ConMem: Detecting Crash-Triggering Concurrency Bugs through an Effect-Oriented Approach

Wei Zhang[1]
Chong Sun[1]
Junghee Lim[1]
Shan Lu[1]
Thomas Reps[1,2]
[1]Computer Sciences Department, University of Wisconsin– Madison
[2]GrammaTech,Inc
{wzh,chong,junghee,shanlu,reps}@cs.wisc.edu

Multicore technology is making concurrent programs increasingly pervasive. Unfortunately, it is difficult to deliver reliable concurrent programs, because of the huge and non-deterministic interleaving space. In reality, without the resources to thoroughly check the interleaving space, critical concurrency bugs can slip into production versions and cause failures in the field. Approaches to making the best use of the limited resources and exposing severe concurrency bugs before software release would be desirable.

Unlike previous work that focuses on bugs caused by specific interleavings (e.g., races and atomicity violations), this paper targets concurrency bugs that result in one type of severe effect: program crashes. Our study of the error-propagation process of real-world concurrency bugs reveals a common pattern (50% in our non-deadlock concurrency bug set) that is highly correlated with program crashes. We call this pattern concurrency-memory bugs: buggy interleavings directly cause memory bugs (NULL-pointer-dereferences, dangling-pointers, buffer-overflows, uninitialized-reads) on shared memory objects.

Guided by this study, we built ConMem to monitor program execution, analyze memory accesses and synchronizations, and predictively detect these common and severe concurrency-memory bugs. We also built a validator,ConMem-v, to automatically prune false positives by enforcing potential bug-triggering interleavings.

We evaluated ConMem using 7 open-source programs with 10 real-world concurrency bugs. ConMem detects more tested bugs (9 out of 10 bugs) than a lock-set-based race detector and an unserializable-interleaving detector, which detect 4 and 6 bugs, respectively, with a false-positive rate about one tenth of the compared tools. ConMem-v further prunes out all the false positives. ConMem has reasonable overhead suitable for development usage.

Categories and Subject Descriptors: D.2.5 [**Testing and Debugging**]: Testing Tools

General Terms: Languages, Reliability

Additional Key Words and Phrases: Software testing, concurrency bugs

1. INTRODUCTION

1.1. Motivation

Multicore technology is making concurrent programs increasingly pervasive. Unfortunately, concurrent programs are prone to bugs. To exacerbate the problem, concurrency bugs are particularly

An earlier version of this work [Zhang et al. 2010] appeared in the Proceedings of the 15th International Conference on Architecture Support for Programming Language and Operating Systems (ASPLOS'10).

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

difficult to detect and diagnose due to their non-deterministic behavior. Concurrency bugs can cause severe software failures and real-world disasters, such as the Northeastern Blackout of 2003 [SecurityFocus]. As concurrent programs grow increasingly popular, developing effective approaches to detecting concurrency bugs is vital.

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 only manifest 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) [Netzer and Miller 1991; Savage et al. 1997; Yu et al. 2005; Flanagan and Freund 2009], atomicity violations (an interleaving that makes certain code regions unserializable) [Flanagan and Freund 2004; Xu et al. 2005; Lu et al. 2006; Chen et al. 2008; Flanagan et al. 2008; Sadowski et al. 2009], and order violations (an execution that flips the expected order between two operations from two threads) [Lucia and Ceze 2009; Shi et al. 2010].

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

First, a high false-positive rate could cause programmers to give up on using a tool. Previous research [Narayanasamy et al. 2007; Burnim and Sen 2009] observes that only approximately 2–10% of *real* data races are harmful; a similar trend is also seen among unserializable interleavings [Park et al. 09 a].

Second, not all bugs represent equally harmful end effects, yet the different effects of different bugs are not considered during existing bug-detection processes. Table I illustrates this trend by breaking down the relationship between faults (i.e., buggy interleaving patterns) and failures in 70 real concurrency bugs that have been reported and fixed in open-source software. (Section 3 will explain how we get this data.)

	Crash	Hang	Wrong Output and Other Misbehaviors
Atomicity Violation	26	3	19
Order Violation	11	3	6
Other	0	1	1

Table I: Types of failures vs. types of faults (Note: The above data comes from *fixed* bugs in open-source software. Therefore, bugs that lead to minor or benign effects are under-represented in this table.)

Figure 1 depicts the limitations of previous work (and our opportunities) by projecting a concurrent program's interleavings onto a two-dimensional space. The x-axis and y-axis represent different *effects* and different patterns of interleavings (i.e., *failures* and *faults* for buggy interleavings), respectively. 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. This difficulty is reflected in every column in Table I: no single interleaving pattern can capture one type of failure.

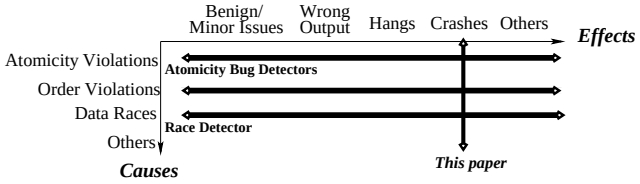


Fig. 1: A conceptual two-dimensional depiction of approaches to finding flaws in concurrent programs

Second, a large number of **false positives**. This is observed in the real world [Burnim and Sen 2009; Narayanasamy et al. 2007; Park et al. 09 a], and is reflected in Figure 1, where each horizontal stripe inevitably covers interleavings with benign effects.

Third, a lack of **severity differentiation**. *Severity* is a qualitative metric for software failures. In practice [Apache Bugzilla ; Bugzilla@Mozilla], bugs that lead to "*crashes and loss of data*" are considered to have high *severity*, and bugs that only lead to "*minor loss of function or cosmetic problems*" are considered *minor* or *trivial*. Without considering different effects of bugs, as shown in each row of Table I, the horizontal approach cannot focus on severe bugs.

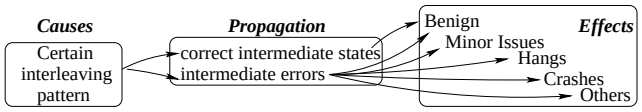


Fig. 2: The cause–effect chain

We can deepen our understanding of the false positive and severity issues by looking at the cause–effect chains in concurrent programs. As shown in Figure 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 [Narayanasamy et al. 2007]), 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 [Yu et al. 2005]). A pair of concrete examples is shown in Figure 3. These two real-world bugs start with similar bug-triggering interleavings, both involving data races and unserializable interleavings. However, one causes a server crash, while the other has an almost invisible effect at the end.

