ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach

Wei Zhang Chong Sun Shan Lu
Computer Sciences Department, University of Wisconsin– Madison
{wzh,chong,shanlu}@cs.wisc.edu

Abstract

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 runs 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 effects: 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-dereference, dangling-pointer, buffer-overflow, uninitialized-read) on shared memory objects.

Guided by this study, we built ConMem to monitor program execution, analyze memory accesses and synchronizations, and predicatively 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 9 real-world severe concurrency bugs. ConMem detects more tested bugs (8 out of 9 bugs) than a lock-set-based race detector and an unserializable-interleaving detector that detect 4 and 5 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

Keywords Software testing, concurrency bugs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03. . . $10.00

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 difficult to detect and diagnose due to their unique non-determinism. Concurrency bugs can cause severe software failures and real-world disasters, such as the Northeastern Blackout of 2003 [41]. 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 to the execution length for each input). Thoroughly checking this large space is crucial because concurrency bugs only manifest under certain interleavings. Unfortunately, in reality, software development resources 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 and open problem.

To address the above challenge, previous concurrency bug detection and testing works [14, 31, 40, 50] intelligently focus on certain interleaving patterns that are prone to concurrency bugs. Widely used patterns include data races (un-synchronized conflicting accesses to shared variables) and atomicity violations (an interleaving that makes certain code regions unserializable).

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

Firstly, large number of false positives could keep programmers away from bug reports. Previous research [4, 29] observes that only approximately 2–10% of *real* data races are harmful; a similar trend is also seen among unserializable interleavings [34].

Secondly, not all bugs present equally harmful end effects, while they are not differentiated by many previous work. Table 1 illustrates this trend by breaking down the relationship between faults (i.e., buggy interleaving patterns) and failures in 70 real concurrency bugs that are reported and fixed in open-source software (Section 3 will explain how we get this data).

	Crash	Hang	Minor Func. Issues
Atomicity Violation	26	3	19
Order Flip	11	3	6
Other	0	1	1

Table 1: Types of failures vs. types of faults (Note: Since this only includes fixed bugs, the real percentages of minor failures for each row should be even larger.)

Figure 1 visualizes the limitations of previous works (and our opportunities) by projecting a concurrent program's interleavings into a two-dimension space. The x-axis and y-axis represent differ-

ent *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.

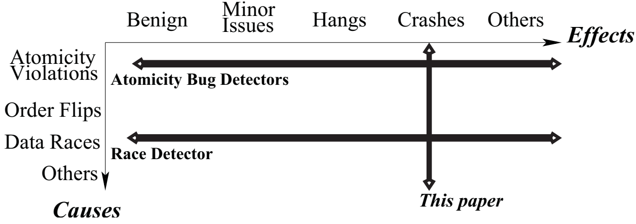

Figure 1: A conceptual two-dimension interleaving space

Previous concurrency bug detection work each takes a *horizontal stripe* of the above 2-D interleaving space. These horizontal approaches inevitably suffer from the following limitations.

Firstly, lack of good **coverage for certain type of failures**. Developers naturally want to know about all (or most) interleavings that can cause certain negative effects, such as software crashes. Unfortunately, interleavings that cause certain effects span vertically in the space and are difficult to adequately capture through a horizontal approach. This difficulty is reflected in every column in Table 1: no single interleaving pattern can monopolize one type of failures.

Secondly, an inevitably large number of **false positives**. This is clearly shown by the horizontal stripes in Figure 1 and is observed in the real world [4, 29, 34].

Thirdly, a lack of **severity differentiation**. *Severity* is a qualitative metric for software failures. In practice [1, 3], 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*. This lack of severity differentiation can be seen in each row from Table 1.

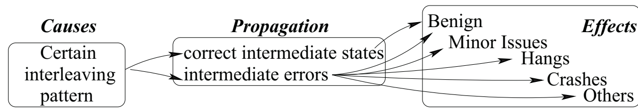

Figure 2: The cause–effect chain

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 wrong states, their intermediate errors might be masked during further propagation (e.g., due to redundant paths [29]), or end up as a minor issue hardly visible to users, in many of which cases the data races or unserializable interleavings are intentionally left there by developers for better performance (e.g., conflicting accesses to performance counter [50]). A pair of concrete examples are shown in Figure 3 and Figure 4. 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.

The false positive issue has already caught the attention of many researchers. Various innovative approaches, such as training [23], automated testing [34, 42] and heuristics-based ranking, have been

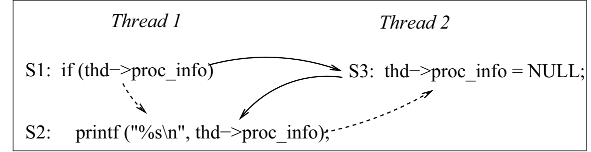

Figure 3: A severe real-world concurrency bug from MySQL database server. (MySQL execution usually follows the dotted line, but it crashes when its interleaving follows the solid line.)

