RADBench: A Concurrency Bug Benchmark Suite

Nicholas Jalbert
UC Berkeley
jalbert@cs.berkeley.edu

Cristiano Pereira
Intel
cristiano.l.pereira@intel.com

Gilles Pokam
Intel
gilles.a.pokam@intel.com

Koushik Sen
UC Berkeley
ksen@cs.berkeley.edu

Abstract

Testing and debugging tools for concurrent programs are often validated on known bugs. To aid the development of these tools, we present the Race, Atomicity, and Deadlock Benchmark (`RADBench`) suite. The `RADBench` suite contains the full source of 10 real concurrency bugs found in large open-source software projects including Mozilla SpiderMonkey, Mozilla NSPR, Memcached, Apache Web Server, and Google Chromium Web Browser. We discuss the difficulties we have found in reproducing these bugs that must be accounted for when building testing and debugging tools. Finally, we propose an approach to reproducibility that has a number of benefits over standard deterministic replay for debugging. `RADBench` is open source and publicly available.

1 Introduction

Concurrent programs have become more common with the proliferation of parallel hardware. Bugs that appear only under very specific thread interleavings—concurrency bugs—are one challenge that comes with increased concurrency. Even when the program is executed with the same input in the same environment, these bugs may appear only rarely due to nondeterminism in the thread scheduler and underlying operating system. This nondeterminism complicates the discovery of these bugs during testing. Even when a concurrency bug is confirmed to be in a program, the debugging process is often more difficult because cyclic debugging techniques do not work if a bug cannot be consistently reproduced.

Many researchers have jumped at the opportunity to develop novel algorithms and tools to address the problems presented by concurrency bugs. The past few years have been marked by rapid advancement in testing tools [13, 27, 10], model checking [19, 20] and verification tools [8, 10], concurrency bug avoidance systems [30], determinism-by-default systems [12, 22], bug reproduction tools [25, 6], and debugging systems [15].

A common approach to evaluate tools and systems in this space is to run them on benchmarks with known bugs. This can serve as a sanity check for tool developers as well as a basis for comparison with competing techniques. Moreover, new tools are often developed with specific types of bugs or benchmarks in mind allowing for quicker iteration and more targeted impact.

In our own work, we are interested in developing tools to quickly reproduce concurrency bugs in large open-source C/C++ programs to speed up the debugging process. There are few publicly available C/C++ concurrency bug benchmark suites. The BugBench suite [16] is designed to be a general bug benchmark suite and therefore includes only 4 concurrency bugs from 2 projects. A number of concurrency bugs have been discovered in the PARSEC [7] benchmark suite over its lifetime, but the suite itself is designed for performance benchmarking. The Inspect Model Checking Suite [29] contains a number of smaller buggy concurrent programs. Finally, the authors of [30] maintain a directory of concurrency bugs used in their work.

We believe that a large, diverse set of benchmarks encourages the development of robust tools. Thus, as a first step towards tackling the reproducibility problem, we have built a benchmark suite of 10 concurrency bugs taken from 5 different software projects. Our Race, Atomicity, and Deadlock Benchmark suite, `RADBench`, includes known bugs from large, real-world projects such as Mozilla SpiderMonkey, Mozilla NSPR, Memcached, Apache Web Server, and Google Chromium Browser. `RADBench` includes full source snapshots of each program, inputs and test harnesses to exercise the benchmark, and is distributed with a Linux VM image to solve problems with quirky dependencies.

In this paper, we analyze the bugs in `RADBench` to understand why they are not straightforwardly reproducible. We then propose a novel lightweight bug reproduction approach to efficiently tackle the problem of reproducibility in the realm of debugging. Finally, given

the benefits of having a common set of benchmarks the research community can draw upon, we are publicly releasing the RADBench benchmark suite so that other researchers can test tools and algorithms on these bugs.

The rest of this paper is organized as follows. In Section 2 we describe the programs and bugs included in RADBench. In Section 3, we describe the difficulties involved in finding, reproducing, and fixing these bugs. In Section 4, we propose a novel approach to the problem of reproducibility. In Section 5, we describe related work. Finally, we wrap up in Section 6.