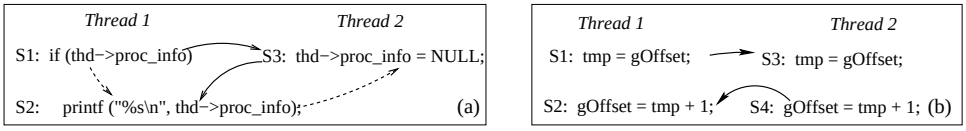


Fig. 3: (a) A severe real-world concurrency bug from MySQL database server. (MySQL execution usually follows the dotted line, but it crashes when its interleaving follows the solid line.)
(b) A non-critical concurrency bug that existed in Mozilla for years without any complaint. (`gOffset` holds browsing statistics. Throughout Mozilla, it is read only once in a statistics-printing function.)

The false-positive issue has already caught the attention of many researchers. Various innovative approaches, such as training [Lu et al. 2006], automated testing [Park et al. 09 a; Sen 2008] 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.

The severity issue has not received the attention it deserves in concurrency-bug -detection research. Severity guidance is important for concurrent programs due to several reasons: (1) in general, developers use severity to prioritize their diagnosis and repair efforts [Apache Bugzilla ; Bugzilla@Mozilla]. This is also observed by a recent study of Linux kernel developers' reactions to static bug-detection reports [Guo and Engler 2009]; (2) the huge interleaving space makes the prioritization process extremely important; (3) non-determinism makes minor-impact concurrency bugs more trivial than their sequential counterparts; and (4) fixing concurrency bugs often results in performance penalties, which make developers more reluctant to fix inconsequential concurrency bugs. In reality, programmers are even willing to introduce new non-severe concurrency bugs in order to fix severe concurrency bugs [Mozilla Developers].

In summary, this paper presents a bug-detection approach that focuses on certain *vertical stripes* of the interleaving space — specifically, the *crash* stripe 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.

1.2. Contributions

This paper proposes a concurrency bug-detection-tool, ConMem, which is guided not by certain interleaving patterns, but by one important class of bug effects, namely, program crashes. By doing so, we circumvent the problem of detecting the complicated root causes. We essentially mitigate the problem by identifying the most common patterns of program crashes and let ConMem go after each. As a dynamic monitoring tool, ConMem **accurately and predictively detects severe concurrency bugs that can lead to program crashes, no matter which interleaving pattern (race, atomicity violation, order violation, etc.) is the cause**.

To capture the crash stripe in the interleaving space, we need to look backward along the cause–effect chain and find a pattern that can predict crashes.

Our characteristics study of the *cause-effect chains* of 70 real-world concurrency bugs in Section 3 reveals an error-propagation pattern that is common and highly correlated with software crashes. This pattern covers almost half of the examined bugs and constitutes more than 85% of crash-inducing bugs. It occurs when unsynchronized memory operations *directly* lead to memory errors, including NULL-shared-pointer dereferences, dangling pointers to shared memory, un-initialized shared-memory reads, and shared-buffer overflows. We refer to this pattern as *concurrency-memory errors*.

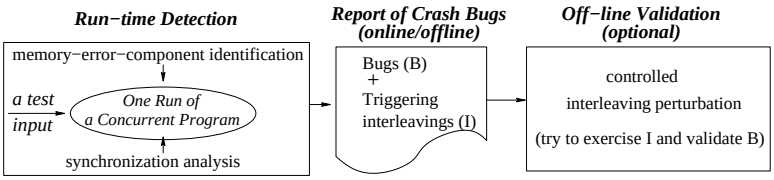


Fig. 4: The flow and components of ConMem

Based on the above observation, ConMem (Figure 4) is designed to predictively catch concurrency-memory errors and to report fatal interleavings before they occur. Under each test input, ConMem monitors one run of the test concurrent program. It uses run-time information to first identify ingredients of potential concurrency-memory errors (e.g., a NULL-assignment and

a dereference of a shared pointer from different threads are ingredients of a concurrency-NULL-dereference bug). It then analyzes synchronizations around these suspect ingredients to decide whether fatal interleavings exist that trigger a concurrency-memory error.

Furthermore, we built an active-testing tool ConMem-v to automatically validate whether the fatal interleavings reported by ConMem can truly occur. Through ConMem-v, developers can easily validate a ConMem report and reliably repeat true bugs.

Overall, this paper makes the following contributions:

First, the paper reports the first characteristic study on the cause-effect chains of real-world concurrency bugs. Our study is based on 70 fixed, real-world concurrency bugs collected by a previous study [Lu et al. 2008] from four widely-used C/C++ applications (Apache, Mozilla, MySQL, and OpenOffice). The study reveals several interesting findings: (1) concurrency bugs that can cause program crashes are common among fixed bugs, constituting approximately 50% of non-deadlock bugs in the study; (2) interleaving patterns have little correlation with bug severity; (3) most (about 85%) examined crash concurrency bugs share one error-propagation pattern: the buggy interleavings directly cause memory bugs on shared-memory objects; and (4) above concurrency-memory errors can be further classified into four types: NULL-shared-pointer dereferences, shared-buffer-overflows, uninitialized reads to shared variables, and dangling pointers to shared memory.

Second, the paper introduces a new perspective on checking the huge interleaving space. Traditional bug detectors focus on the *cause* of concurrency bugs and work horizontally in the interleaving space shown in Figure 1. ConMem complements them by focusing on certain *effects* and working vertically. Specifically, traditional tools identify all instances of certain interleaving patterns and rely on testing, training, or manual inspection to determine which can truly cause (severe) failures. ConMem benefits from its effect-oriented vertical approach and effectively prioritizes its bug-detection effort towards severe software bugs, instead of benign or trivial interleaving problems.

Third, the paper provides a bridge between the well-studied memory-bug problem and the challenging concurrency-bug issue. Memory-bug-detection techniques are already mature for sequential programs [Coverity ; Hastings and Joyce 1992; Nethercote and Seward 2007]. However, they are not as effective in concurrent programs for several reasons. First of all, dynamic memory-bug detectors are sensitive to interleaving. They can only catch bugs when they occur, unable to predict what might happen under future interleavings. Furthermore, even for those bugs that do occur during the monitored run, memory-bug detectors cannot identify the root cause(i.e., buggy interleaving) and cannot help developers fully understand and fix the bug. Static analysis is not sensitive to interleaving. However, even with recent inspiring progress [Chugh et al. 2008], its scalability and effectiveness in concurrent programs are still limited by the fundamental pointer-alias and concurrency-analysis problems. ConMem combines classic memory-bug-detection techniques with predictive interleaving analysis and interleaving testing, thus solving the above problems (more discussion is in Section 8).

