UNIVERSITY OF CALIFORNIA,
IRVINE

On Fuzzing Concurrent Programs With C++ Atomics

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Engineering

by

Zachary Snyder

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank...

- Ahmed for helping gather benchmark programs for this project.
- Weiyu for his assistance in developing the compiler pass.
- The NSF for supporting this work.

This project was partly supported by the National Science Foundation under grants OAC-1740210, CNS-1703598, SHF-1217854, and CCF-1319786.

ABSTRACT OF THE THESIS

On Fuzzing Concurrent Programs With C++ Atomics

By

Zachary Snyder

Master of Science in Computer Engineering

University of California, Irvine, 2019

Brian Demsky, Chair

Concurrent software is difficult to debug. C++ uses a relaxed memory model, so concurrent software written in C++ using atomics is extra difficult to debug. We built a smart fuzzer for concurrent C++ programs that implemented the relaxed memory model for C++ atomics. This paper presents an informal operational semantics of the C++20 memory model, along with a description of the work done to implement it in the fuzzer. Code is instrumented using a compiler pass to replace operations with calls into the fuzzer library which is preloaded into the program. The fuzzer intercepts the main function to take control of the program. We show that the fuzzer can find bugs in small demo programs created as unit tests for other tools. Unfortunately, our approach to program instrumentation is defeated by programming practices that are common to real-world code using C++ atomics. Additional limitations of our instrumentation approach are discussed, as well as other challenges we ran into when working with real-world code. We then propose an alternate instrumentation strategy that could be used to overcome all observed challenges for future work.

Chapter 1

Introduction

Concurrent programs are notoriously difficult to debug. Many concurrent programs are written in C/C++, because it offers a combination of high performance and extensive library support, and broad platform compatibility. The C++ memory model is a relaxed memory model, which introduces additional subtle ways for programs to behave differently from programmers' expectations, making some C++ programs especially difficult to debug. Some bugs in a program that depend on those subtle semantics might not show up on some real hardware architectures due to memory model implementation details in either the hardware or the compiler back-end for that architecture.

Practically speaking, current techniques are insufficient for proving the correctness of these programs, and standard methods of testing don't work on concurrent programs.

There is work on model checkers for the C++ memory model, but model checkers do not scale to real programs. There are just too many states in a real application to practically explore. The only real option left is to write a fuzz tester.

One needs a specially written fuzzer in order to actually test concurrent programs. Existing fuzzers don't implement the subtle semantics of the C++ relaxed memory model, so some buggy behaviors might exist that cannot be found by existing techniques.

The aim of this project is to build a fuzzer that implements the C++ memory model precisely, and can scale up to test real-world programs.

This document describes the current state of the effort to accomplish this goal, and documents what lessons have been learned on the subject thus far.

Chapter 2

Related Work

2.1 Model Checkers with Relaxed Memory Models

Some model checkers have been developed that simulate relaxed memory models. So far, model checkers haven't been able to scale to real-world applications due to the state-space explosion problem, but they're pretty good at checking the implementations of data structures for correctness, given an appropriate set of test drivers. CDSChecker [27, 28] is a model checker against the C++11 memory model with slight modifications so as to avoid out-of-thin-air behavior that uses DPOR [16] and backtracking to explore all the possible behaviors. CDSSpec [30] is an extension for CDSChecker that additionally checks annotations describing the behavioral specification of data structures. DiVinE [6] is an LTL model checker that was extended to support the relaxed constraints of the x86 TSO memory model [33]. RCMC is a model checker against the RC11 [20] memory model, which is a modified version of the C++11 memory model that doesn't allow for out-of-thin-air behavior. RCMC is very similar to CDSChecker in both approach and design goals, but does a better job of avoiding redundant and infeasible executions by exploiting the properties of the RC11 memory model. Tracer [4] is a model checker that only explores the acquire-release fragment of the C++11 memory model. There is work done by Zhang et al. [36] on model-checking TSO and PSO by incrementally relaxing the constraints on executions down from sequential consistency. There is work done by Jonsson [19] on translating C programs under RMO (the SPARC v9 memory model) into models that can be explored by SPIN [18].

2.2 Race Detectors with Relaxed Memory Models

Race detectors look for data races in programs by instrumenting loads, stores, and mutex operations, and comparing the program's behavior against some specification of what constitutes racy and non-racy behavior. Race detectors scale really well, since they just record when certain events happen and validate them against some rules, but don't take control of the program's schedule or attempt to run the program many times. As such, their bug-finding capacity is somewhat limited. They can only really find bugs which happen to occur during a particular execution, or which can be inferred as being possible from that execution. ThreadSanitizer [32] is a very popular sanitizer for concurrent programs that has been extended to reason about the C++11 relaxed memory model [22]. ThreadSanitizer is the only known race detector that makes any attempt to reason about a relaxed memory model.

2.3 Concurrent Fuzzers

Concurrent fuzzers manipulate the program's schedule and run the program many times in order to find bugs. Concurrit [12] is a DSL that allows programmers to specify specific schedulings of operations in their test cases to try to find bugs. Concurrit also allows programmers to search for a schedule that will produce a given behavior. ConTest [14] is the first known tool that attempts to permute a program's schedule in an effort to guide a program into buggy executions. RAPOS [31] improves upon ConTest by employing a heuristic to try to select executions that sample the possible linearizations of the partial orders or operations more uniformly. Relacy [35] is a tool created to reason about unit tests under the C++11 memory model. Relacy uses `sleep ()` calls to manipulate the schedule, and is the only fuzzer-type tool known to the author to reason about a relaxed memory model. PCT [8] observes that some partial orders have many more linearizations than others, and thus a uniform sampling of linearizations will non-uniformly sample the partial orders themselves. PCT uses a random selection strategy that provides nice probabilistic guarantees about how long it will take to find a bug.