2 The RADBench Suite

The RADBench suite is composed of full source snapshots of large open-source software projects containing real concurrency bugs. Each bug in RADBench has a corresponding bug report in its respective project's issue tracker. To assemble the RADBench suite, we searched each project's bug database for bug reports that suggested a concurrency issue. When we found a promising report, we acquired a snapshot of the code base from before any fix was applied. We then ensured we could build and execute the program in a way that would expose the bug. Finally, we analyzed the root cause of the bug by studying developer comments and code changes.

Table 1 describes the bugs included in RADBench. Column 1 (Program) is the name of the application containing the bug. RADBench includes bugs from Mozilla's SpiderMonkey JavaScript engine, Mozilla's Netscape Portable Runtime (NSPR), the Memcached distributed object caching system, Apache's HTTP server, and Google's Chromium Web Browser. Column 2 (KLOC) shows size of each project in thousands of lines C/C++ code and headers.[1] Column 3 (Issue Number) shows the identifier of the bug report that corresponds to the benchmark bug. Column 4 (Manifestation) describes how the bug manifests. Finally, Column 5 (Time to Fix (days)) shows the time from the initial report until it was closed as fixed. The value *n/a* in this column means the report is open as of the time of writing.

2.1 Mozilla SpiderMonkey

Mozilla SpiderMonkey [3] is the C implementation of JavaScript that powers Mozilla's Firefox browser. SpiderMonkey is released as a standalone product so other projects can embed JavaScript with minimal effort. SpiderMonkey exports an API that can be used by arbitrary C/C++ programs, and, when combined with NSPR, can be built in a threadsafe manner. Concurrent garbage collection is a difficult problem and both SpiderMonkey bugs in RADBench result from unexpected interactions during concurrent garbage collection.

SpiderMonkey-1 occurs when the JavaScript garbage collector is delayed in the middle of collection. During this delay, another thread can modify the internal structures the garbage collector uses to track its progress by clearing a JavaScript context. If the buggy interleaving occurs, the garbage collector may then dereference a NULL pointer. The fix forced JavaScript contexts to stay constant during garbage collection.

SpiderMonkey-2 is an instance of incorrect synchronization between threads sharing teardown tasks. The problem stems from a garbage collector state check that occurs outside the garbage collector lock. A thread may check the state and get an incorrect view of the world if another thread is concurrently running the garbage collection routine. The fix for this bug is to ensure the state check only occurs when the garbage collector is not running.

2.2 Mozilla NSPR

Mozilla Netscape Portable Runtime (NSPR) [2] is a library designed to provide a platform neutral API for system functions and threading. NSPR is used by Mozilla's Firefox browser but is also released as a standalone software package that allows anyone to develop platform independent programs by abstracting away system specific quirks. NSPR was inherited from Netscape and has been developed for over a decade.

NSPR-1 results from the incorrect synchronization of a global lock allocation in a threadsafe timing call. Each thread initially checks if the global lock has been allocated. If not, that thread will then allocate the lock. The problem occurs when multiple racing calls are made before the global lock is successfully allocated. This can lead to multiple threads allocating the same global lock and assertion failures when mutual exclusion invariants are not respected.

NSPR-2 is an atomicity violation in the wait construct provided by the NSPR library that can cause a hang when the waiting thread is awoken by an NSPR interrupt. There is a small window of vulnerability in the NSPR wait constructs such that the waiting thread can release the lock and then an interrupt can occur on another thread before the waiting thread registers its presence. In this case, properly synchronized code that uses the NSPR library can hang.

NSPR-3 is a deadlock that occurs in programs using NSPR read/write locks. A waiting writer thread blocks any thread trying to get a read lock. If read locks are used reentrantly, then the reader thread may get blocked by the writer in the critical section on the reentrant call, thus resulting in deadlock.

[1] as counted by the CLOC utility. http://cloc.sourceforge.net

Program	KLOC	Issue Number	Manifestation	Time to Fix (days)
SpiderMonkey-1	121	476934	Segfault	8
SpiderMonkey-2	121	478336	Assertion Failure	3
NSPR-1	125	354593	Assertion Failure	64
NSPR-2	125	164486	Hang	n/a
NSPR-3	125	526805	Hang	n/a
Memcached-1	8	127	Inconsistent Cache State	n/a
Apache-1	231	44402	Segfault	14
Apache-2	231	45605	Assertion Failure	30
Chromium-1	7523	52394	Assertion Failure	2
Chromium-2	7523	49394	Assertion Failure	n/a

Table 1: Bugs included in `RADBench`.