Fourth, the paper describes a tool, ConMem, that effectively detects severe concurrency bugs and validates the results through controlled testing. ConMem is implemented using binary instrumentation. By design, ConMem has several advantages: (1) it uses predictive bug detection, and thus is less sensitive to interleaving; (2) it has no training requirement; (3) it reports easy-to-validate bug-detection results (i.e., memory errors), with no need for manually written oracles to judge execution correctness; (4) it has high accuracy and coverage on severe concurrency bugs; and (5) it supports a simplified diagnosis process via ConMem-v. In fact, the co-design of ConMem and ConMem-v also helped to simplify some detection algorithms in ConMem without causing more false positives to be reported to developers.

ConMem is evaluated on 7 open-source programs with 10 real-world concurrency bugs, 9 of which can cause programs to crash. These programs include three server applications (Apache HTTP server, MySQL database server, and Cherokee HTTP server), three client/utility applications (Mozilla, Transmission, and PBZIP2), and one scientific application from SPLASH2 [Woo et al. 1995].

Our results show that ConMem can effectively detect 9 out of 10 tested concurrency bugs, which is better than both a race-based detector (4 out of 10) and an atomicity-violation based detector (6 out of 10) to which ConMem is compared. Furthermore, ConMem detector's false-positive rate is about one tenth of the race-based and atomicity-violation-based detectors. ConMem-v further prunes all false positives without introducing false negatives. ConMem detection's run-time overhead is comparable to previous software bug-detection tools and is suitable for in-house bug detection. Each ConMem detector introduces 2–16 times slowdown for client software and the SPLASH2 benchmark, and 3–29% overhead for server applications.

ConMem is also evaluated on an open-source program, the *Click* modular router [Click 2010], for which no concurrency bugs were previously known. Using the standard test suite released together with *Click*, ConMem detects 2 previously unknown concurrency bugs that could lead to software crashes. This result further demonstrates ConMem's capability to expose previously unknown concurrency bugs during in-house testing.

The remainder of the paper is organized as follows: background and our cause-effect characteristics study are presented in Sections 2 and 3, respectively. Section 4 discusses ConMem's bug-detection method. The ConMem validator is presented in Section 5. Section 6 and Section 7 present evaluation methodology and experimental results. Finally, related work is presented in Section 8.

2. BACKGROUND

Memory bugs are very common and also severe [Sullivan and Chillarege 1992; Z. Li et. al. 2006]. Many of them can cause program crashes, data loss, and even security problems. This section provides a brief review of memory bugs.

2.1. Typical Memory Bugs

NULL pointer dereferences happen when the program dereferences a NULL-valued pointer. It causes the program to immediately crash. Much work has been done on static detection of NULL-pointer dereferences. However, their accuracy and scalability is limited by pointer-alias problems.

Un-initialized reads occur when a valid memory location is read before it is properly initialized. It could cause incorrect output or a crash. Dynamically detecting un-initialized reads is straightforward. In practice, sophisticated memory detectors, like Valgrind [Nethercote and Seward 2007], also consider the context of the un-initialized read, and only report bugs when the un-initialized value is used in a critical scenario.

Accesses to invalid memory locations include dangling-pointer bugs (accessing memory locations that are already freed), buffer-overflow bugs (accessing memory locations that are beyond the buffer's boundary), and stack smashing (overwriting critical data stored on the stack). These bugs can cause not only incorrect outputs, but also crashes and security vulnerabilities.

Other memory bugs include double-free bugs (a memory location is freed twice), memory-leak bugs, and complicated bugs, such as accessing legitimate but incorrect memory locations. Various algorithms have been proposed to detect such bugs [Jones and Kelly 1997; Ruwase and Lam 2004].

2.2. Memory Bugs in Concurrent Programs

Memory bugs in concurrent programs can be classified into two types. The first type only involves one thread and can be deterministically triggered by special inputs. In terms of dynamic detection, testing, and diagnosis, this type of bugs is no different from those in sequential programs.

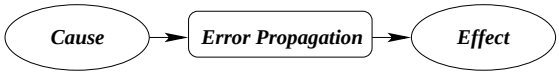
The second type, such as the one shown in Figure 3 (a), is more complicated. They involve more than one thread and require not only special inputs but also special interleavings to occur. These bugs are actually side-effects of more fundamental concurrency bugs. As discussed in Section 1, these bugs cannot be addressed by existing dynamic-memory bug detectors because their existence under future interleavings cannot be predicted by existing dynamic detectors. Even when they do occur under the current interleaving, their root causes still cannot be correctly identified.

Categories	Description
Con-Memory Errors*	Wrong execution order among shared memory operations directly transit to memory bugs
Buffer Overflow	Conflicting accesses to shared buffer and buffer index/boundary variables cause buffer overflow.
Dangling Pointer	A thread deallocates a shared buffer before another thread accesses it (Figure 8).
NULL Pointer	A thread dereferences a shared pointer that is assigned NULL by another thread (Figure 3(a)).
Uninitialized Read	A thread reads a shared variable before the variable is initialized by another thread (Figure 7).
Con-Semantic Errors	Interleaving causes unexpected variable values and program states.

Table II: Categorization of intermediate errors directly caused by buggy interleavings (*:memory bugs such as double-frees and memory-leaks are unlikely to happen as direct effects of buggy interleavings).

3. CAUSE-EFFECT CHARACTERISTICS

Before describing ConMem, we first present a study of the error-propagation chains of 70 real-world concurrency bugs. This study will help us understand how buggy interleavings gradually affect the program state and ultimately cause various software failures, especially those that are severe (e.g., crashes).



Buggy Interleavings Intermediate Errors Visible Software Failures

Fig. 5: Cause-effect chain

3.1. Methodology and Caveat

Bug Source This study uses a set of 70 real-world non-deadlock concurrency bugs collected in [Lu et al. 2008]. [1] 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.

Characteristics in study Previous characteristics studies [Farchi et al. 2003; Lu et al. 2008] focus primarily on the interleaving patterns that cause the concurrency bugs. This work will study the error-propagation process from its cause (buggy interleavings), through intermediate errors, to the final effects (demonstrated by Figure 5).

In terms of *causes*, we refer to previous work [Lu et al. 2008] to consider two causes: atomicity-violation and order-violation. Data races are orthogonal to these two and are not separately considered here.

In terms of *effects*, we follow previous general bug-characteristics studies [Z. Li et. al. 2006; Sullivan and Chillarege 1992] and consider three main effects: crashes, hangs, and minor wrong functionality issues (including wrong outputs). Strictly speaking, there could also be severe bugs like loss of data, but the bug set we use does not contain such examples.

The most difficult part of our categorization is the *intermediate errors*. Since there has been no previous study regarding this, based on our own observations and inspiration from studies of general software bugs, we classify intermediate errors into two major categories: *intermediate memory errors* and *intermediate semantic errors*.