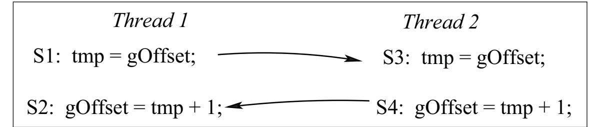

Figure 4: A non-critical concurrency bug that existed in Mozilla for years without any complaint. (`gOffset` holds browsing statistics. Throughout Mozilla, it is read only once in a statistics-printing function.)

proposed to mitigate this problem. However, without changing the underlying horizontal mechanism, these proposals still require significant manual efforts in specification writing and test-oracle design, as well as large amount of computational resources in 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 fixing efforts [1, 3]. This is also observed by a recent study of Linux kernel developers' reactions to static bug detection reports [16]; (2) The huge interleaving space makes the prioritization process extremely important; (3) The unique 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 [26].

In summary, this paper plans to explore 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 pattern categories. This vertical approach will complement existing bug detection work 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 certain general and severe bug effects, program crashes. As a dynamic monitoring tool, ConMem **accurately and predicatively detects severe concurrency bugs that can lead to program crashes, no matter which interleaving pattern (race, atomicity violation, order violation, etc.) is the cause**.

In order 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 the crash effect. Fortunately, we find one.

Our characteristics study (Section 3) of the *cause-effect chains* of 70 real-world concurrency bugs reveals an error-propagation pattern that is both common (including almost half of the examined bugs) and highly correlated with software crashes ($>$ 85% observed conditional probability on both directions). This pattern occurs

when interleavings cause an incorrect order among shared memory operations and *directly* turn to memory errors, including NULL-shared-pointer dereference, dangling pointers to shared memory, un-initialized shared-memory read, and shared-buffer overflow. We refer to this pattern as concurrency-memory errors.

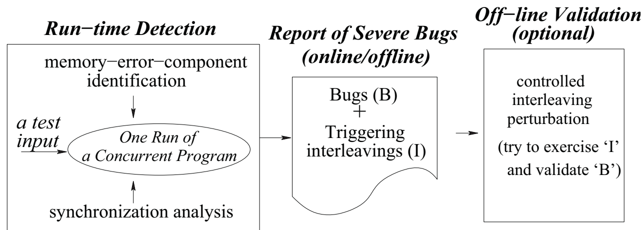

Figure 5: The flow and components of ConMem

Based on the above observation, ConMem (Figure 5) is designed to predicatively 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 to trigger the concurrency-memory error.

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

Overall, this paper has made the following contributions.

- 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 [22] from four widely-used C/C++ applications (Apache, Mozilla, MySQL, and OpenOffice). The study reveals several interesting findings, as follows: (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 dereference, shared-buffer-overflow, uninitialized read to shared variables, and dangling pointers to shared memory.
- A new perspective to check 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.
- 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 [8, 17, 30]. 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 causes (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 [7], 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).
- A tool, ConMem, that effectively detects severe concurrency bugs and validates the results through controlled testing. By design, ConMem has several advantages: (1) predictive bug detection, and thus, insensitive to interleaving; (2) no training requirement; (3) easy-to-validate bug detection results (i.e., memory errors), with no need for manually written oracles to judge execution correctness; (4) high accuracy and coverage on severe concurrency bugs; and (5) simplified diagnosis process supported by ConMem-v. In fact, the co-design of ConMem and ConMem-v also helped to simplify some detection algorithms in ConMem without introducing more false positives to developers.

  ConMem is implemented using binary instrumentation. It is evaluated on 7 open-source programs with 9 severe, real-world concurrency bugs that 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 [47].

  Our results show that ConMem can effectively detect 8 out of 9 tested severe concurrency bugs, better than a race-based detector (4 out of 9) and an atomicity-violation based detector (5 out of 9) in comparison. Furthermore, ConMem detector's false positive rate is about one tenth of the race-based and atomicity-violation based detectors in comparison. 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 slow down for client software and SPLASH2 benchmark, and 3–29% overhead for server applications.

In the following, background and our cause-effect characteristics study are presented in Section 2 and 3. Section 4 discusses ConMem bug detection, followed by ConMem validator in Section 5. Section 6 and Section 7 present evaluation methodology and experimental results. Finally, related works are presented in Section 8.

2. Background

Memory bugs are very common and also severe [44, 51]. 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 dereference happens 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 dereference. However, their accuracy and scalability is limited by pointer alias problems.

Un-initialized read occurs when a valid memory location is read before well initialized. It could cause wrong output or crash. Dynamically detecting un-initialized reads is straightforward. In prac-

tice, sophisticated memory detectors, like Valgrind [30], also consider the context of the un-initialized read, and only report bugs when the un-initialized value is used in critical scenarios.