2.4 Adversarial Memory

The adversarial memory approach instruments loads and stores so that loads return valid but unexpected values in a way designed to trigger unexpected conditions in the target program. Jumble [15] exploits the range of allowed behavior in relaxed memory models to manipulate the loads of a program to return atypical values, like those from old or stale writes. Jumble uses a race checker internally, and goes further than just detecting races, but doesn't manipulate the schedule like other, more aggressive concurrency testing tools.

2.5 The C++ Memory Model

Other work has been done on modeling the C++ memory model - much of it more formal than will be found in this paper. An axiomatic presentation of the C++11 memory model was presented by Batty et al. [7]. A 'repaired' version called RC11 was presented by Lahav et al. [20] that prohibits 'out-of-thin-air' behaviors. An operational semantics for the C++11 memory model, as formalized by Batty et al. was presented by Neinhuis et al. [26].

Chapter 3

Motivating Example

Consider the following implementation of a single producer, single consumer queue.

```
#include <assert.h>

#include <atomic>
#include <condition_variable>
#include <mutex>

template <typename Element>
struct SPSCQueue
{
    SPSCQueue ()
    :    head (new Node ()), tail (this -> head), consumer_waiting (false)
    {
    }

    ~SPSCQueue ()
    {
        assert (this -> head == this -> tail);
        delete this -> head;
    }

    void
    enqueue (Element element)
    {
        Node * node = new Node (element);
        // The next line should be a release.
        this -> head -> next . store (node, std::memory_order_relaxed);
        this -> head = node;

        std::unique_lock lock (this -> m);
        if (this -> consumer_waiting)
        {
            this -> consumer_waiting = false;
            this -> cv . notify_one ();
        }
    }

    Element
    dequeue ()
    {
        // The next line should be an acquire.
        Node * next = this -> tail -> next . load (std::memory_order_relaxed);

        if (! next)
        {
            {
                std::unique_lock lock (this -> m);

                Node * next =
                    this -> tail -> next . load (std::memory_order_relaxed);

                if (! next)
                {
                    this -> consumer_waiting = true;

                    do this -> cv . wait (lock);
                    while (consumer_waiting);
                }
            }

            next = this -> tail -> next . load (std::memory_order_relaxed);
        }

        assert (next);

        Element element = next -> element;

        delete this -> tail;
        this -> tail = next;

        return element;
    }

private:

    struct Node
    {
        std::atomic <Node *> next;
        Element element;

        Node (Element element = Element ()) : next (nullptr), element (element)
        {
        }
    };

    Node * head;
    Node * tail;

    bool consumer_waiting;
    std::mutex m;
    std::condition_variable cv;
};
```

By using atomics for the next pointers in the nodes of the queue, we can separate the producer and consumer so that they may concurrently access the queue without blocking each other.

This implementation has a bug, as indicated by the comments in the code. The write to the next pointer in `enqueue ()` should be a release, and the first attempt to read the next pointer in `dequeue ()` should be an acquire. The reason that this is a bug is that a race condition can occur between reads and writes to the `element` member of the `Node` type.

Consider the partial execution in figure 3.1 of a program that creates a queue, `enqueue ()`s an element in one thread, `dequeue ()`s an element in another thread, and then deletes the queue.