An *intermediate memory error* occurs when the buggy interleaving changes the execution order of a set of shared memory operations so that these operations themselves *directly* instantiate a memory bug. Afterward, the program fails similarly to those caused by memory bugs in sequential programs.

[1] The original list in [Lu et al. 2008] includes 74 bugs. 4 of them do not have enough error-propagation information and are discarded in this study.

This paper refers to such error chains as *concurrency-memory errors*. They are further classified based on which types of memory bugs are instantiated, as shown in Table II.

An *intermediate semantic error* occurs when the buggy interleaving causes new and unexpected program states that are not handled by the program. Once that unexpected state happens, the program fails, as happens with semantic bugs in sequential programs.

These two categories are usually easy to classify, except for a few complicated cases, such as the Mozilla bug shown in Figure 6. This bug and the MySQL bug in Figure 3(a) both result in NULL-pointer dereferences followed by a crash. However, they have different error-propagation processes. In the MySQL example, the NULL pointer is a shared variable, and the NULL-pointer dereference is a *direct* result of the buggy interleaving. However, in the Mozilla bug, the NULL-assignment (*S*2) and NULL-dereference (*S*3) both occur in one thread as a result of an unexpected {`id`,`key1`} pair caused by the buggy interleaving. Our principle is to categorize errors based on the *direct* impact of interleavings. Therefore, Figure 6 is considered a concurrency-semantic error.

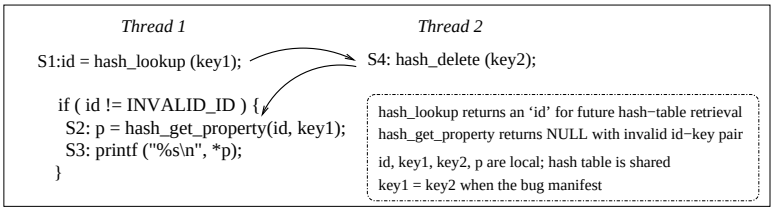


Fig. 6: A complicated concurrency bug with an intermediate semantic error (simplified from a real Mozilla bug). The buggy interleaving causes an (unexpectedly) invalid {`id`,`key1`} pair, which causes `hash_get_property` to return NULL.

Caveats We attempted to the best of our ability to use representative bugs and correctly classify them. We do not intend to draw general conclusions for all bugs and all applications. We only plan to use those trends that are consistent throughout our bug set to guide effect-oriented concurrency-bug detection. We warn readers to interpret the findings below with the methodology in mind. Because this study focuses on C/C++ programs, the cause-effect characteristics may not apply to other types of programs, e.g., Java programs. Of course, because many multi-threaded programs, especially client/server programs, are still written in C/C++, we believe our study is representative of a large class of important applications.

3.2. Results and Implications

Many interesting results were revealed in this study. Here we limit ourselves to findings that are closely related to the design of effect-oriented concurrency bug-detection tools.

	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 III: Effects (failure types) of concurrency bugs

Finding 1 *Approximately 50% of the studied non-deadlock bugs can cause program crashes*, as shown in Table III. This indicates that crash concurrency bugs are not only severe, but also common among those reported-and-fixed bugs. Detecting them is crucial.

Finding 2 *There is no correlation between the cause and the effect of a concurrency bug.* A breakdown between the types of interleaving patterns (causes) and the types of failures (effects) is presented in Table I. As discussed in Section 1, it is difficult to predict the final effect or severity of a concurrency bug based on its root cause interleaving pattern.

	Crash	Hang	Minor Func. Issues
Con-Memory err.	**31**	0	3
Con-Semantic err.	6	7	23

Table IV: Types of failures vs. types of intermediate errors

Finding 3 *Approximately 84% (31 out of 37) of the studied concurrency bugs that cause crashes have concurrency-memory error patterns*, as shown in Table IV. The few exceptions are similar to the Mozilla bug shown in Figure 6. This finding provides a promising avenue for tool builders: by focusing on the concurrency-memory error pattern, we can handle most severe concurrency bugs that can cause program crashes (at least in C/C++ programs).

Finding 4 *Approximately 90% (31 out of 34) of the intermediate memory errors in our bug set cause program crashes at the end*, as shown in Table IV. This finding is consistent with the trend in sequential programs [Z. Li et. al. 2006]. It further demonstrates that by targeting concurrency-memory errors, we can effectively focus the bug-detection and testing effort upon severe concurrency bugs, without wasting resources on benign or non-critical interleaving problems.

	Memory Errors				Semantic
	NULL	UnInit	Dangling	Overflow	Errors
Mozilla	9	0	8	4	19
MySQL	3	1	1	0	10
Apache	2	0	3	1	4
OpenOffi	1	1	0	0	3
ALL	15	2	12	5	36

Table V: Breakdown of intermediate errors

Finding 5 *Concurrency-memory errors include four common patterns.* As we can see in Table V, all the concurrency-memory errors in our study fall into four well-defined categories: NULL-pointer dereferences, dangling-pointers, buffer-overflows, and uninitialized-reads. For simplicity, we will refer to these four sub-types of concurrency-memory errors as follows: **Con-NULL** (NULL-pointer dereference directly caused by a buggy interleaving), **Con-UnInit** (uninitialized read directly caused by a buggy interleaving), **Con-Dangling** (dangling pointer directly caused by a buggy interleaving), and **Con-Overflow** (buffer overflow directly caused by a buggy interleaving). By "directly caused", we mean a memory error is caused by unsynchronized shared-memory accesses, such as the bug shown in Figure 3 (a), instead of local-memory accesses that have data/control dependence with unsynchronized shared-memory accesses, such as the bug shown in Figure 6.

These regular bug patterns provide clear guidance to an effort directed at concurrency-bug detection; we will show that, by focusing on these four types of bugs, we can build a bug-detection tool that finds serious concurrency errors with a low false-positive rate. This is precisely the approach followed by ConMem.

4. DETECTING SEVERE CONCURRENCY BUGS

4.1. Overview

ConMem includes four dynamic bug-detection modules that are responsible for detecting Con-NULL, Con-UnInit, Con-Dangling, and Con-Overflow bugs, respectively. The design of ConMem is guided by our characteristics study, and follows three principles:

	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) *a*1: from t1, accesses *v* (2) *a*2: from t2, accesses *v*	Data race between *a*1 and *a*2 (approximated condition)	Yes	Yes

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

(1) Effect-oriented, 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.

(2) 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.

(3) Balance between analysis accuracy and complexity. Because the validator ConMem-v can help prune out false positives, ConMem has the luxury of trading accuracy for simplicity, when necessary.