Accessing invalid memory locations includes dangling pointer bugs (accessing memory locations that are already freed), buffer overflow bugs (accessing memory locations that are beyond the buffer-boundary), and stack smashing (overwriting critical data stored on stack). These bugs can cause not only wrong 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 wrong memory locations. Various algorithms have been proposed to detect these bugs [18, 38].

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 of no difference with those in sequential programs.

The second type such as the one shown in Figure 3 is more complicated. They involve more than one thread and require not only special inputs but also special interleavings to trigger. 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.

3. Cause-Effect Characteristics

Before designing the concurrency bug detection tools, we first study the error propagation process of 70 real-world concurrency bugs. This study will help us understand how buggy interleavings gradually deteriorate program states and ultimately cause various software failures, especially those that are severe (e.g., crashes).

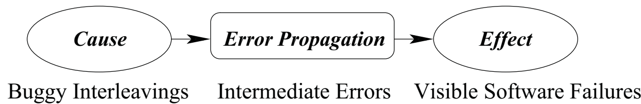

Figure 6: Cause-effect chain

3.1 Methodology and Caveat

Bug Source This study uses a set of 70 real-world non-deadlock concurrency bugs collected in [22][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 toolkits. 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 [12, 22] 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 6).

[1] The original list in [22] includes 74 bugs. 4 of them do not have enough error propagation information and are discarded in this study.

In terms of *causes*, we refer to previous work [22] 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 [44, 51] 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 generalize 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. This paper refers to this as *concurrency-memory errors*. They are further classified based on which types of memory bugs are instantiated, as shown in Table 2.

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 7. This bug and the MySQL bug in Figure 3 both result in NULL-pointer dereference and crashes. 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 based on the *direct* impact of interleavings. Therefore, Figure 7 is considered a concurrency-semantic error.

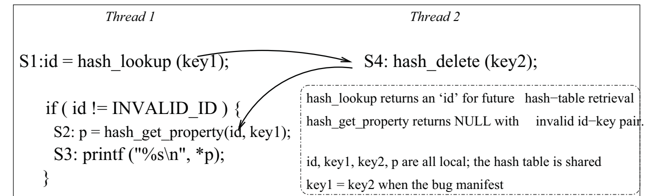

Figure 7: A complicated concurrency bug with intermediate semantic error (simplified from a real Mozilla bug). The buggy interleaving causes an (unexpectedly) invalid {`id`,`key1`} pair, which causes `hash_get_property` to return NULL.

Caveats We attempted to the best of our ability to use representative bugs and correctly classify them. We do not intend to draw general conclusion 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 following characteristics with the methodology in mind. Since 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, since 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 are revealed in this study. Due to the space limit, we only emphasize several findings that are closely related to the design of effect-oriented concurrency bug detection.

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 shared buffer index/boundary variables cause buffer overflow.
Dangling Pointer	Interleaving causes one thread to deallocate a shared buffer before another thread accesses it (Figure 9).
NULL Pointer	Interleaving causes one thread to dereference a shared pointer assigned NULL by another thread (Figure 3).
Uninitialized Read	Interleaving pushes a shared memory read to occur before the intended initialization from a remote thread (Figure 8).
Con-Semantic Errors	Interleaving causes unexpected variable values and program states.

Table 2: Categorization of intermediate errors directly caused by buggy interleavings (*:memory bugs such as double-free and memory-leak are unlikely to happen as direct effects of buggy interleavings)

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

Table 3: Effects (failure types) of concurrency bugs

Finding 1 *Approximately 50% of the studied non-deadlock bugs can cause program crashes*, as shown in Table 3. 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 in our study.* A breakdown between the types of interleaving patterns (causes) and the types of failures (effects) is presented in Table 1. 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 4: 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 4. The few exceptions are similar with the Mozilla bug shown in Figure 7. This finding provides a promising implication: 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 4. This finding is consistent with the trend in sequential programs [51]. 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 Errors
	NULL	UnInit	Dangling	Overflow	
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 5: Breakdown of intermediate errors

Finding 5 *Concurrency-memory errors include four common patterns.* As we can see in Table 5, all the concurrency- memory errors in our study fall into four well-defined categories: NULL-pointer dereference, dangling-pointer, buffer-overflow and uninitialized-read. For simplicity, we will refer to these four sub-types of concurrency-memory errors as follows: **Con-NULL** (NULL pointer dereference directly caused by buggy interleavings), **Con-UnInit** (uninitialized read directly caused by buggy interleavings), **Con-Dangling** (dangling pointer directly caused by buggy interleavings), and **Con-Overflow** (buffer overflow directly caused by buggy interleavings).