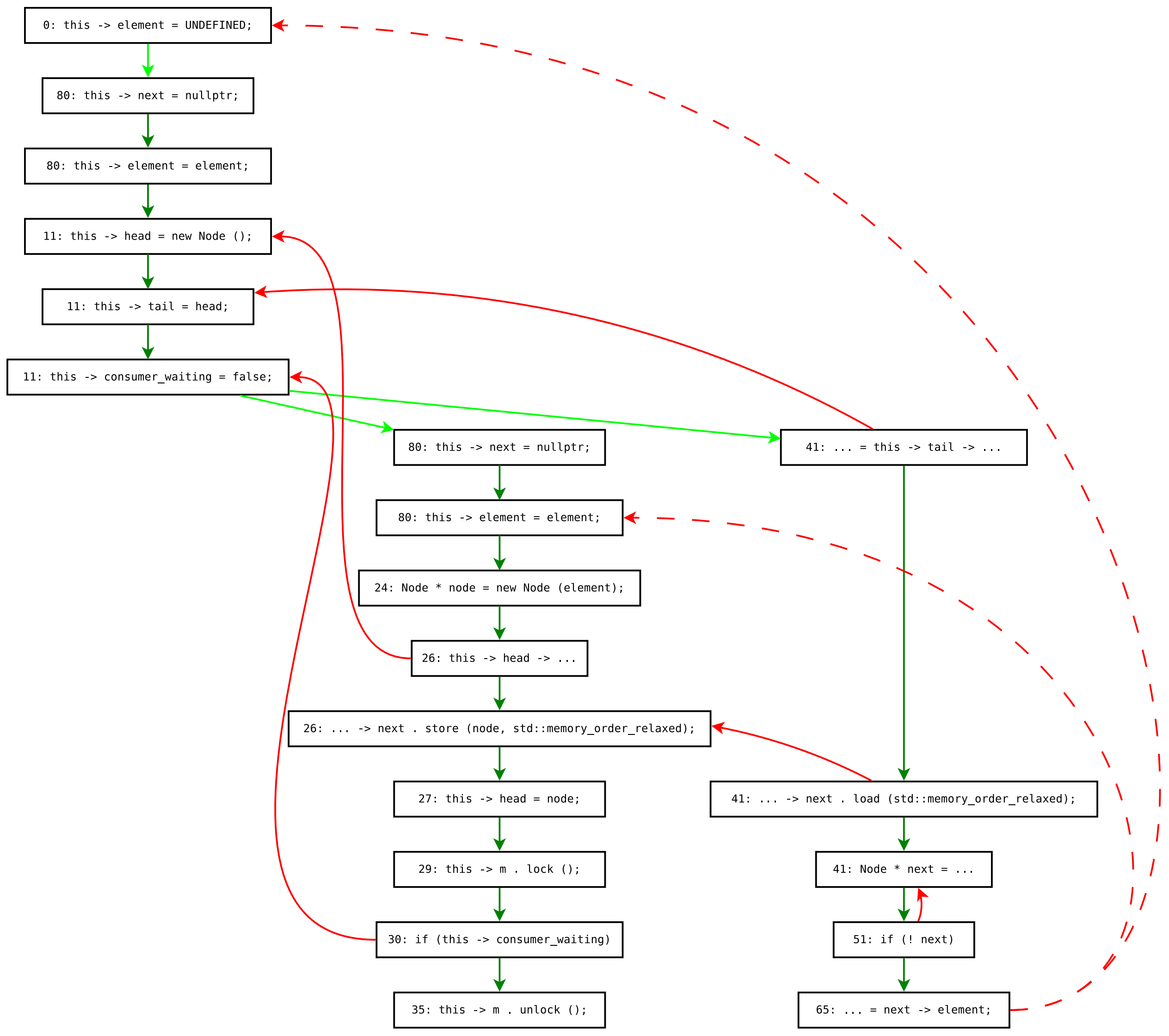
0: this -> element = UNDEFINED;
80: this -> next = nullptr;
80: this -> element = element;
11: this -> head = new Node ();
11: this -> tail = head;
11: this -> consumer_waiting = false;
80: this -> next = nullptr;
80: this -> element = element;
24: Node * node = new Node (element);
26: this -> head -> ...
26: ... -> next . store (node, std::memory_order_relaxed);
27: this -> head = node;
29: this -> m . lock ();
30: if (this -> consumer_waiting)
35: this -> m . unlock ();
41: ... = this -> tail -> ...
41: ... -> next . load (std::memory_order_relaxed);
41: Node * next = ...
51: if (! next)
65: ... = next -> element;

Figure 3.1: An execution of the buggy version of the SPSC queue program.

In this execution, the first attempt to retrieve the enqueued node is successful. Unfortunately, because the read of the `next` field in `dequeue ()` does not acquire from the write to that field in `enqueue ()`, no happens-before is established between the different threads, and so the read of the `element` field races with the write to that same field and can potentially return an uninitialized value.

If read and write are strengthened to acquire/release semantics so that the read acquires from the write, we end up with an execution that looks like the one in figure 3.2.