2.3 Memcached

Memcached [1] is a general purpose, in-memory distributed caching system. Memcached is designed to speed up websites by caching commonly requested data to ease back-end processing and database loads. Memcached has been used by popular sites like Wikipedia, Twitter, and Craigslist.

The `Memcached-1` bug results from built in increment and decrement functions that are not threadsafe. Multiple threads can request increment or decrements for certain cache data and these updates can get lost. As of writing, the trade offs between safe increments using locks and higher performance is being debated.

2.4 Apache HTTP Server

Apache HTTP Server [4] is an open source web server. The Apache HTTP Server project has been under development for over 15 years and is used by over 100 million web sites worldwide. The server itself can be built to use either a threaded or process based concurrency model. The Apache bugs included in `RADBench` result from rare interleavings that can occur in the threaded version (especially when under heavy load).

`Apache-1` is an atomicity violation in an idle worker thread tracking system. The system uses an atomic compare-and-swap to update a global data structure when a worker becomes idle. However, the check that the compare-and-swap succeeded is not atomic. The problem occurs when the compare-and-swap succeeds and then another thread modifies the global data structure directly afterwards. This can make the compare-and-swap appear as a failure. This false failure can result in duplicate updates to the global data structure and general memory corruption that can lead to a segmentation fault. Interestingly, the level of compiler optimization affects whether or not this bug can occur.

`Apache-2` is also related to idle worker tracking. In situations with a very few idle workers, incorrect synchronization can result in the number of idle workers being set to a negative number. This underflow can then result in assertion failures and segmentation faults.

2.5 Google Chromium

Google Chromium [5] is the open source project from which Google draws the core code for the Chrome Web Browser. The Chromium architecture contains both multithreaded and multiprocess elements and is optimized for high performance and security. The Chromium concurrency bugs in `RADBench` manifest as assertion failures in the debug build of the browser.

`Chromium-1` is an atomicity violation in browser initialization that is similar to `NSPR-1`. Multiple threads each try to allocate a set of global statistics tracking objects. If incorrectly interleaved, two threads can allocate the same object, resulting in an assertion failure.

`Chromium-2` is a bug where an untrusted download can be mishandled causing an assertion to fail. This bug involves specifically ordered GUI events combined with threaded handling of the events.

3 Difficulties in Reproducing

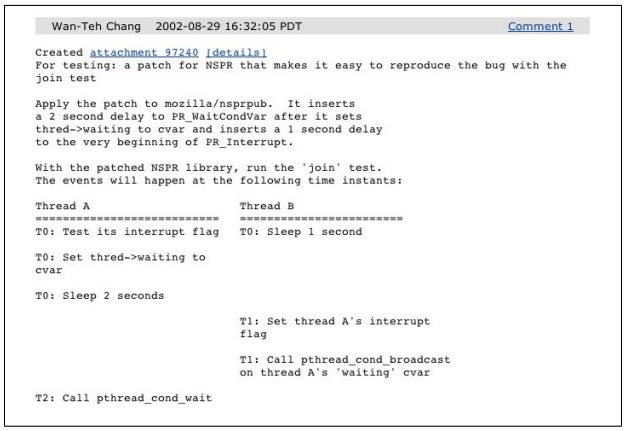
`RADBench` was created as part of a study to look into reproducing concurrency bugs. Reproduction is an important step in fixing concurrency bugs–developers are rarely able to fix a bug they cannot reproduce. As we examined the bugs in `RADBench`, we discovered a number of reoccurring obstacles to reproduction. We next describe these obstacles.

3.1 Environmental Dependencies

In the simplest case, the thread schedule would be the only nondeterministic input that affects bug manifestation. Bugs dependent only on the thread schedule would deterministically occur if you could guarantee a particular ordering between the execution of specific relevant instructions. We, however, found that many of the bugs in `RADBench` depend on other environmental factors.