Following these principles, 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 VI. The following sub-sections will elaborate on how to detect each sub-type of concurrency-memory error.

4.2. Con-NULL Detection

4.2.1. 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 (a). As we can see there, *S*2 from thread 1 dereferences a shared pointer variable `thd`→`proc_info`, and *S*3 from thread 1 assigns NULL to the same variable. Under a buggy interleaving, *S*3 executes right between *S*1 and *S*2, immediately causing a NULL-pointer dereference and a MySQL crash. Of course, the above buggy interleaving occurs only rarely, and MySQL mostly behaves correctly.

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.

4.2.2. 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, [2] 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 [Netzer and Miller 1991; Park et al. 09 a], 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 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 VIII.

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 Section 7.

4.2.3. Implementation. ConMem implements the above algorithm using run-time recording (with PIN [Luk et al. 2005] 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

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

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 [Lamport 1978]. 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. Under 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 [Savage et al. 1997]. 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. Section 4.6 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 Section 7.

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 dif-

ferent 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 [Savage et al. 1997; Flanagan and Freund 2004; Chen et al. 2008; Joshi and Sen 2008; Yi et al. 2009; Park et al. 09 a]. 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 [Sen and Agha 2006].

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 [Park et al. 09 b]. Based on our experience, it is rarely a problem for Con-NULL, as shown in Section 7. 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.

4.3. Con-UnInit Detection

Thread 1	Thread 2
h = malloc();	
/* h is shared; S1 is expected to initialize h−>band */	
S1 h−>band = tr_bandNew(h);	
	S2 assert(is_band(h−>band));

Fig. 7: A concurrency bug that leads to an undefined read and finally causes crash (from Transmission-1.42)

4.3.1. What is a Con-UnInit bug?. Con-UnInit bugs are un-initialized memory reads directly caused by buggy interleavings. An example of a Con-UnInit bug is shown in Figure 7. In this example, a shared variable `h→ bandwidth` is initialized at *S*1 in thread 1. Read accesses to this variable are supposed to occur after *S*1. Unfortunately, without proper synchronization, *S*2 in thread 2 can execute before *S*1 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., *S*2 in Figure 7), 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.

4.3.2. 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 *v* occurs, *Initializer* is checked

to determine whether this is the first write to *v* 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 *v*. 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 [Lamport 1978]. 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 *v* 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 [Nethercote and Seward 2007], 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 [Lu et al. 2006]. It will *not* increase with longer executions, as long as the program's active memory consumption does not change.

4.4. Con-Dangling Detection

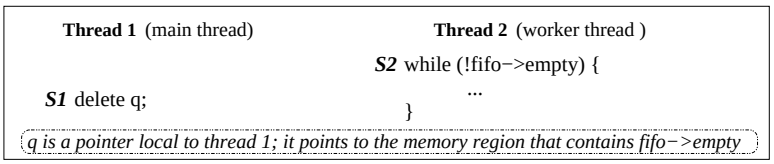


Fig. 8: A concurrency bug that leads to a dangling pointer and finally causes crash (from PBZIP2-0.9.4)

4.4.1. What is a Con-Dangling bug?. A Con-Dangling bug occurs when buggy interleavings directly cause dangling pointer accesses. Figure 8 demonstrates a bug from PBZIP2. In this example, pointer `q` (a local variable in 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 *S*2 when it is already deleted by thread 1 at *S*1, 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.

4.4.2. 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.

4.5. Con-Overflow Detection

4.5.1. 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.

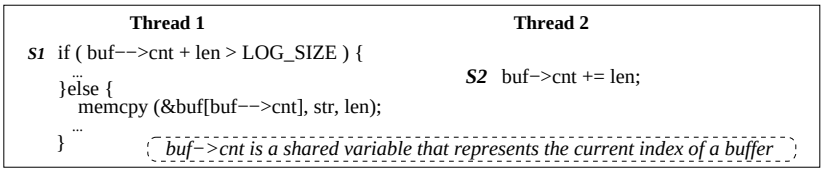


Fig. 9: A concurrency bug that can lead to a buffer overflow and subsequent crash (from Apache-2.0.45)

Figure 9 shows an example of a typical Con-Overflow bug. Thread 1 conducts a sanity check at *S*1 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 [Cadar et al. 2008] 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 Overflow-Con bugs, and we rely on our

ConMem-validator (Section 5) to prune out false positives. We leave the more general Con-overflow detection problem to future work.

4.5.2. 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 [Savage et al. 1997]. 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.

4.6. Handling Spin-Loop Synchronizations

As discussed in Sections 4.2 and 4.3, a major source of false positives in ConMem is custom-synchronization operations, as demonstrated by Figure 10(a). This subsection discusses 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 Section 7.2.

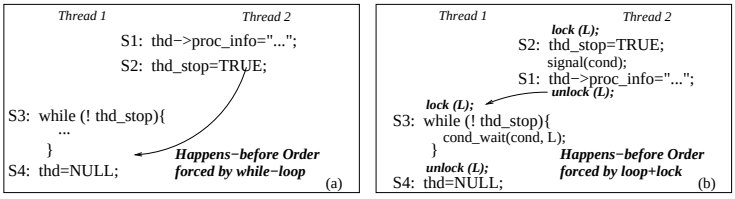


Fig. 10: Examples of spin-loop synchronization (thd_stop is a volatile variable). (a) A NULL-pointer dereference can never occur between S4 and S1, because thread 1 cannot execute S4 until its S3-loop is terminated by S2 in thread 2. (b) Synchronization is achieved by a spin loop **and** locks. Without 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.

4.6.1. Spin-loop identification. This analysis algorithm is inspired by SyncFinder [Xiong et al. 2010], and involves two steps.

First, *identifying loops.* This step is conducted through CodeSurfer/x86 [Balakrishnan et al. 2005], 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 [Bourdoncle 1993], 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 r for each loop L — a potential loop-exit read that obtains the same value from a variable v in all but the last iteration of L. We then identify the write w that defines the value read by r in the last loop iteration. L is considered to be a synchronization loop if w always comes from a different thread than r. In that case, w, such as S2 in Figure 10(a), is marked as a synchronization write, and the loop-exit read, such as S3 in Figure 10(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 [Tian et al. 2008; Chen et al. 2008; Xiong et al. 2010]. Our approach is inspired by SyncFinder [Xiong et al. 2010]. 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 [Tian et al. 2008; Chen et al. 2008; Xiong et al. 2010], 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.

4.6.2. Integrating synchronization-loops into ConMem. A synchronization loop is one type of 'order synchronization' discussed in Table VI — 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 10.