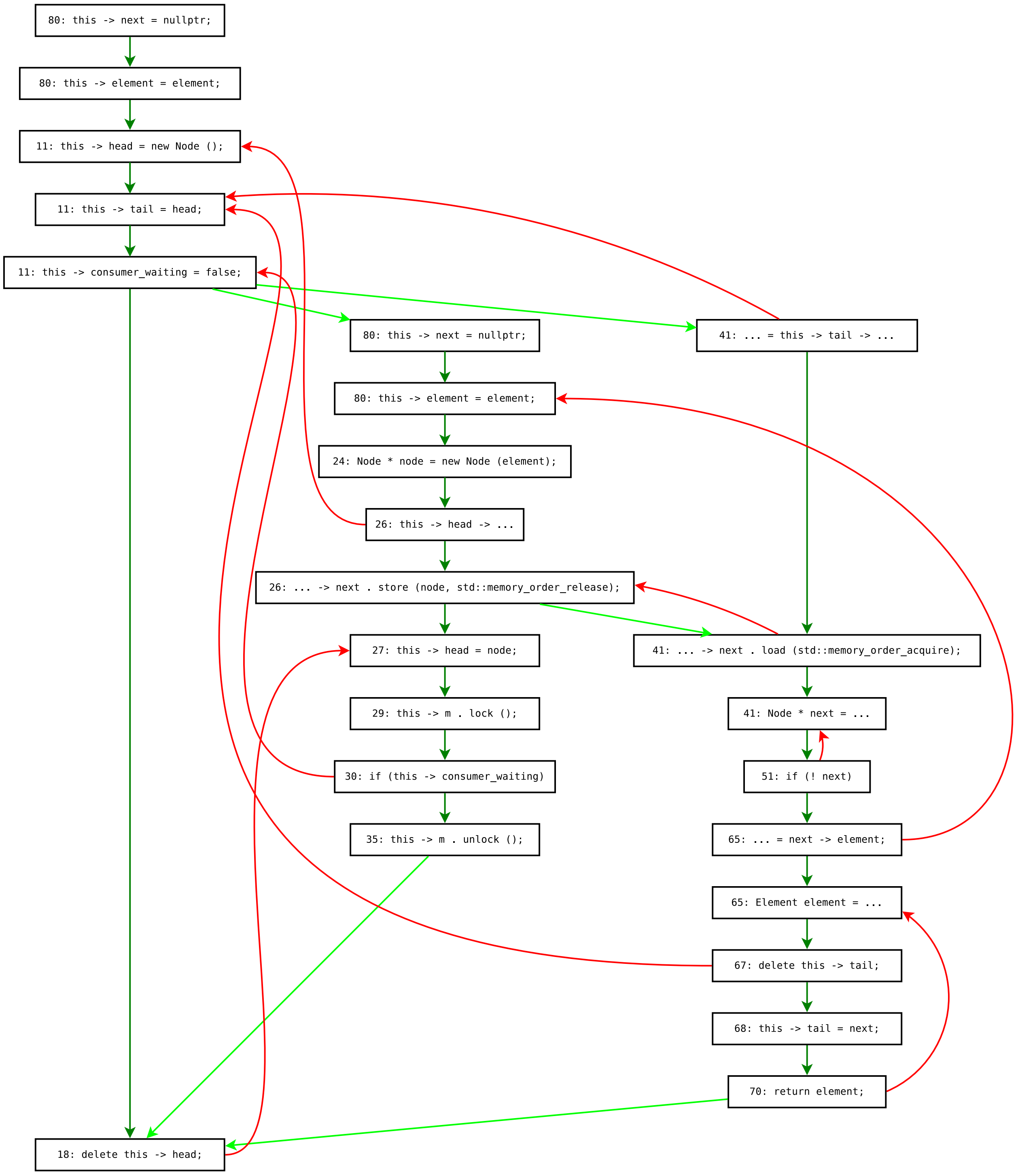
80: this -> next = nullptr;
80: this -> element = element;
11: this -> head = new Node ();
11: this -> tail = head;
11: this -> consumer_waiting = false;
80: this -> next = nullptr;
41: ... = this -> tail -> ...
80: this -> element = element;
24: Node * node = new Node (element);
26: this -> head -> ...
26: ... -> next . store (node, std::memory_order_release);
27: this -> head = node;
41: ... -> next . load (std::memory_order_acquire);
29: this -> m . lock ();
41: Node * next = ...
30: if (this -> consumer_waiting)
51: if (! next)
35: this -> m . unlock ();
65: ... = next -> element;
65: Element element = ...
67: delete this -> tail;
68: this -> tail = next;
70: return element;
18: delete this -> head;

Figure 3.2: An execution of a the fixed version of the SPSC queue program.

Now, because the read of the `next` field in `dequeue ()` acquires from the write to that field in `enqueue ()`, the read of the `element` in `dequeue ()` field happens after the write to that field in `enqueue ()`, and thus does not race with that write, and cannot read an earlier value.

The especially devious thing about this bug is that it doesn't exist on x86 hardware, because normal loads and stores have release-acquire semantics on x86 hardware. This means that testing done on x86 hardware won't uncover this bug. The bug could stay hidden until some poor user comes along with a different hardware architecture that has weaker loads and stores. The only way a developer working on x86 hardware could find this bug is if they used some tool that understood the C++ memory model.

Chapter 4

Initial Assumptions

When we started out on this project, we had a picture of the kinds of programs we wanted to target. In our picture, we were targeting whole applications. Some of the points of our picture were critical to performing a meaningful evaluation of our tool.

- These would be programs that actually served a functional purpose, rather than existing just to be a toy for playing around with concurrency. We want our benchmarks to be representative of real applications that people would actually use the tool to debug.
- These applications would use C++ atomics in a nontrivial way, and would use relaxed memory orderings at least some of the time. While our tool will be able to detect race conditions due to improper use of locks around shared memory, there are more effective tools for dealing with those races in the absence of atomics.
- Most of these applications would implement their own concurrent data structures using C++ atomics, rather than just relying on a well-known library like LibCDS [1] (though being able to test LibCDS would also be good). We need a variety of implementations to test. Otherwise, we're just testing the same implementations in the same library over and over again.

- These applications would be written by people who were familiar with the C++ memory model. A novice is likely to make mistakes that are easily caught with simple code review. Also, a novice will not be able to use our tool, since we will report race conditions that they won't be able to understand.

- These applications would not use concurrency primitives like thread creation or atomic operations inside of static initializers, because that would be an egregious abuse of global variables. The use of global variables is bad practice, and we are generally of the opinion that developers who do such a terrible thing as spawn threads from the initializers of global variables should suffer through the pain that they brought upon themselves.

- These applications would not replace core libraries (like malloc) with libraries that used concurrency primitives internally. Since the fuzzer will also depend on those libraries, being loaded in as a shared library, putting concurrency primitives inside of those libraries will completely break our instrumentation schema, as described later.

Our goal in this project was ultimately to produce two fuzzers. One that was smart, and one that was dumb. The smart fuzzer would have higher overhead per operation than the dumb fuzzer, simply due to being smart. We wanted to see which fuzzing strategy was the best at finding bugs in the target applications. The smart fuzzer was written from scratch, while the dumb fuzzer was adapted from CDS-checker, a model-checker previously developed by our research group.

Chapter 5

An Informal Operational Semantics of the C++ Memory Model

A concurrent fuzzer that fuzzes C++ programs needs to implement the C++ memory model. Typical presentations of the C++ memory model are highly declarative, which doesn't translate well into procedural code.