For example, In `Chromium-1`, the bug occurs due to unexpected interference between multiple threads executing a method to initialize statistics gathering objects. However, the bug can only occur when very specific values are passed as parameters to this initialization method. Most calls to the method cannot result in the bug even in the presence of outside interference. A general way to identify the vulnerable calls is not obvious.

As another example, the buggy interleaving in `Apache-1` can corrupt a global data structure. However, whether or not this corruption results in a crash depends on a large number of other nondeterministic factors, including the memory allocator, the status of other threads in the system, OS timing, synchronization events, and the work load. Just enforcing the interleaving may not be enough to guarantee the bug's manifestation.


Wan-Teh Chang 2002-08-29 16:32:05 PDT Comment 1

Created attachment 97240 [details]
For testing: a patch for NSPR that makes it easy to reproduce the bug with the join test

Apply the patch to mozilla/nsprpub. It inserts a 2 second delay to PR_WaitCondVar after it sets thred->waiting to cvar and inserts a 1 second delay to the very beginning of PR_Interrupt.

With the patched NSPR library, run the 'join' test. The events will happen at the following time instants:

Thread A	Thread B
T0: Test its interrupt flag	T0: Sleep 1 second
T0: Set thred->waiting to cvar	
T0: Sleep 2 seconds	
	T1: Set thread A's interrupt flag
	T1: Call pthread_cond_broadcast on thread A's 'waiting' cvar
T2: Call pthread_cond_wait	

Figure 1: `NSPR-2` bug report comment. Developers often use delays to reason about concurrency bugs.

3.2 Highly Constrained Bugs

Standard order violations require one constraint to be satisfied between two instructions to cause the bug (e.g. instruction A must execute before instruction B). Standard atomicity violations require two constraints to be satisfied between three instructions to cause the bug (see [17] for detailed descriptions of these patterns). A number of bugs in `RADBench` require more constraints than the one or two required by the standard concurrency bug patterns to guarantee the bug's manifestation.

For example, the scenario concocted by developers to explain the `SpiderMonkey-2` bug involves a fairly intricate set of interactions among three threads. While there are certainly many bugs that fit the standard bug patterns, in our experience there are also many concurrency bugs that do not fit these patterns. It is also worth noting that bugs with a large number of constraints are often more difficult to reproduce by chance.

3.3 Context is Important

Dynamic context is often very important in analyzing these concurrency bugs. A simplifying abstraction made by tools like AVIO [18], AtomTracker [21], and Falcon [24] is to reason about the interleaving among static instruction identifiers. We have found a number of bugs in `RADBench` where this abstraction loses too much information, making the analysis results less useful.

For example, `Chromium-1` looks like a standard atomicity violation. However, the atomicity violation occurs in C++ STL code that is reused in multiple parts of the system. An atomicity violation occurring in one context in this STL code (e.g. updating lossy statistics counters) may be benign, whereas atomicity violations in other contexts may result in assertion failure (e.g. browser initialization). Reasoning only with static instruction identifiers can lose context information and conflate benign and buggy interleavings when bug manifestation depends on *both* interleaving and context.

3.4 Programming Paradigm Interactions

Large software systems contain many interacting components. Often these components have very different functions and may even represent completely different programming paradigms. For example, the `Chromium-2` bug results from the interaction between multiple threads interleaving over shared memory *and* a set of specifically timed events propagating into the system from the event-driven GUI. Reproducing bugs that result from the interaction of multiple programming paradigms like this require tools that understand each component and paradigm individually as well as the interaction boundaries between the components.

4 Tackling Reproducibility

The classic approach to reproducing concurrency bugs is to record all nondeterministic events that occur during program execution [26, 11]. These nondeterministic events can include reads from memory, OS interactions, internal timing characteristics, and even the thread schedule. If one is able to capture all relevant nondeterminism during a program execution, then one can guarantee the same thread interactions will occur and any concurrency bug that manifests during recording can be replayed. These systems have a number of drawbacks. Without custom hardware, this level of recording can come with a very high overhead. Further, these system are often extremely difficult to build and maintain–a significant hurdle if they are to be used in a production environment with tight deadlines.