These regular bug patterns provide clear guidance to our concurrency bug detection; by focusing on these four types of bugs, the bug detection will have small false positives and small false negatives regarding concurrency bugs with crash-effects. This is precisely the guidance utilized for ConMem.

4. Detecting Severe Concurrency Bugs

The ConMem design is guided by the above characteristics study. It includes four dynamic bug detection modules that are responsible for detecting Con-NULL, Con-UnInit, Con-Dangling, and Con-Overflow bugs, respectively.

4.1 Overview

The design of ConMem follows three principles:

(1) Effect-oriented, instead of interleaving-oriented. ConMem tries not to analyze an interleaving pattern unless it is related to concurrency-memory errors. In the mean-time, 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 targets reporting concurrency bugs that could occur under future interleavings. This property is critical due to concurrent programs' non-determinism.

(3) Balanced analysis accuracy and complexity. Since validator ConMem-v can help prune out false positives, ConMem has some luxury of trading accuracy for simplicity, when necessary.

Following these principles, ConMem dynamically detects concurrency-memory errors in two steps.

First, detecting the basic ingredients of concurrency-memory errors, which are mostly insensitive to interleavings. A concurrency-memory error's basic ingredients are its 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, checking the timing condition among the basic ingredients. The appropriate timing, such as de-allocating a memory object *before* another thread accesses it, turns a set of memory operations into a true bug. Whether a timing condition can be satisfied in future interleavings depends on the synchronizations 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 errors is shown in Table 6. The following sub-sections will elaborate on how to detect each sub-type of concurrency-memory errors, one by one.

	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 (this is an approximated condition)	Yes	Yes

Table 6: The conditions for Concurrency-Memory errors. (*: order synchronization represents barrier-style synchronizations).

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. 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 conducts 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 suspect is reported as a Con-NULL bug only if the timing condition can be satisfied.

4.2.2 Con-NULL detection algorithm

Detecting the basic ingredients Detecting the {`wp`, `rp`} pair for Con-NULL 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 other interfering definition in between.

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

Order synchronization operations [31, 34], 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 certain code regions, e.g., those 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 from `wp`; or (2) `wp` and a later write to `ptr` from the same thread are atomic from `rp`. In the former case, `rp` always uses 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 well protected and report the remaining suspects as Con-NULL bugs.

Of course, our synchronization analysis is neither sound nor complete, mainly because it does not consider potential control flow changes under future interleavings. We believe it provides a good balance between analysis complexity and the 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 [25] binary instrumentation) and off-line trace analysis. We choose trace analysis over pure run-time detection due to the algorithm complexity.

The run-time logs three types of information. The first is the information of a global or heap memory access, which is used to identify basic ingredients (i.e., {`wp`, `rp`}). The second part 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 part is the information of all malloc/free operations. Since virtual addresses could be recycled through malloc/free, this part of information helps us to identify which memory locations are truly holding the same memory object. This 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 pointer accesses from non-pointer accesses based on the read/write values. That is, we ignore memory accesses when the involved values are clearly out of the range of stack, heap, global data region, or 0. This scheme works well in practice. It can be further improved by static analysis.

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

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

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 synchronization can protect a {wp, rp} pair from NULL-pointer dereference. The straightforward solution here is to calculate the vector timestamps for all memory accesses involved in the NULL-Con suspects and to compare their timestamps, according to the condition discussed above. The timestamp calculation is not difficult, because the trace includes information for every order synchronization executed at run-time, including pthread_mutex_create/join and barrier. Using this information, we can easily calculate vector timestamps based on the Lamport logical-timestamp algorithm [20].

The time-consuming part of the above analysis is that for a {wp, rp} pair that accesses memory location ptr, we must find *every* memory access to ptr and check whether it is strictly ordered between wp and rp. The complexity is linear to the number of dynamic write instances to ptr. Our implementation uses a heuristic to simplify the complexity to constant time: If there exists such a ptr-definition that is strictly ordered between wp and rp, it usually comes from either the same thread as wp or the same thread as 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. This heuristic works well in our experiments, never introducing false positives.

Discussions The *false positives* of Con-NULL detection mainly come from two sources. The first is un-identified custom synchronization, an issue shared with many previous concurrency bug detection tools [40]. Without the knowledge of some customized synchronization, ConMem will mistakenly consider some timing conditions as feasible and report false positives. The second are those 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 dereference. 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.

In terms of *false negatives*, in general, the bug detection of ConMem is sensitive to code/path coverage and *in*sensitive to the timing (i.e., interleavings) among a set of code statements. Specifically, with a given test input, if an instruction is executed only under certain rare interleavings, or if two instructions access the same memory location only under certain rare interleavings, ConMem may not be able to catch all basic ingredients of potential Con-NULL bugs, thus resulting in false negatives. This type of false negative is general to all ConMem detection. Fortunately, it rarely occurs based on our experience. In addition, this problem can be mitigated by making ConMem observe more than one runs of the program under the same input. It can also benefit from work that focuses on code coverage in concurrent programs [43].