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 10(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 Section 7.2.

5. BUG EXPOSING AND VALIDATION

The design of ConMem-v is inspired by previous tools that validate data-race [Park and Sen 2008] and atomicity-violation bug reports [Park et al. 09 a]. 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 [Luk et al. 2005] 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 11 (consider A as `wp`, B as `rp`).

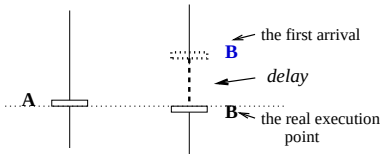


Fig. 11: Illustration of how ConMem-v perturbs execution

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 11). 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* pretty much 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 11 (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 [Nethercote and Seward 2007; Hastings and Joyce 1992].

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 [Musuvathi et al. 2008; Sen 2008]; the other is to insert artificial delays [Park et al. 09 a; Edelstein et al. 2002]. 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.

6. 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) [Woo et al. 1995].

Apart from these 7 applications, ConMem is also evaluated on the latest version of a multi-threaded software system, Click [Click 2010], 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 Section 7.5.

Bugs in evaluation For evaluation, we use 10 real-world concurrency bugs[3] 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 representative, covering different types of faults and error-propagation patterns, as shown in Table VII. 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.

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 [Luk et al. 2005] binary instrumentation framework for all our tools. We use Valgrind–Helgrind [Nethercote and Seward 2007] as the race-detection front-end for Con-Overflow.

[3]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.

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

Our experiments use bug-triggering inputs reported by the user, like previous dynamic concurrency-bug detectors [Xu et al. 2005; Lu et al. 2006]. 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 [Park et al. 09 a; Musuvathi et al. 2008], 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 (Section 4.6) is **not** applied. We evaluate how that algorithm further improves the accuracy of ConMem in Section 7.

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 [Dinning and Schonberg 1991; O'Callahan and Choi 2003], originally implemented in the widely-used Valgrind-Helgrind detector [Nethercote and Seward 2007] and slightly modified by us for better race coverage. *Atom* was implemented by us based on an algorithm described in previous work [Park et al. 09 a]. 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 [Lu et al. 2008; Vaziri et al. 2006; Lu et al. 2006]). There are other race and atomicity bug detectors, such as happens-before race detectors [Netzer and Miller 1991] and training-based atomicity detectors [Lu et al. 2006]. We did not choose them, because their training requirement or interleaving-sensitive design makes for an apples-to-oranges comparison.

7. EXPERIMENTAL RESULTS

7.1. Overall Results

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

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

[4]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 VIII: Bug-detection results (Key: ✓– bug was detected; × – bug not detected.)

App.	# ShrMem Inst		Races	Atom.	Con-Null	Con-Dangling	Con-UnInit	Con-Ovfl	ConMem Total
	Static	Dynamic	#FP:#Bug	#FP:#Bug	#FP:#Bug	#FP:#Bug	#FP:#Bug	#FP:#Bug	#FP:#Bug
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			**369:11**	**458:15**	16:8	10:5	19:5	0:2	**45:20**

Table IX: Bug reports and false positives before ConMem-v pruning (Note: 1. the bug report number here is larger than that in Table VIII, because some bug reports share one root cause. There are 9 distinct root causes of these 20 bug reports. 2. #FP: # of false positives; #Bug: # 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 X)

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.

ConMem still misses one severe bug in Mozilla. This is a complicated concurrency bug that requires more than one rare timing condition to manifest. 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 is 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 severe 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.

7.2. False-Positive Results

Before automated pruning

Table IX 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* [Park et al. 09 a], 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 [5]. Since some bug reports in Table IX share the same root cause, the total number of true bug reports there is larger than that in Table VIII.

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-positive rate (about 2.5 false positives per true bug) is reasonably low considering ConMem's predictive detection capability on severe 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 IX (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 1). As discussed in Section 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 X 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 Section 4.3.

Note that, according to Table X, *almost all buggy interleavings reported by ConMem are true and severe bugs, 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 [Narayanasamy et al. 2006; Burnim and Sen 2009; Park et al. 09 a].

Pruning false positives via custom-synchronization analysis

We also evaluated the synchronization-loop analysis discussed in Section 4.6. As shown in Table X, 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. 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 with 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 IX 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 VIII and Table VII.

[5] Code regions that are problematic only under weak memory consistency models are not considered as bugs here, similar with previous work [Park et al. 09 a; Savage et al. 1997]

App.	Benign UnInit	# of F.P. caused by custom synchronization		# of F.P. pruned by Section 4.6 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 X: Causes of ConMem false positives

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 Section 4.2, the bug-detection ability of ConMem is sensitive to the code/path coverage, like all dynamic bug detectors [Savage et al. 1997; Lu et al. 2006; Nethercote and Seward 2007], 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.

7.3. Time and Space Overhead

Table XI 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 [Lu et al. 2006; Xu et al. 2005; Savage et al. 1997] 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 XI: 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.)

7.4. 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 XII 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 12(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 12(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*.

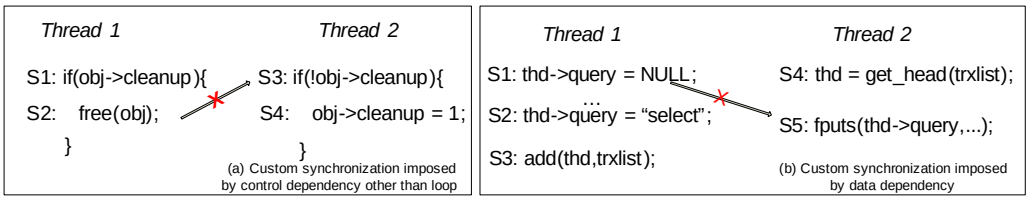


Fig. 12: Two false positive examples caused by unidentified custom synchronization

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 XII: Bug suspects pruned by synchronization analysis

7.5. 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* [Click 2010], 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.[6] 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 XIII. 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 are caused by **2** different root causes in the program. One buggy interleaving reported by Con-Dangling is demonstrated in Figure 13. 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*.

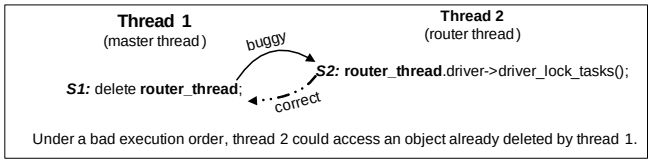


Fig. 13: A concurrency bug that leads to a dangling pointer and finally a crash (from *Click*-1.8.0)

ConMem has about a 1:1 false-positive-vs-true-bug rate for *Click*, which is consistent with the earlier experiments shown in Table IX. 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 XIII, we also tried *Races* and *Atom* on these 7 test cases. The results follow a similar trend to that in Table IX. *Races* and *Atom* cannot detect the bugs that ConMem detected. For example, the bug depicted in Figure 13 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 13 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.