Recently, researchers have experimented with *execution sketching* to mitigate the overhead problem [25, 6]. The key insight is that only a subset of the information required for deterministic replay must be recorded during runtime, thus saving on overhead. Then, if replay

is desired, a high overhead offline search step is used to fill in the details missing from the sketch. While these approaches go a long way toward solving the overhead problem, they still require an expensive offline search as well as a full deterministic record and replay system to perform this search. The overhead in producing a usable replay in this offline search step can foil cyclic debugging efforts and other classic debugging approaches.

While analyzing actual bug reports at the start of the project, we were struck by the method that developers used to reason about concurrency bugs. Figure 1 shows a comment from the actual bug report for the `NSPR-2` bug. Much of the discussion and reproduction efforts were centered around adding delays (e.g. sleep, semaphores, etc.) to make a rare concurrency bug more likely to occur. Moreover, we noticed that most bugs can be reproduced by adding only a small number of delays to code.

We believe that this is a promising approach for bug reproduction; much as other work have given up deterministic recording in favor of sketching, we believe the next logical step is to give up deterministic replay. Such a system would sketch a buggy execution and, instead of replay, do an analysis of the sketch to determine the likely causes to the bug. It would then use heuristics to add in delays to an execution to make that bug more likely to reoccur. While reproduction is not guaranteed, more frequent bug manifestation would aid debugging.

This approach to reproducibility has a number of advantages: the interface with the program under test is the simple addition of delays, hence the effort to implement such a system is reduced compared deterministic record and replay. Moreover, the system can be more robust since it does not have to cope with details like specific OS interfaces. Further, the output of such a system would not be an execution log, but rather it would be the placement of the delays–something developers already use to reason about concurrency bugs. Finally, it leverages the reduced overhead of sketching without expensive offline search, thus enabling cyclic debugging.

We are currently developing this lightweight bug reproduction tool using the `RADBench` bugs for design guidance and as an initial testbed.

5 Related Work

Many deterministic record and replay systems have been proposed. Systems like PinPlay [26] and DejaVu [11] record enough information at runtime to deterministically replay a run. It is worth noting that many of the verification and debugging techniques proposed for concurrent program [20, 15] are built on top of a deterministic record and replay system. LEAP [14] is a record and replay system built for Java that attempts to reduce overhead by reducing the need for global synchronization during record and focusing on recording only globally visible accesses. PRES [25] and ODR [6] are both systems that give up deterministic record in favor of sketching. Sketching greatly reduces the overhead during record time, at the cost of a potentially expensive offline search step when replay is desired.

Our lightweight bug reproduction system builds upon many of the insights underlying concurrency testing systems. The IBM ConTest tool uses schedule perturbation and deterministic record to make concurrency bugs more likely to occur and easier to debug [13]. Race directed active testing [27] uses a predictive analysis to guide testing efforts. Tools like Ctrigger [23], ConMem [31], and Penelope [28] focus on pattern identification to root out bugs. The PCT tool [9] uses a randomized algorithm to insert a small set of delays and provides probabilistic guarantees associated with the discovery of bugs. Compared to these techniques, we believe that we will be able to glean valuable information by analyzing a sketch of an actual buggy execution that predictive techniques would have no way of knowing–information like specific contexts where the bug can occur (and contexts where suspicious races are benign).

The Falcon tool [24] attempts to analyze a buggy execution and rates the suspiciousness of interleavings. We believe this type of analysis can be adapted for lightweight bug reproduction. We aim to take the next step and automatically insert delays to make the bug more likely to occur. Tools like AVIO [18] and AtomTracker [21] are similar in spirit as they want to identify possible atomicity violations and avoid them by analyzing concurrent executions.

6 Conclusion

In this paper we have discussed the importance of reproduction for debugging concurrency bugs and proposed a lightweight bug reproduction approach for addressing the problems still remaining with current techniques. We believe that giving up deterministic replay will make it easier to build reproduction systems for concurrency bugs in industrial settings. We have also described `RADBench`, a collection of 10 real world concurrency bugs from large open-source software projects. `RADBench` is open-source and available for download at the first author's website. `RADBench` includes full source code, test harnesses, and documentation.

7 Acknowledgments

Significant portions of this work were performed by the first author while on internship at Intel. This research is supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906, CCF-0747390, CCF-1018729, and CCF-1018730, and by a Sloan Foundation Fellowship.

References

[1] memcached - a distributed memory object caching system. http://memcached.org.