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 [35]. Based on our experience, it is rarely a problem for Con-NULL, as shown in Section 7. Future work can also split the trace of a long-running program into several sub-traces and extend Con-NULL algorithm to process sub-traces.

4.3 Con-UnInit Detection

4.3.1 What is a Con-UnInit bug?

```
            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));
```

Figure 8: A concurrency bug that leads to undefined read and finally causes crash (from Transmission-1.42)

Con-UnInit bugs are un-initialized memory reads directly caused by buggy interleavings. An example of Con-UnInit bugs is shown in Figure 8. In this example, a shared variable h→ bandwidth is initialized on *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 variable, which causes an assertion failure later.

Con-UnInits' *basic ingredients* typically include a read access, denoted as *r* (e.g., the *S*2 in Figure 8), to a memory location that depends on other threads to initialize. The *timing condition* for a Con-UnInit is to execute *r* before the initializations from other threads.

Note that, when we observe an *r* reading a value defined by its own thread, 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 as run-time detection without trace analysis.

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

This task is quite straight-forward to implement in 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 accesses to heap variables 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 avoid this type of bugs and is not considered here.

Conducting this task at run-time requires several pieces of information. Suppose 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 [20]. 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 we 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 uninitialized reads may not cause negative effects, a property different from NULL-pointer dereference, dangling pointer and buffer-overflow. Previous sequential bug detectors, such as Valgrind [30], have considered this and choose to report bugs only when the uninitialized value is used for critical operations, including system calls, condition checking, and memory address calculation. ConMem can borrow this idea to prune this set of false positives.

Different from Con-NULL, Con-UnInit does not dump traces and does not have the trace size issue. However, since Con-UnInit conducts all the 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 to the heap/global memory footprint of a program, like many previous dynamic bug detectors [23]. It will *not* increase with longer execution, as long as the program's active memory consumption does not change.

4.4 Con-Dangling Detection

4.4.1 What is a Con-Dangling bug?

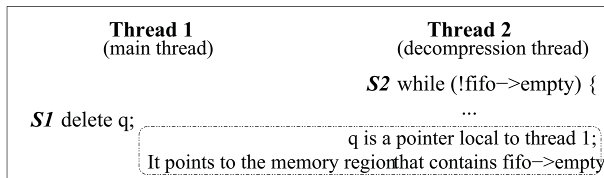

Figure 9: A concurrency bug that leads to dangling pointer and finally causes crash (from PBZIP2-0.9.4)

Con-Dangling occurs when buggy interleavings directly cause dangling pointer accesses. Figure 9 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 upon 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 that in 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 accesses. 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 to `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 (based on timestamps) access 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.

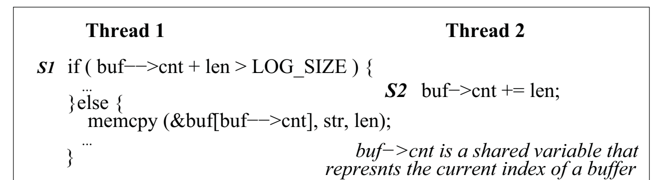

Figure 10: A concurrency bug that can lead to buffer overflow and crash (from Apache-2.0.45)

Figure 10 shows an example of typical Con-Overflow bugs. Thread 1 conducts 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 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 solver techniques [5] can potentially address this value issue.

In our current prototype, we only consider a common sub-set of Con-Overflow bugs: conflicting accesses to shared buffer index variables cause buffer overflows. Specifically, we report all data races among instructions that access 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 [40]. 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 receive the information of data races, the program is executed a second time. Whenever a memory location involved in races 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. Fortunately, we only need to track those accesses and memory locations related to races. Therefore, the overhead is acceptable. We expect that this second step 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 usage. Static analysis can also help us and enable ConMem to report Con-Overflow bug candidates at run-time.

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 at post-mortem analysis. It is also conceivable to combine all these four modules into one big run-time bug detection tool in the future.

5. Bug Exposing and Validation

The design of ConMem-v is inspired by previous tools that validate data race [33] and atomicity violation bug reports [34]. ConMem-v takes every bug reports from ConMem as inputs. It tries to enforce those buggy interleavings predicted in ConMem's bug reports by carefully perturbing the concurrent execution. The whole process is automated.

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

In the following, we will discuss the design and implementation of ConMem-v, explaining what is the interleaving enforcement target and how to enforce a specific timing condition. ConMem-v is implemented using PIN [25] binary instrumentation. Due to the space limit, some implementation details are omitted.

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

In order 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 going 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 into this thread, waiting for the other instruction to arrive from a different thread. This process is demonstrated in Figure 11 (consider A as `wp` and B as `rp`).

