir is a pure static analysis approach thus there is no need to change OS or hardware to make it work.

To conclude, sharing the same insight and principle, our bug detection and failure recovery work complement each other to improve concurrent software reliability significantly.

6.2 Future work

Three pieces of future work are foreseeable for this dissertation:

6.2.1 Input generation for concurrency bug detection tools

In practice, concurrency bug detection is usually based on dynamic analysis. All these dynamic analysis tools, including our own, require bug-triggering inputs. Thus the ability to detect bugs is limited by if the provided inputs will lead programs to the failure sites. However, generating the high-quality input to trigger a bug is hard. Traditional input generation tools attempt to cover as many code paths as possible. For concurrent software testing, path-coverage is not the only critical metric, because we also need to consider the thread interleaving coverage. How to effectively design input for concurrency bug detection remains an open problem.

6.2.2 Handle I/O and shared memory writes for concurrency failure recovery

Idempotent regions in this dissertation are very restricted – it cannot contain any shared memory write or I/O. Although empirical data shows that such operations are rare in the re-execution region, it would be desirable to be able to recover at the presence of such operations. Viable choices are buffering (aka lazy update) and speculatively-execute-and-undo (aka eager update). Each choice has its own limitation and sometime neither of them is feasible. How to handle the shared memory writes and I/O operations in a re-execution region remains an open problem.

6.2.3 Assertion (invariant) placement for concurrent programs

Assertion is a very powerful tool. It conveys the message of what properties should a program hold. It is helpful for our bug detection and failure recovery tools, as it is an easily identifiable

potential failure site for both ConSeq and ConAir. Furthermore, as demonstrated in the ConAir experiment, as long as assertions do not outnumber heap/global pointer dereferences, to harden them incurs negligible overhead. Thus, clever placement of assertions can greatly help improve concurrent software reliability while having minimal impact on software performance. What assertions to put and where to put them in a concurrent software remains an open problem.

BIBLIOGRAPHY

[1] Invalid memory access. Intel Inspector XE 2011 manual.

[2] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[5] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *CC*, 2005.

[6] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, 2004.

[7] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[8] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA*, 2009.

[9] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[10] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, 1993.

[11] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[12] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, 2009.

[13] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[14] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[15] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI*, 2004.

[16] CERT. Cert advisories. http://www.cert.org/advisories/.

[17] F. Chen, T. F. Serbanuta, and G. Rosu. jpredictor: A predictive runtime analysis tool for java. In *ICSE*, 2008.

[18] Cherokee. Cherokee: The Fastest free Web Server out there! http://www.cherokee-project.com/.

[19] L. Chew and D. Lie. Kivati: Fast detection and prevention of atomicity violations. In *EuroSys*, 2010.

[20] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI*, 2008.

[21] Click. The Click Modular Router Projec. http://read.cs.ucla.edu/click/click, 2010.

[22] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Trans. Program. Lang. Syst.*, 13(4), Oct. 1991.

[24] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

[25] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *MICRO*, 2011.

[26] M. de Kruijf and K. Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *CGO*, 2013.

[27] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *PLDI*, 2012.

[28] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[29] M. Dimitrov and H. Zhou. Anomaly-based bug prediction, isolation, and validation: an automated approach for software debugging. In *ASPLOS*, 2009.

[30] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Not.*, 26:85–96, December 1991.

[31] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multi-threaded java program test generation. *IBM Systems Journal*, 2002.

[32] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[33] Y. h. Eom and B. Demsky. Self-stabilizing java. In *PLDI*, 2012.

[34] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[35] M. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[36] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[37] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[38] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, 2003.

[39] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August. Encore: low-cost, fine-grained transient fault recovery. In *MICRO*, 2011.

[40] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[41] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[42] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.

[43] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[44] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, 2011.

[45] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[46] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, 2005.

[47] P. Godefroid and N. Nagappani. Concurrency at Microsoft – an exploratory survey. Technical report, MSR-TR-2008-75, Microsoft Research, May 2008.

[48] J. Gray. Why do computers stop and what can be done about it? Technical report, Tandem Computers, Inc, June 1985.

[49] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of linux kernel behavior under errors. In *DSN*, 2003.

[50] M. Hampton and K. Asanović. Implementing virtual memory in a vector processor with software restart markers. In *ICS*, 2006.

[51] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference*, 1992.

[52] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Syst. J.*, 7(2):74–84, June 1968.

[53] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *TOPLAS*, 1990.

[54] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two hardware-based approaches for deterministic multiprocessor replay. *Commun. ACM*, 52(6), June 2009.

[55] J. Jackson. Nasdaq's facebook glitch came from race conditions. http://www.cio.com/article/706796/.

[56] R. Jhala and R. Majumdar. Software model checking. *Computing Surveys*, 41(4), 2009.