[2] Netscape Portable Runtime (NSPR) - Mozilla. http://www.mozilla.org/projects/nspr.

[3] SpiderMonkey (JavaScript-C) Engine. http://www.mozilla.org/js/spidermonkey.

[4] The Apache HTTP Server Project. http://httpd.apache.org.

[5] The Chromium Projects. http://sites.google.com/a/chromium.org/dev/.

[6] ALTEKAR, G., AND STOICA, I. ODR: output-deterministic replay for multicore debugging. In *ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)* (2009), ACM, pp. 193–206.

[7] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th International Conference on Parallel Architectures and Compilation Techniques* (2008).

[8] BURCKHARDT, S., ALUR, R., AND MARTIN, M. M. K. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)* (New York, NY, USA, 2007), ACM, pp. 12–21.

[9] BURCKHARDT, S., KOTHARI, P., MUSUVATHI, M., AND NAGARAKATTE, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (New York, NY, USA, 2010), ACM, pp. 167–178.

[10] BURNIM, J., AND SEN, K. Asserting and checking determinism for multithreaded programs. In *7th joint meeting of the European software engineering conference/ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)* (2009), ACM, pp. 3–12.

[11] CHOI, J.-D., AND SRINIVASAN, H. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (New York, NY, USA, 1998), SPDT '98, ACM, pp. 48–59.

[12] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. DMP: deterministic shared memory multiprocessing. In *14th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (2009), ACM, pp. 85–96.

[13] EDELSTEIN, O., FARCHI, E., GOLDIN, E., NIR, Y., RATSABY, G., AND UR, S. Framework for Testing Multi-Threaded Java Programs. In *Concurrency and Computation: Practice and Experience* (2003), pp. 485–499.

[14] HUANG, J., LIU, P., AND ZHANG, C. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *Eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE)* (New York, NY, USA, 2010), ACM, pp. 385–386.

[15] JALBERT, N., AND SEN, K. A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 57–66.

[16] LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. Bugbench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools* (2005).

[17] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *13th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (2008), ACM, pp. 329–339.

[18] LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. *IEEE Micro* (2007), 26–35.

[19] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)* (2007), ACM, pp. 446–455.

[20] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI)* (2008), USENIX Association, pp. 267–280.

[21] MUZAHID, A., OTSUKI, N., AND TORRELLAS, J. Atomtracker: A comprehensive approach to atomic region inference and violation detection. *IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2010), 287–297.

[22] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In *International Conference on Architectural support for programming languages and operating systems (ASPLOS)* (2009), ACM, pp. 97–108.

[23] PARK, S., LU, S., AND ZHOU, Y. Ctrigger: exposing atomicity violation bugs from their hiding places. In *14th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (New York, NY, USA, 2009), ACM, pp. 25–36.

[24] PARK, S., VUDUC, R. W., AND HARROLD, M. J. Falcon: fault localization in concurrent programs. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE)* (New York, NY, USA, 2010), ACM, pp. 245–254.

[25] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: probabilistic replay with execution sketching on multiprocessors. In *22nd symposium on Operating systems principles (SOSP)* (2009), ACM, pp. 177–192.

[26] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *8th annual IEEE/ACM international symposium on Code generation and optimization (CGO)* (New York, NY, USA, 2010), ACM, pp. 2–11.

[27] SEN, K. Race directed random testing of concurrent programs. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)* (2008), ACM, pp. 11–21.

[28] SORRENTINO, F., FARZAN, A., AND MADHUSUDAN, P. Penelope: weaving threads to expose atomicity violations. In *Eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE)* (New York, NY, USA, 2010), ACM, pp. 37–46.

[29] YANG, Y., CHEN, X., AND GOPALAKRISHNAN, G. Inspect: A Runtime Model Checker For Mulithreaded C Programs. Tech. Rep. UUCS-08-004, University of Utah, 2008.

[30] YU, J., AND NARAYANASAMY, S. A case for an interleaving constrained shared-memory multi-processor. In *36th annual international symposium on Computer architecture (ISCA)* (New York, NY, USA, 2009), ACM, pp. 325–336.

[31] ZHANG, W., SUN, C., AND LU, S. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (New York, NY, USA, 2010), ACM, pp. 179–192.