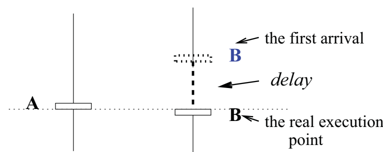

Figure 11: ConMem-v perturbation illustration

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 delay inserted by ConMem-v has time-out, so that the program will not hang.

Validating Con-UnInit bug 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 order 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-Dang bug reports The input to ConMem-v here is a list of instruction pairs $\{F,a\}$. F is a `call` instruction that invokes the de-allocation to 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 order for the F to occur first, which is conducted as Figure 11 (F is A, a is B). In order to know whether a dangling pointer has truly occurred, ConMem-v records and compares the memory address accessed by a and the range of the memory region freed by F.

Validating Con-Overflow bug reports The input here 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 with those for the above three. The unique complexity of Con-Overflows is that even if a buffer index is corrupted to a wrong value through data races, 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 designed for sequential programs [17, 30].

Discussions Two types of interleaving enforcement approaches were proposed before. One is to execute programs on single-core machines and control the scheduling [27, 42]; the other is to insert artificial delays [11, 34]. ConMem-v chooses the latter one for better performance.

In summary, ConMem-v will not report any 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. However, ConMem-v could have false negatives. Some timing conditions require sophisticated interleaving manipulation and may be missed by ConMem-v.

6. 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 (FFT) from SPLASH2 [47]. Except for FFT, all of them contain real concurrency bugs that can cause crashes.

Bugs in evaluation We use 9 concurrency bugs[3] that can cause client and server crash for evaluation. These 9 bugs are all real bugs introduced by the original developers. We carefully set up this bug set to make sure it is representative, covering different

[3] One of these 9 bugs, PBZIP2-2, was not reported in previous document. It is detected in ConMem experiment.

types of faults and error propagation patterns, as shown in Table 7. FFT contains a non-severe concurrency bug introduced by external library developers. We add it into our bug set to measure the false positive rate and overhead of ConMem on scientific applications.

Bug-ID	Causes	Effect Description
MySQL-1	Atom.	Server crash at NULL-ptr dereference
MySQL-2	Atom.	Server crash at NULL-ptr dereference
PBZIP2-1	Order/Atom.	Crash at NULL-ptr dereference
Apache-1	Multi-Atom.	Crash due to dangling ptr
Mozilla	Multi-Atom.	Crash due to dangling ptr
PBZIP2-2	Order	Crash due to dangling ptr
Apache-2	Atom.	Crash/corrupted-log due to overflow
Cherokee	Atom.	Crash/wrong-message due to overflow
Transmission	Order	Crash due to uninitialized read
FFT	Order/Atom.	Wrong output due to uninitialized read

Table 7: 10 bugs in evaluation (Atom.: single-variable atomicity violation; Order: order violation; Multi-Atom.: multi-variable involved atomicity violation.)

Experiment setup

The experiments are conducted on dual quad-core Intel Xeon (2.67GHz) machines, with Linux, version 2.6.18. We use the PIN [25] binary instrumentation framework for all our tools. We use Valgrind–Helgrind [30] as the race detection front-end for Con-Overflow.

Our experiments use bug-triggering inputs reported by the user, like previous dynamic concurrency bug detectors [23, 40, 48]. Note that the bugs **never** manifest during our bug detection runs. Actually, many concurrency bugs do not manifest even after days' execution with bug triggering inputs [27, 34], 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 the false positives and the performance. The reported performance number is the average across multiple runs.

ConMem includes four sub-tools for four types of concurrency-memory errors. Each application will be executed with the bug-triggering input once for each sub-tool. We present the bug detection results of each sub-tool. When ConMem is compared with other detection tools, the true bugs as well as the false positives from all the four sub-tools are put together. The artificial delay used by ConMem-v is 1 milli-second at a time.

We also compare ConMem with two state-of-the-art interleaving checking approaches: race-based (short for *Race*) and atomicity-violation-based (short for *Atom*). *Race* is a lock-set–happens-before hybrid race detector, originally implemented in widely-used Valgrind-Helgrind detector [30] and slightly modified by us for better race coverage. *Atom* is implemented by us based on an algorithm described in previous work [34]. It predicatively 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 bugs [22, 23, 46]). There are other race and atomicity bug detectors, such as happens-before race detectors [31] and training-based atomicity detectors [23]. We did not choose them, because their training requirement or interleaving-sensitive design will make the comparison apples to oranges.

7. Experimental Results

7.1 Overall Results

Overall, as shown in Table 8, 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.

Bug-ID	ConMem	Race	Atom
MySQL-1	Y	Y	Y
MySQL-2	Y	N	Y
PBZIP2	Y	Y	Y
Apache-1	Y	N	N
Mozilla	N	N	N
PBZIP2-2	Y	N	N
Apache-2	Y	Y	Y
Cherokee	Y	Y	Y
Transmission	Y	N	N
FFT	Y	N	Y

Table 8: Bugs detection results (are the bugs detected?)