Also, those presentations over-constrain what a valid execution looks like, so that there are multiple valid executions according to the usual formulation of the memory model that are behaviorally indistinguishable from each other. To be behaviorally indistinguishable means that the reads-from relations are exactly the same between executions. When the reads-from relations are the same between executions, the program logic necessarily takes exactly the same paths, since there is no way for the program to observed subtle differences in the memory model graphs that don't affect which values are returned by the read operations of the program.

Here, we present a way of looking at the C++ memory model which resembles an operational semantics, and which also relaxes the definition of sequential consistency so that there are fewer memory model graphs that exist for the same reads-from relation. This relaxation will help us reduce the number of executions that we have to explore to find bugs when we implement it in the fuzzer. Note that this formulation does not admit cycles in the happens-before $\cup$ reads-from graph, so out-of-thin-air behaviors are not possible. Also note that this is a formulation of the C++ 20 memory model, rather than the C++11 memory model.

5.1 Basic Terms

An *action* is any read, write or atomic fence.

An *atomic* is any atomic read, atomic write, or atomic fence.

An *operation* is any read or write.

Thus, an *atomic operation* is any atomic read or atomic write.

A *non-atomic* is any non-atomic read or non-atomic write.

All non-atomics happen to be operations.

5.2 Sequences

A *sequence* is an ordered collection of writes. When a read-modify-write is executed, it is appended to the sequence containing the write that it reads from if that sequence starts with a write-release. Otherwise, the read-modify-write is placed into a new sequence by itself. All other writes are always placed into new sequences by themselves when executed. When a write is appended to a sequence, we say that it is *sequence-ordered* after the other writes in the sequence. This is not to be confused with *sequenced before* or *sequenced after*, which refers to the ordering of operations within threads.

The *release-heads* of a sequence are the writes-release in a sequence.

The *leading modification set* of a write is the set of writes containing all writes which are sequence-ordered before the write and the write itself.

5.3 Sequenced-Before

A *thread* is an ordered collection of actions. When an action is executed, it is appended to a thread. When an action is appended to a thread, we say that it is *sequenced after* all the other actions in the thread. Conversely, when an action is appended to a thread, we say that all other actions in the thread are *sequenced before* the new action.

5.4 Reads-From

Reads *read from* writes. Each read reads from exactly one write. A write may be read from by at most one read-modify-write. We refer to the write that a read reads from as the *target* of the read. We may also refer to the reads that read from a write as the reads belonging to that write, or the write's reads.

5.5 Happens-Before

Sequenced-before is happens-before, so whenever we record that an action is sequenced-before another action, we also record that the first action happens before the second action.

The release surface of a write-release is the set containing itself. The release surface of a write-relaxed is the set of fence-releases that are sequenced-before it. The release surface of a read-relaxed is the union of the release-surfaces of the leading modification set of the target.

When we say an action acquires from another action, we mean that the second action and all operations that happen before it are recorded as happening before the first action. When we say an atomic-acquire acquires from a release surface, we mean that it acquires from each operation in the release surface.

A read-acquire acquires from the union of the release-surfaces of the leading modification set of the target. A fence-acquire acquires from the union of the release surfaces of the reads-relaxed that are sequenced before itself.

In order to handle the constraints caused by forking and joining, threads also keep around a set of *previous actions*. Those are actions which would happen before the next action that the thread executes were it to execute an action.

When a thread takes a fork step, the forked thread's previous actions are the same as the forking thread's previous actions at that point. When a thread takes a join step, the joining thread's previous actions is extended with the terminated thread's previous actions. When a thread executes an action, that action acquires from all actions in the thread's set of previous actions that aren't sequenced before the action. The thread's set of previous actions is also extended with the new action.

5.6 Modification Order

Actions *observe* writes. Actions observe writes via happens before, and they observe writes via sequential consistency. The writes observed by an action via an order are all writes that occur before the action within that order, as well as all writes read from by all reads that occur before itself within that order.

When a write is executed, that write is recorded as being modification ordered after all writes that it observes via happens-before. If that write is appended to an existing sequence, then it and all writes observed via happens-before are also recorded as being modification ordered before all writes that were previously modification ordered after the previous write in the sequence.

When a read is executed, the target is recorded as being modification ordered after all writes that the read observes via happens before. Additionally, all writes that the read observes via happens before are recorded as being modification-ordered before all writes that are modification-ordered after the read's target.

5.7 Coherence Order

Coherence order is an extension of modification order that includes reads, so whenever we say that a write is modification-ordered before another write, we also say that the first write is coherence-ordered before the second write. Additionally, if a write is coherence-ordered before another write, then the first write's reads are coherence-ordered before the second write and its reads if it has any, and the second write is coherence-ordered after the first write and its reads if it has any.

5.8 Sequential Consistency

Sequential consistency is consistent with coherence order. Only sequentially consistent actions participate, however. To keep things simple, we define some SC bookkeeping sets. Only atomics maintain these bookkeeping sets.

The SC happens-before set of an atomic is the set of SC fences that happen before itself. The SC happens-before coherence-ordered-before set of an operation is the union of the SC happens-before sets of all operations that are coherence-ordered-before itself. The SC happens-before coherence-ordered-before happens-before set of an action is the union of the SC happens-before coherence-ordered-before sets of the actions that happen before itself.

