C11Tester: A Fuzzer for C/C++ Atomics Extended Abstract

Weiyu Luo
weiyul7@uci.edu
University of California, Irvine

Brian Demsky
bdemsky@uci.edu
University of California, Irvine

1. Motivation

The C/C++11 standards added a weak memory model with support for low-level atomics operations [3, 7] that allows developers to craft efficient concurrent data structures that scale better or provide stronger liveness guarantees than lock-based data structures. The potential benefits of atomics can lure both experts and novice developers to use them. However, writing correct concurrent code using these atomics operations is extremely difficult.

Simply executing concurrent code is not an effective approach to testing. Exposing concurrency bugs often requires executing a specific path that might only occur when the program is heavily loaded during deployment, executed on a specific processor model, or compiled with a specific compiler. Thus, it is necessary to develop tools to help test for concurrency bugs.

2. Limitations of the State of the Art

The state of the art works falls into two categories. The first category is model checking, which can be useful for finding bugs in small unit tests of concurrent data structures. Recently, much work has been done on developing sophisticated partial order reduction algorithms to scale model checkers to longer executions [1, 2, 4, 5, 6, 8, 9, 12]. Despite all of this work, model checkers still do not scale beyond short executions due to fundamental limitations that arise because the number of concurrent executions grows exponentially with the execution length. This limits the applicability of model checking to small unit tests—it is extremely difficult to scale to full applications or even the larger unit tests that are required to test functionality such as resizing a data structure.

The second category of prior work is race detectors. The tsan11 [10] and tsan11rec [11] data race detectors have been developed for a modified version of the C/C++ memory model, but they implement a significantly stronger memory model than the real C/C++ memory model. Both of these tools can only generate a subset of the executions allowed by the C/C++ memory model. In particular, they can only generate executions in which the writes are ordered in the same order that they process them. More precisely, they can only generate executions in which $hb \cup sc \cup rf \cup mo$ is acyclic and thus miss potential bug-revealing executions that both are allowed by the C/C++ memory model and can be produced by mainstream hardware including ARM processors. We have example bugs that tsan11 and tsan11rec miss because they involve executions with cycles in $hb \cup sc \cup rf \cup mo$, while C11Tester can detect these bugs.

```
1  atomic<int> x(0), y(0);
2
3  void threadA() {
4    x.store(1, memory_order_relaxed);
5    y.store(1, memory_order_relaxed);
6  }
7  void threadB() {
8    r1 = y.load(memory_order_relaxed);
9    x.store(2, memory_order_relaxed);
10   printf("r1 = %d\n", r1);
11 }
12 void threadC() {
13   r2 = x.load(memory_order_relaxed);
14   r3 = x.load(memory_order_relaxed);
15   printf("r2 = %d\n", r1);
16   printf("r3 = %d\n", r2);
17 }
```

Figure 1: Example of execution that tsan11 and tsan11rec miss. Both tools cannot generate the execution where `r1=1`, `r2=2`, and `r3=1`.

Figure 1 presents an example of an execution that tsan11 and tsan11rec cannot generate, but C11Tester can generate. For tsan11 and tsan11rec, if `r1=1`, then the $hb \cup rf$ edges in this example force the store at line 4 to be modification ordered before the store at line 9, and hence, the result that `r2=2` and `r3=1` is not allowed. To forbid this execution under the C/C++ memory model and ARM/PowerPC processors, the store to `y` must be a release and the load from `y` must be an acquire. *This example shows a problematic pattern for real world code, because the same pattern appears in commonly used data structures such as the write-lock of a reader-writer lock and the writer of a seqlock.* Tsan11 and tsan11rec would not find bugs in which the wrong memory orders were used in a `write_lock` method from a reader-writer lock if the lock protected a critical section with atomic operations or in the writer of a seqlock. C11Tester can find such bugs.

C11Tester's constraint-based approach to modification order supports a much larger fragment of the C/C++ memory model than tsan11 and tsan11rec. The memory model fragment that C11Tester supports requires $hb \cup sc \cup rf$ be acyclic. C11Tester fully supports the C/C++ memory model for release, acquire, and sequentially consistent atomics. C11Tester adds minor constraints to forbid out-of-thin-air (OOTA) executions for relaxed atomics. Furthermore, the constraints that C11Tester places on the C/C++ memory model to forbid OOTA executions appear to incur minimal overheads on existing ARM processors [13] while x86 and PowerPC processors already implement these constraints.