ConMem shows a good bug detection capability on these evaluated bugs, because it effectively captures the most common error propagation pattern among concurrency bugs with crash-effects. Specifically, three bugs (MySQL-1, MySQL-2, and PBZIP2) are detected by Con-NULL; Apache-1 and PBIP2-2 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. The reason is that the buggy interleaving in Mozilla first causes semantic errors before it finally corrupts memory states and crashes Mozilla. This type of concurrency–semantic error pattern is not captured by ConMem. Because of the complexity of this bug, *Race* and *Atom* also failed to correctly detect this bug.

Atom and *Race* failed to detect 3 and 4 severe concurrency bugs that can be detected by ConMem, mainly because these bugs are not caused by data races or simple atomicity violation. 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 a good coverage on the evaluated real-world concurrency bugs that can cause crashes, not limited to any specific interleaving patterns. It can well complement existing race and atomicity-violation bug detection tools.

7.2 False Positive Results

Before automated pruning

Table 9 shows the numbers 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 interleaving, can dereference NULL-pointer, access freed memory regions, etc. Since some bug reports in Table 9 share the same root cause, the total number of true bug reports there is larger than that in Table 8.

In general, ConMem' false positive rate is much lower (about one tenth) than *Race* and *Atom*, benefiting from its effect-oriented approach. ConMem's false positive rate (about 2.5 false positives per true bug) is reasonably low considering ConMem' 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 9 (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 to that number. Considering

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	8	0	5	0	3	1	0	0	16	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**		22:8		12:5		19:5		0:2		**53:20**	

Table 9: Bug reports and false positives before ConMem-v pruning (Note: 1. the bug report number here is larger than that in Table 8, because some bug reports share one root cause. 2. #FP: # of false positive; #Bug: # of bugs; #ShrMem Inst: instructions that access variables truly shared with other threads.)

that, the interleavings singled out by *Race*, *Atom* and ConMem are much fewer.

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 interleaving 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 10 provides a further breakdown for the false positives of ConMem. As we can see, 51 of these 53 false positives are caused by un-identified custom synchronizations. These 51 bug reports involve infeasible interleavings and can never occur. ConMem mistakenly reported these 51 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 read, as discussed in Section 4.3.

Note that, according to Table 10, *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 data race detection tools, where only about 2–10% feasible data races are true bugs [4, 28].

App.	# Customized synchronization		Benign
	If/While-flag	Producer-Consumer Queue	UnInit
Apache	5	5	0
MySQL	3	13	0
Transm.	14	0	2
PBZIP2	6	3	0
FFT	0	0	0
Cherokee	0	0	0
Mozilla	2	0	0

Table 10: Causes of ConMem false positives

Automated false positive pruning of ConMem-v

All the 73 bug reports of ConMem are sent to ConMem-v for validation. As a result, ConMem-v automatically prunes out *all 53* false positives, without introducing any false negative regarding the bugs shown in Table 8 and Table 7.

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' bug diagnose and fixing. 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. Note that some bug reports in Table 9 have one common root cause and are grouped into one bug in Table 7 and 8. Because of this, failing to expose these 5 bug reports did not cause ConMem-v to miss any root bug there.

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 for 30 times.

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

As discussed in Section 4.2, the bug detection of ConMem is sensitive to the code/path coverage, like all dynamic bug detectors [23, 30, 40], and is mostly insensitive to the timing among certain code. Therefore, we expect ConMem to report more true bugs and more false positives when it observes more program runs that touch previously unobserved code/path.

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 conducts few NULL-pointer assignments, there will be few bug reports, no matter how long the execution is.

7.3 Time and Space Overhead

Table 11 shows the run-time overhead of ConMem tools. Overall, ConMem tools 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 [23, 40, 48] and is suitable for developers' use.

Table 11 excludes the trace-analysis time of Con-NULL, which takes less than 10% of the base-line execution time in our experiments. 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.

Bug-ID	Con-UnInit	Con-Dangling	Con-NULL
Apache	28%	28%	19%
MySQL	13%	24%	29%
Cherokee	6.6%	2.7%	7.6%
Mozilla	196%	185%	505%
PBZIP2	78%	76%	116%
Transmission	80%	79%	82%
FFT	1556%	1285%	1113%

Table 11: ConMem Run-time Performance (overhead %)

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. With the disk size keeps increasing, we believe the trace size will not be an issue for the usage of Con-NULL.

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 synchronizations. Table 12 shows the number of bug suspects that are pruned out by this analysis. As we can see, the pruning is effective.

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 12: Bug suspects pruned by synchronization analysis

8. Related Work

Many related works are discussed in earlier sections. Here, we only discuss a few that are closely related and not discussed yet.