The SC coherence-ordered-before set of an operation is the set of SC operations that are coherence-ordered-before itself. The SC coherence-ordered-before happens-before set of an action is the union of the SC coherence-ordered-before sets of the operations that happen before itself.

The SC sequenced-before set of an action is the set of SC atomics which are sequenced before itself. The SC sequenced-before happens-before set of an action is the union of the SC sequenced-before sets of all atomics that happen before itself. The SC sequenced-before happens-before sequenced-before set of an action is the union of the SC sequenced-before happens-before sets of all atomics that are sequenced before itself.

When executing an SC operation, it is recorded as being SC-ordered after all SC atomics in the operation's SC happens-before coherence-ordered-before, SC coherence-ordered-before and SC sequenced-before happens-before sequenced-before sets. When executing an SC fence, it is recorded as being SC-ordered after all SC atomics in the fence's SC happens-before coherence-ordered-before happens-before, SC coherence-ordered-before happens-before, and SC sequenced-before happens-before sequenced-before sets.

When a source operation becomes coherence-ordered before a destination operation, some updates to the SC ordering and sets occur. The SC happens-before and SC happens-before coherence-ordered-before sets of the source operation are imported into the target's SC happens-before coherence-ordered-before set. If the source is an SC operation, then the source is added to the target's SC coherence-ordered-before set. The SC happens-before coherence-ordered-before happens-before and SC happens-before-coherence-ordered-before sets of the source are imported into the SC happens-before coherence-ordered-before happens-before sets of the atomics that happen after the destination. The SC coherence-ordered-before set of the source is imported into the SC coherence-ordered-before happens-before sets of the atomics that happen after the destination.

When an SC atomic is added to an SC operation's SC happens-before coherence-ordered-before set or SC coherence-ordered-before set, it is recorded as being SC-ordered before the SC operation. When an SC atomic is added to an SC fence's SC happens-before coherence-ordered-before happens-before set or SC coherence-ordered-before happens-before set, it is recorded as being SC-ordered before the SC fence.

5.9 Race Conditions

When a non-atomic read is executed, it must happen after all non-atomic writes to the same variable. Otherwise, we record the read as *racing* any non-atomic writes to the same variable that do not happen before the read. When a non-atomic write is executed, it must happen after all non-atomic operations on the same variable. Otherwise, we record the write as *racing* any non-atomic operations to the same variable that do not happen-before the write.

If we record that any operation is racing with another operation, then we have found a race condition, and thus a bug.

5.10 Init Writes

The first operation on an atomic variable may be a non-atomic write, known as a write-init. All other operations on that variable must be atomic.

If a variable was initialized with a non-atomic write, then all other operations on that variable must happen after the write-init. Otherwise, the write-init is *racing* with any atomic operations which do not happen after itself.

5.11 Mutexes

Mutexes are objects in the memory model which can be acquired, owned, and released. Only one thread may own a mutex at a time. When a thread performs a mutex lock action on a mutex, they acquire the mutex and now own the mutex until they perform an unlock action on that mutex. A lock action acquires from the previous release action on that mutex. When a thread performs an unlock action on a mutex, they release the mutex and no longer own the mutex. If a thread's next step would be to perform a lock action on a mutex which is already owned, then that thread is blocked and cannot take a step until the mutex becomes unowned.

5.12 Taking a Step

We start with a single unblocked thread. This would be the main thread of the program. At each step, a thread which is not blocked takes a step.

If no thread is *active*, then some unblocked thread is chosen to take a step. If a thread is *active*, then that thread takes a step.

The possible steps available to a thread are initialization, fork, join, read, write, atomic read, atomic write, atomic read-modify-write, atomic fence, lock, unlock, and terminate. A thread always performs an initialization step as the first step, and a terminate step as the last step. A thread never takes an initialization step after the first step, and never takes a terminate step before the last step.

If a thread takes an initialize, fork, join, read, write, atomic write, atomic fence, lock, or unlock step, it is marked as the *active* thread. The exception to this rule is when the thread is blocked and cannot take another step. If a thread takes a step that doesn't result in the thread being marked as *active*, then the thread is cleared of *active* status.

If a thread takes a lock step, then any thread that wants to take a lock step on the same mutex is *blocked* until the thread that locked the mutex takes an unlock step on that mutex.

When a thread takes a fork step, a new thread is created and put in the *fork set*. When a thread is being chosen to take a step due to no thread being active, threads are first taken from the fork set before they are taken from the general pool of unblocked threads.

When a thread wants to take a join step, that thread is blocked until and unless the thread that it is joining has taken its terminate step.

5.13 Choosing What Writes to Read From

The write read from by a read is chosen from the set of writes which are not modification-ordered before the writes observed via happens-before by the read. If the read is sequentially consistent, then the write read from by the read is also restricted to the set of writes which are not modification-ordered before the writes observed by the read via SC.

Chapter 6

The Fuzzer as an Implementation of the C++ Memory Model

The fuzzer is basically a direct implementation of the operational semantics model of the C++ memory model, with optimizations and adjustments for dealing with the details of real programs.

The decision procedures that fill in the choice points of the operational semantics above and the bookkeeping that supports them are what make the fuzzer smart.

6.1 Graph Searches

When performing graph searches through the memory model graph, we want to minimize the number of actions that we search through. We especially would like to avoid having these searches grow with the size of the execution.