[57] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[58] P. Joshi and K. Sen. Predictive typestate checking of multithreaded java programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 288–296, Washington, DC, USA, 2008. IEEE Computer Society.

[59] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[60] N. Kidd, P. Lammich, T. Touilli, and T. Reps. A static technique for checking for multiple-variable data races. *Software Tools for Technology Transfer*, 2010.

[61] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Trans. Program. Lang. Syst.*, 28(5), Sept. 2006.

[62] S. T. King, G. W. Dunlap, and P. M. Chen. Operating systems with time-traveling virtual machines. In *Usenix*, 2005.

[63] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with ddt. In *USENIX*, 2010.

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

[65] J.-C. Laprie. Dependable computing: concepts, limits, challenges. In *Proceedings of the Twenty-Fifth international conference on Fault-tolerant computing*, FTCS'95, pages 42–54, Washington, DC, USA, 1995. IEEE Computer Society.

[66] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[67] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *ASPLOS*, 2010.

[68] I. Lee and R. K. Iyer. Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system. *IEEE*, pages 20–29, 1993.

[69] Z. Letko, T. Vojnar, and B. Křena. AtomRace: data race and atomicity violation detector and healer. In *PADTAD*, 2008.

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

[71] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *HPCA*, 2007.

[72] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

[73] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, 2007.

[74] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[75] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[76] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[77] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *Proceedings of the eighteenth international conference on Architectural*

support for programming languages and operating systems, ASPLOS '13, pages 39–50, New York, NY, USA, 2013. ACM.

[78] B. Lucia, L. Ceze, and K. Strauss. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA*, 2010.

[79] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[80] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[81] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.-M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Trans. Comput. Syst.*, 11(4), Nov. 1993.

[82] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989.

[83] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. MIT-CSAIL-TR-2010-038.

[84] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 446–455, New York, NY, USA, 2007. ACM.

[85] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[86] MySQL. Mysql 5.6 reference manual. http://dev.mysql.com/doc/refman/5.6/en/.

[87] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. Revivei/o: Efficient handling of i/o in highlyavailable rollback-recovery servers. In *In HPCA*, pages 203–214, 2006.

[88] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS*, 2006.

[89] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *PLDI*, 2007.

[90] National Institute of Standards and Technlogy (NIST), Department of Commerce. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002-10, June 2002.

[91] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[92] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '91, pages 133–144, New York, NY, USA, 1991. ACM.

[93] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[94] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[95] OESD. The software industry adds more than $260 billion in value to the us economy. http://www.oecd.org/unitedstates/.

[96] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[97] M. Olszewski, J. Ansel, and S. P. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[98] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.

[99] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their finding places. In *ASPLOS*, 2009.

[100] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 245–254, New York, NY, USA, 2010. ACM.

[101] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[102] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions. http://www.pcworld.com/businesscenter/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[103] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, 2004.

[104] S. Qi, N. Otsuki, L. O. Nogueira, A. Muzahid, and J. Torrellas. Pacman: Tolerating asymmetric data races with unintrusive hardware. In *HPCA*, 2012.

[105] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies c a safe method to survive software failures. In *SOSP*, 2005.

[106] S. K. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS*, 2009.

[107] G. Ramalingam and K. Vaswani. Fault tolerance via idempotence. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 249–262, New York, NY, USA, 2013. ACM.

[108] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. G. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPOPP*, 2009.

[109] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *WDDD*, 2009.

[110] C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, 2009.

[111] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *TOCS*, 1997.

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

[113] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[114] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FSE*, 2006.

[115] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, 2005.

[116] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition?: DefUse: definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[117] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS*, 2009.

[118] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, 2008.

[119] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Automatic atomic region identification in shared memory SPMD programs. In *OOPSLA*, 2010.

[120] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM.

[121] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[122] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, 2011.

[123] H. Volos, A. J. Tack, M. M. Swift, and S. Lu. Applying transactional memory to concurrency bugs. In *ASPLOS*, 2012.

[124] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, 2008.

[125] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*, 2010.

[126] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the positive: atomicity inference and enforcement using correct executions. In *OOPSLA*, 2011.

[127] M. Weiser. Program slicing. In *IEEE Transactions on Software Engineering*, 1984.

[128] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[129] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI*, 2010.

[130] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[131] J. Yi, C. Sadowski, and C. Flanagan. Sidetrack: generalizing dynamic atomicity analysis. In *PADTAD*, 2009.

[132] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[133] H. Yousuf. Sec oks nasdaq's $62 million facebook payout. http://buzz.money.cnn.com/2013/03/25/sec-nasdaq-facebook-compensation/?iid=EL.

[134] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[135] J. Yu and S. Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *MICRO*, 2010.

[136] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[137] Z. Li et. al. Have things changed now? – an empirical study of bug characteristics in modern open source software. In *ASID workshop in ASPLOS*, 2006.

[138] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys*, 2010.

[139] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[140] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, 2010.