Concurrent programs' empirical study Due to the lack of concurrency bug sources, only a few studies [12, 22] have been done, and they mostly focus on the interleaving patterns of concurrency bugs. Most recently, interesting studies are also conducted to evaluate how new synchronization primitives (such as Transactional Memory) can be used to write concurrent programs [37]. Our paper complements previous studies by looking at concurrency bugs' error propagation process.

Concurrency bug detection, testing and avoidance Existing concurrency bug detection tools can be categorized into race detection [14, 31, 40, 50], atomicity violation detection [6, 13, 15, 23, 39, 48] and deadlock detection [19, 21]. ConMem complements existing tools by focusing on the crash effect, instead of specific interleaving pattern. The predictive interleaving analysis in ConMem is inspired by previous predictive race and atomicity violation detectors [6, 13, 40].

Many innovative approaches, such as training [23] and interleaving testing [27, 34, 42], have been proposed to address the false positive problem in concurrency bug detection. ConMem uses similar synchronization analysis and perturbation-based interleaving enforcement techniques with some of these tools [34]. ConMem complements these tools by handling the problem from a different perspective. It focuses on certain effect of concurrency bugs, instead of a specific interleaving pattern.

Atom-Aid [24] and PSet [49] extended existing dynamic bug detectors by prohibiting certain patterns of interleavings at run time through hardware support in order to survive concurrency bugs during production runs. Software-only tools like Grace [2] and Kendo [32] achieve similar goals for certain types of multithreaded programs at runtime. ConMem can well complement these works by exposing concurrency bugs before they escape to the production run.

Interleaving testing [11, 27] works on systematically exploring the interleaving space. ConMem can complement these works by providing a different perspective on splitting the interleaving space. Deterministic execution works [9, 32] also try to solve the interleaving space challenge. They try to achieve that 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 in concurrent programs. A recent study [7] inventively proposes leveraging race detection to improve data flow analysis in concurrent programs. The idea is very inspiring. However, due to pointer-alias 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 completely different design target with these 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-alias problem and can achieve better accuracy and scalability.

Model checking can also be used to validate certain properties in concurrent programs. Recently, a lot of progress has been made [27, 36] in model checking big concurrent programs. However, the state explosion problem still exists. We expect the effect-oriented approach and the error-propagation characteristics studied in this paper to help provide heuristics to future model checking.

General software failure diagnosis The effect-oriented approach used in ConMem shares a similar flavor with failure diagnosis tools [10, 45]. These tools 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. Different from them, ConMem 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 predicatively detects concurrency bugs with crash effects. In our evaluation with 9 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 reliable ways 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 can well complement 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 customized synchronizations and further decrease the remaining false positives of ConMem. Finally, we can also apply the effect-oriented idea to detecting other types of severe bugs (e.g., security vulnerability, silent data corruption, etc.) in both C programs and Java programs.

10. Acknowledgments

We would like to thank our shepherd, Emery Berger, and the anonymous reviewers for their invaluable feedback. We thank the Opera group from UCSD for sharing with us their bug benchmarks. We also thank Mark Hill for his invaluable feedback and suggestions. This research is partially supported by a Claire Boothe Luce faculty fellowship.

References

[1] Apache Bugzilla. How important is the bug? http://issues.apache.org/bugwritinghelp.html.

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

[3] Bugzilla@Mozilla. A bug's life cycle. https://bugzilla.mozilla.org/page.cgi?id=fields.html#severity.

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

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

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

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

[8] Coverity. Software quality and security analysis. http://www.coverity.com/.

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

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

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

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

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

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

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

[16] P. J. Guo and D. Engler. Linux kernel developer responses to static analysis bug reports. In *USENIX*, 2009.

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

[18] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, 1997.

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

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

[21] T. Li, C. Ellis, A. Lebeck, and D. Sorin. On-demand and semantic-free dynamic deadlock detection with speculative execution. In *USENIX ATC*, 2005.

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

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

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

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

[26] Mozilla Developers. Bug 123930 (deadlock). https://bugzilla.mozilla.org/show_bug.cgi?id=123930. Let them eat races.

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

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

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

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

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

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

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

[34] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their finding places. In *ASPLOS*, 2009.

[35] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[36] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, 2004.

[37] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *WDDD*, 2009.

[38] O. Ruwase and M. Lam. Cred: A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[39] C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, 2009.

[40] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[41] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[42] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[43] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FSE*, 2006.

[44] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *FTCS*, 1992.

[45] N. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In *Fundamental Approaches to Software Engineering*, 2009.

[46] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[47] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[48] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[49] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[50] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[51] Z. Li et. al. Have things changed now? – an empirical study of bug characteristics in modern open source software. In *ASID workshop in ASPLOS*, 2006.