6.1.1 Simple Ordering Query

The simple ordering query answers the question of whether or not action a is ordered before action b. For answering that question, we use a simple forward search through the memory model subgraph that carries the ordering. We use a forward search (usually depth-first) to take advantage of the fact that actions a and b will typically be near the terminal nodes of the memory model subgraphs, though exactly how near will depend on the program being tested.

6.1.2 Uncovering Search

The uncovering search is a way of searching for the most immediate predecessors of an action that satisfy certain criteria within a DAG. These would be the set of actions that satisfy the criteria, are ordered before the starting actions of the search, and are not ordered before any other actions meeting the criteria that are ordered before the starting actions of the search.

We take our starting actions and put them in a queue. We take the actions which are search-ordered after our starting actions and put them in a map, associating each of those actions with the number of predecessors in the search order that are either starting actions or search-ordered after those starting actions.

Until the queue is empty, we pull an action off the queue, check if it meets the criteria, and if it does, we look at its predecessors and decrement the count of un-visited successors. If any of those predecessors has a count of 0 after having it decremented, then that predecessor is 'uncovered'. When this happens, the uncovered action is taken out of the map, and put on to the queue. The predecessors of the uncovered action are also put into the map (if they're not already in the map) with a count that corresponds to the number of successors that are ordered before the starting actions.

Note that the successors are only updated when we don't find what we're looking for. This causes the search to ignore any action which is ordered before the actions that we're looking for. We only use this search to find predecessors because of the simple ordering query implied by the constraint on which predecessors are put into the map. This query can potentially touch all actions ordered after the starting actions, and as such this search is only bounded when going backwards from a starting actions that are within some bounded distance of the terminal actions of the graph.

6.1.3 Subtracting Search

The subtracting search is a counterpart to the uncovering search that finds the most immediate successors to a set of actions that meet certain criteria. These would be actions that satisfy the criteria, are ordered after the starting actions of the search, and are not ordered after any other actions meeting the criteria that are ordered after the starting actions of the search.

A subtracting search is a depth-first search that adds or subtracts actions from sets based on whether or not a particular kind of action is in the path from the starting action to the currently visited action. If the particular kind of action is not in the path, then the add operation is performed. If the particular kind of action is in the path, then the subtract operation is performed. An action that has been visited with a subtract operation does not need to be visited again by either operation. An action that has been visited by an add operation may be visited by a subtract operation later if we find a path to that action that passes through the kind of action we're looking for.

By recording the actions meeting the criteria in the add operation, and removing record of them in the subtract operation, the search ends up with a set of the kinds of actions meeting the criteria where no action in the set is ordered before or after any other action in the set.

The search will touch the actions ordered after the starting actions at most twice, keeping the cost bounded if the starting actions are within some bounded distance of the terminal actions of the graph.

6.2 Lazy Evaluation of the Reads-From Set

We need to be able to determine what set of writes a read may read from. We can't know when a variable will no longer be used, so any sort of eager tracking of what writes can be read from at various points in the execution would incur an expense that is at least $O(n^2)$ in the general case. As such, we compute the set of writes that a read may read from lazily.

Tracking the observed writes in the memory model graph is easy. When searching backwards through an order from an action, all writes encountered are observed, and all writes read from by all reads encountered are observed. In practice, it is useful to restrict the search to those reads and writes to a specific variable. The writes observed via an order are found using an uncovering search from the action backwards over the order.

We only need to generate the set of writes a read may read from for the purpose of making the decision about what write a read will read from. There are no other uses of this set. The set, as defined in the operational semantics, is the set of writes that are not modification-ordered before any write observed by the read. As such, if we perform an uncovering search through the modification order starting from all of the terminal nodes and stopping at the writes observed by the read, we will visit exactly the writes that belong in the set.

6.3 Happens-Before

We need to be able to enumerate the set of writes observed via happens-before for any atomic operation. Reads need to know the set of writes observed via happens-before in order to determine what writes they may read from. Writes need to know the set of writes observed via happens-before in order to determine what writes are modification-ordered before themselves.

We also need the happens-before graph to support the SC graph, since strongly-happens-before implies SC between two SC operations.

Last but not least, we need to be able to check non-atomic operations for race conditions.

Before we continue, we will define some terms.

A *joining* action is an action that happens immediately after one or more actions from another thread. The first action of a thread, the action immediately after a thread join, and acquire atomics that acquire from a release action are all joining actions.

A *branching* action is an action that happens immediately before one or more actions from another thread. The last action of a thread, the action immediately before a thread fork, and release atomics that are acquired from are all branching actions.

Within threads, most actions are neither branching nor joining. Additionally, most actions will be non-atomics, which are not at all interesting for answering the second of our two questions. When an action is taken, all actions that happen before it are known already.

One well-known technique for determining if one operation happens-before another is clock vectors [24]. Unfortunately, the size of the clock vectors in a program grows with the number of threads, so using clock vectors for happens-before will result in a bookkeeping cost of $O(n^2)$, in general. For that reason, we don't want every action to be keeping track of a vector that knows about a bunch of other threads. However, the advantage of doing a simple comparison between thread indices to determine whether or not action a happens before action b cannot be ignored. In a program with hundreds or even thousands of non-atomic operations between branching or joining actions, the ability to directly compare two actions in that thread without traversing all actions that may lie in-between can and does result in an enormous savings in time. So in order to maintain both performance and scalability, we'd like to form a graph between threads, but keep track of action indices within threads.