[6]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.

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 XIII: *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.).

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 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 runtime 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 a lot of redundancy among the four tools. (2) Saving redundant interleaving testing among inputs that have overlapped code coverage. This is obviously more challenging, but is also promising. As discussed in connection with Table XIII, there is overlap among the concurrency bugs revealed by different inputs.
— While we have had a fairly positive experience with applying ConMem to Click, some additional features can make ConMem easier to use in the future: (1) Providing call stack information for reported bugs to ease debugging. (2) Providing information about why a bug suspect is not exposed by ConMem-v.

8. RELATED WORK

Much related work has been discussed in earlier sections. In this section, we only discuss a few that are closely related and were not discussed previously.

Empirical studies of concurrent programs Due to the lack of concurrency bug sources, only a few studies [Farchi et al. 2003; Lu et al. 2008] 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 [Rossbach et al. 2009]. Our paper complements previous studies by looking at the error-propagation process in concurrency bugs.

Concurrency bug detection, testing, and avoidance Existing concurrency-bug detectors can be categorized as performing race detection such like [Netzer and Miller 1991], Eraser [Savage et al. 1997], RaceTrack [Yu et al. 2005] and FastTrack [Flanagan and Freund 2009]; atomicity-violation detection such like Atomizer [Flanagan and Freund 2004], SVD [Xu et al. 2005], AVIO [Lu et al. 2006], jPredictor [Chen et al. 2008], Velodrome [Flanagan et al. 2008], and SingleTrack [Sadowski et al. 2009]; and multi-variable atomicity violation detection such like ColorSafe [Lucia et al. 2010]. ConMem complements existing tools by focusing on crash effects, instead of specific interleaving patterns. The predictive interleaving analysis in ConMem is inspired by previous predictive race and atomicity-violation detectors like Atomizer [Flanagan and Freund 2004] and jPredictor [Chen et al. 2008]. Many innovative approaches, such as training (AVIO [Lu et al. 2006]), noise-making (ConTest [Edelstein et al. 2002]) and active testing (RaceFuzzer [Sen 2008],CTrigger [Park et al. 09 a]), have been proposed to address the false-positive problem in concurrency-bug detection. ConMem uses synchronization analysis and perturbation-based interleaving-enforcement techniques that is similar to some of these tools like CTrigger [Park et al. 09 a]. ConMem complements those tools by considering the problem from a different perspective. It focuses on certain effects of concurrency bugs, instead of a specific interleaving pattern.

Atom-Aid [Lucia et al. 2008] and PSet [Yu and Narayanasamy 2009] extended existing dynamic bug detectors by prohibiting certain patterns of interleavings at run time by using hardware support to survive concurrency bugs during production runs. Software-only tools like Grace [Berger et al. 2009] and Kendo [Olszewski et al. 2009] achieve similar goals for certain types of multithreaded programs at runtime. ConMem complements such work by exposing concurrency bugs before they escape to production runs.

Interleaving testing tool such as CHESS [Musuvathi et al. 2008] works by systematically exploring the interleaving space. ConMem complements such work by providing a different perspective on splitting the interleaving space. Work on deterministic execution such like DMP [Devietti et al. 2009] and Kendo [Olszewski et al. 2009] also tries to solve the interleaving space challenge by limiting the number of interleavings that a program can follow.

Concurrent program analysis and model checking A lot of inspiring research has been conducted on static analysis and model checking of concurrent programs. A recent study [Chugh et al. 2008] 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 has a completely different design goal from static-analysis tools. ConMem does not aim to provide any guarantee. Actually, ConMem also does not aim to report all potential memory errors in concurrent programs. By focusing on the concurrency-memory error pattern, ConMem can use relatively simple algorithms to effectively detect severe concurrency bugs. In addition, as a dynamic bug-detection tool, ConMem naturally has 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. A lot of progress has been made [Godefroid 1996; Qadeer and Wu 2004; Flanagan and Godefroid 2005; Musuvathi et al. 2008] 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 paper will help provide heuristics that can be used in future model checkers.

General software failure diagnosis The effect-oriented approach used in ConMem shares a similar flavor with failure-diagnosis tools [Sumner and Zhang 2009; Dimitrov and Zhou 2009] that look for the root causes of observed failures through data slicing. A failure that has already occurred and been recorded is essential to these tools. Diagnosis tools are also designed to focus on concurrency bugs, such as CCI [Jin et al. 2010], Falcon [Park et al. 2010], and Recon [Lucia et al. 2011]. These tools can identify shared-memory accesses that are statistically correlated with software failures and

thus help debugging. Where ConMem differs is that it searches for unknown interleaving errors that can cause previously unobserved failures.

9. CONCLUSIONS AND FUTURE WORK

This paper proposes an effect-oriented approach to detecting severe concurrency bugs. By focusing on the concurrency-memory error-propagation pattern revealed by our characteristics study, ConMem effectively and predictively detects concurrency bugs with crash effects. In our evaluation with 10 real-world severe concurrency bugs, ConMem detects more bugs with significantly fewer false positives than race and atomicity-violation detectors. In addition, ConMem-v prunes out all false positives and provides a reliable way to expose all the true bugs reported by ConMem.

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

In the future, ConMem can be extended in the following ways. First, we could use static analysis to improve ConMem's ability to identify pointer variables and buffer-index variables. Second, we could try to identify more kinds of customized synchronization and further decrease the remaining false positives of ConMem. Finally, we could also apply the effect-oriented idea to detecting other types of severe bugs (e.g., security vulnerabilities, silent data corruption, etc.) in both C programs and Java programs.

10. ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their invaluable feedback. Shan Lu is supported by a Claire Boothe Luce faculty fellowship, and her research group is supported by NSF grant CCF-1018180 and CCF-1054616. Thomas Reps's research group is supported by NSF under grants CCF-0810053 and CCF-0904371, by ONR under grants N00014-09-1-0510 and N00014-10-M-0251, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. Thomas Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

REFERENCES

APACHE BUGZILLA. How important is the bug? http://issues.apache.org/bugwritinghelp.html.

BALAKRISHNAN, G., GRUIAN, R., REPS, T., AND TEITELBAUM, T. 2005. CodeSurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *CC*.

BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. 2009. Grace: safe multithreaded programming for c/c++. In *OOPSLA*.

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

BUGZILLA@MOZILLA. A bug's life cycle. https://bugzilla.mozilla.org/page.cgi?id=fields.html#severity.