Finally, some prior race detectors [10] rely on the operating system scheduler to control the scheduling of threads and do not support controlled scheduling nor can they easily reproduce the same schedule.

3. Key Insights

This paper has the key insight of taking a constraint-based approach to implement the C/C++ memory model's modification order (cache coherence order). This approach enables C11Tester to support a much larger fragment of the C/C++ memory model. The second key insight is that clock vectors can be used to track constraints on the modification order relation much more efficiently than previous approaches, making a constraint-based approach to modification order feasible for executions with millions of atomic operations.

These insights advance the state of the art because they enable C11Tester to handle a much larger fragment of the C/C++ memory model at same performance as previous tools that only handle a smaller fragment of the memory model.

4. Main Artifacts

Our main artifacts are (1) a new fuzzing algorithm for the C/C++11 memory model that can efficiently support a large fragment of the C/C++ memory model and (2) an implementation of this artifact in the C11Tester fuzzer. We then evaluate the algorithm on several benchmark applications and compare our results to tsan11 and tsan11rec.

5. Key Results and Contributions

This paper presents a new constraint-based approach to fuzzing C/C++ programs with atomics that can produce executions from a much larger fragment of the C/C++ memory model than was previous possible.

This paper makes the following contributions:

- It presents a fuzzing tool for the C/C++ memory model that can fuzz full programs.
- It presents a fuzzing tool that supports a larger fragment of the C/C++ memory model, *i.e.*, $hb \cup sc \cup rf$ being acyclic, than previous fuzzing tools.
- It presents a constraint-based approach to the C/C++ modification order relation. This technique enables it to efficiently support the larger fragment of the C/C++ memory model.
- It develops a new technique to allow fibers to provide efficient controlled scheduling for threads while supporting thread-local storage that is required by essentially all real applications.
- It presents techniques for reducing the memory usage of the fuzzer.
- It integrates this approach with a data race detection algorithm.
- It evaluates C11Tester on several applications and compares against both tsan11 and tsan11rec. It shows that C11Tester can find bugs that tsan11 and tsan11rec miss. It shows that C11Tester has the same performance or is faster than previous race detectors that support controlled execution.

C11Tester improves over previous work by supporting a larger fragment of the C/C++ memory model. This larger fragment is important because it can catch more bugs than previous work and thus provides developers with stronger assurances that their code will work correctly when deployed.

Prior data race detection tools could only generate executions in which the modification order was consistent with the order the tools processed memory operations. C11Tester overcomes the limitations of previous work by developing an efficient constraint-based approach to implementing modification order that allows C11Tester to efficiently generate executions in which the modification order is not constrained to be the same order C11Tester processes the statements.

6. Why ASPLOS

This paper touches on all three areas of ASPLOS: architecture, programming languages, and operating systems. Memory models arise primarily because of optimizations from the architecture community and to a lesser degree due to optimizations in compilers. System software often makes use of the atomic primitives, and thus the systems community is a prime candidate to benefit from C11Tester. Finally, C11Tester contributes new techniques to the programming language community.

7. Summary

This paper shows how to efficiently fuzz programs with respect to a larger fragment of the C/C++ memory model than was previously possible. The paper develops new techniques that enable it to efficiently support a constraint-based approach to handling the modification order from the C/C++ memory model. The technique is demonstrated to be effective on a range of C/C++ software that makes use of atomic operations.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):135:1–135:29, October 2018.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2):7:1–7:74, July 2014.

[3] Pete Becker. ISO/IEC 14882:2011, Information technology – programming languages – C++, 2011.

[4] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, pages 113–124, 2011.

[5] Brian Demsky and Patrick Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Proceedings of the 2015 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–36, October 2015.

[6] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 2015 Conference on Programming Language Design and Implementation*, pages 165–174, 2015.

[7] ISO JTC. ISO/IEC 9899:2011, Information technology – programming languages – C, 2011.

[8] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proceedings of the ACM on Programming Languages*, 2(POPL):17:1–17:32, December 2017.

[9] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 96–110, 2019.

[10] Christopher Lidbury and Alastair F. Donaldson. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 443–457, New York, NY, USA, 2017. ACM.

[11] Christopher Lidbury and Alastair F. Donaldson. Sparse record and replay with controlled scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 576–593, 2019.

[12] Brian Norris and Brian Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 131–150, October 2013.

[13] Peizhao Ou and Brian Demsky. Towards understanding the costs of avoiding out-of-thin-air results. *Proceedings of the ACM on Programming Languages Volume 2 Issue OOPSLA*, 2(OOPSLA):136:1–136:29, October 2018.