BURNIM, J. AND SEN, K. 2009. Asserting and checking determinism for multithreaded programs. In *FSE*.

CADAR, C., DUNBAR, D., AND ENGLER, D. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*.

CHEN, F., SERBANUTA, T. F., AND ROSU, G. 2008. jpredictor: A predictive runtime analysis tool for java. In *ICSE*.

CHUGH, R., VOUNG, J. W., JHALA, R., AND LERNER, S. 2008. Dataflow analysis for concurrent programs using datarace detection. In *PLDI*.

CLICK. 2010. The Click Modular Router Projec. http://read.cs.ucla.edu/click/click.

COVERITY. Software quality and security analysis. http://www.coverity.com/.

DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. 2009. Dmp: deterministic shared memory multiprocessing. In *ASPLOS*.

DIMITROV, M. AND ZHOU, H. 2009. Anomaly-based bug prediction, isolation, and validation: an automated approach for software debugging. In *ASPLOS*.

DINNING, A. AND SCHONBERG, E. 1991. Detecting access anomalies in programs with critical sections. *SIGPLAN Not. 26*, 85–96.

EDELSTEIN, O., FARCHI, E., NIR, Y., RATSABY, G., AND UR, S. 2002. Multi-threaded java program test generation. *IBM Systems Journal*.

FARCHI, E., NIR, Y., AND UR, S. 2003. Concurrent bug patterns and how to test them. In *IPDPS*.

FLANAGAN, C. AND FREUND, S. N. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*.

FLANAGAN, C. AND FREUND, S. N. 2009. Fasttrack: efficient and precise dynamic race detection. In *PLDI*.

FLANAGAN, C., FREUND, S. N., AND YI, J. 2008. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*.

FLANAGAN, C. AND GODEFROID, P. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '05. ACM, New York, NY, USA, 110–121.

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

GUO, P. J. AND ENGLER, D. 2009. Linux kernel developer responses to static analysis bug reports. In *USENIX*.

HASTINGS, R. AND JOYCE, B. 1992. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference*.

JIN, G., THAKUR, A., LIBLIT, B., AND LU, S. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. OOPSLA '10. ACM, New York, NY, USA, 241–255.

JONES, R. W. M. AND KELLY, P. H. J. 1997. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*.

JOSHI, P. AND SEN, K. 2008. Predictive typestate checking of multithreaded java programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. IEEE Computer Society, Washington, DC, USA, 288–296.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7, 558–565.

LU, S., PARK, S., SEO, E., AND ZHOU, Y. 2008. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*.

LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. 2006. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*.

LUCIA, B. AND CEZE, L. 2009. Finding concurrency bugs with context-aware communication graphs. In *MICRO*.

LUCIA, B., CEZE, L., AND STRAUSS, K. 2010. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10. ACM, New York, NY, USA, 222–233.

LUCIA, B., DEVIETTI, J., STRAUSS, K., AND CEZE, L. 2008. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*.

LUCIA, B., WOOD, B. P., AND CEZE, L. 2011. Isolating and understanding concurrency errors using reconstructed execution fragments. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. PLDI '11. ACM, New York, NY, USA, 378–388.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*.

MOZILLA DEVELOPERS. Bug 123930 (deadlock). https://bugzilla.mozilla.org/show_bug.cgi?id=123930. Let them eat races.

MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. 2008. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*.

NARAYANASAMY, S., PEREIRA, C., AND CALDER, B. 2006. Recording shared memory dependencies using strata. In *ASPLOS*.

NARAYANASAMY, S., WANG, Z., TIGANI, J., EDWARDS, A., AND CALDER, B. 2007. Automatically classifying benign and harmful data racesallusing replay analysis. In *PLDI*.

NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*.

NETZER, R. H. B. AND MILLER, B. P. 1991. Improving the accuracy of data race detection. In *PPoPP*.

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

OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. P. 2009. Kendo: efficient deterministic multithreading in software. In *ASPLOS*.

PARK, C.-S. AND SEN, K. 2008. Randomized active atomicity violation detection in concurrent programs. In *FSE*.

PARK, S., LU, S., AND ZHOU, Y. 2009 a. Ctrigger: Exposing atomicity violation bugs from their finding places. In *ASPLOS*.

PARK, S., VUDUC, R. W., AND HARROLD, M. J. 2010. Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. ACM, New York, NY, USA, 245–254.

PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. 2009 b. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP*.

QADEER, S. AND WU, D. 2004. Kiss: keep it simple and sequential. In *PLDI*.

ROSSBACH, C. J., HOFMANN, O. S., AND WITCHEL, E. 2009. Is transactional programming actually easier? In *WDDD*.

RUWASE, O. AND LAM, M. 2004. Cred: A practical dynamic buffer overflow detector. In *NDSS*.

SADOWSKI, C., FREUND, S. N., AND FLANAGAN, C. 2009. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP*.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*.

SECURITYFOCUS. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

SEN, K. 2008. Race directed random testing of concurrent programs. In *PLDI*.

SEN, K. AND AGHA, G. 2006. Automated systematic testing of open distributed programs. In *FSE*.

SHI, Y., PARK, S., YIN, Z., LU, S., ZHOU, Y., CHEN, W., AND ZHENG, W. 2010. Do i use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*.

SULLIVAN, M. AND CHILLAREGE, R. 1992. A comparison of software defects in database management systems and operating systems. In *FTCS*.

SUMNER, N. AND ZHANG, X. 2009. Algorithms for automatically computing the causal paths of failures. In *Fundamental Approaches to Software Engineering*.

TIAN, C., NAGARAJAN, V., GUPTA, R., AND TALLAM, S. 2008. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*.

VAZIRI, M., TIP, F., AND DOLBY, J. 2006. Associating synchronization constraints with data in an object-oriented language. In *POPL*.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*.

XIONG, W., PARK, S., ZHANG, J., ZHOU, Y., AND MA, Z. 2010. Ad hoc synchronization considered harmful. In *OSDI*.

XU, M., BODÍK, R., AND HILL, M. D. 2005. A serializability violation detector for shared-memory server programs. In *PLDI*.

YI, J., SADOWSKI, C., AND FLANAGAN, C. 2009. Sidetrack: generalizing dynamic atomicity analysis. In *PADTAD*.

YU, J. AND NARAYANASAMY, S. 2009. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*.

YU, Y., RODEHEFFER, T., AND CHEN, W. 2005. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*.

Z. LI ET. AL. 2006. Have things changed now? – an empirical study of bug characteristics in modern open source software. In *ASID workshop in ASPLOS*.

ZHANG, W., SUN, C., AND LU, S. 2010. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*.