To this end, each action keeps track of the thread that contains it, and the index of itself within that thread. This way, given two actions within the same thread, we can easily determine which action happens before the other. Given two actions in general, we can also easily determine whether or not they are in the same thread, and thus whether or not their indices are directly comparable.

Each action also keeps track of the next branching action and previous joining action within the same thread, if they exist. Each action also keeps track of the next operation and previous operation within the same thread, if they exist. This helps with the enumeration of observed writes.

All branching actions keep track of the actions in other threads that immediately succeed them in the happens-before graph, and all joining actions keep track of the actions in other threads that immediately precede them in the happens-before graph.

To check for race conditions, we essentially need to ask a series of questions of the form 'Given actions a and b, does action a happen before action b?'. These questions are answered with a simple ordering query, except that only action a and the actions in other threads after the branching actions sequenced after any action we visit that doesn't happen before action b are visited. Because of the index comparison, even though only a small subset of actions are actually visited, we check all actions that are sequenced after any action we visit as well as the actions we visit. This will potentially check all actions that happen after action a, which is what we want.

Writes observed via happens-before are enumerated using an uncovering search. The uncovering search uses modified successor () and predecessor () methods, to accommodate and exploit the unusual representation of happens-before and avoid visiting actions that are not operations whenever possible, minimizing the search.

6.4 Modification Order and Coherence-Ordered-Before

Modification order is recorded directly using the rules given in the operational semantics. The only change is that the sets of observed writes are smaller in the fuzzer's implementation, in general, so not all of the transitive edges are directly recorded. This is an optimization to save memory and time. Coherence order is derived from modification order and the reads-from relation on demand.

6.5 Sequential Consistency

We need to be able to enumerate the set of writes observed via SC for any SC read. SC reads need to know the set of writes observed via SC in order to determine what writes they may read from.

We keep track of the SC bookkeeping sets as described in the operational semantics. We do so incrementally, so the SC bookkeeping sets of an action are determined by the SC bookkeeping sets of actions which happen before or are coherence-ordered before itself. These sets are minimized to not contain SC operations that are relatively SC-ordered when they're created for a new action. Given two SC atomics, the one that is SC-ordered after the other will be preferred as a member of these sets.

The bookkeeping sets are divided into two families.

The first family tracks strongly-happens-before and its precursors. These would be the sequenced-before (SB), sequenced-before happens-before (SB-HB), and sequenced-before happens-before sequenced-before (SB-HB-SB) sets. These exist for each atomic, and track the SC atomics which are ordered before the atomics via the composite ordering after which the sets are named. When applying an SC atomic, SC edges are created from the SC atomics in the SC-HB-SC to the new SC atomic.

The second family supports the creation of SC edges that arise from the interaction between SC and coherence-order. These would be the happens-before (HB), coherence-ordered-before (COB), happens-before coherence-ordered-before (HB-COB), coherence-ordered-before happens-before (COB-HB), and happens-before coherence-ordered-before happens-before (HB-COB-HB) sets. Fences are SC ordered after the atomics in their COB-HB and HB-COB-HB sets. Operations are SC ordered after the atomics in their COB and HB-COB sets. They exist for each atomic, and track the SC atomics which are ordered before the atomic via the composite ordering after which the sets are names, with the following exceptions. SC operations do not pass their HB or HB-COB sets into the COB set of operations that are coherence-ordered after them. This is because any SC operations that would be coherence-ordered after them will also be SC-ordered after them, and any fences that are coherence-order happens-before ordered after them would also be SC-ordered after them. Those SC edges along with the edges that already exist between the members of the HB and HB-COB sets of the SC operation transitively imply the SC edges that would have been added were those sets passed on.

When edges are added into the COB graph, the SC graph may need to be updated, and some atomics may need their SC sets updated. There may be multiple edges added at once, but they will always end at the same destination action in a given update.

The SC edge predecessors are the SC sources of the new edges and the HB-COB and COB sets of the non-SC sources of the new edges. If the destination of the new edge is not SC, the SC edge successors are the SC operations which are coherence-ordered after the destination, and the SC fences which are coherence-order happens-before ordered after the destination. If the destination of the new edge is SC, then the SC edge successors are just the destination.

We also keep track of the intermediate atomics that occur between the sources and the SC edge successors. Those intermediate atomics, along with the SC edge successors, require updates to their SC sets so that the fuzzer has all the right information when applying later actions. The COB and HB-COB sets of the intermediate atomics and SC edge successors that are coherence-ordered after the sources need to be extended with the COB and HB-COB sets respectively of the SC edge predecessors enumerated from non-SC sources. The SC sources get added directly to the COB sets of the intermediate atomics and SC edge successors that are coherence-ordered after the sources. Similarly, the COB-HB and HB-COB-HB sets of the intermediate atomics and SC edge successors that are coherence-order happens-before ordered after the sources need to be extended with the COB and HB-COB sets respectively of the SC edge predecessors enumerated from non-SC sources. The SC sources get added directly to the COB-HB sets of the intermediate atomics and SC edge successors that are coherence-order happens-before ordered after the sources. Finally SC edges are recorded between the SC edge predecessors and the SC edge successors.

We use a subtracting search to locate the SC edge successors and enumerate the intermediate atomics.

The writes observed via SC can be easily enumerated using an uncovering search through the SC graph, starting from the observing action.

6.6 The Digest Graph

We observe that, in most real programs, bugs are not caused by all atomic operations leading up to the bug from the beginning of the program's execution. Really, they are caused by only a few atomic operations leading up to the bug. This is really the big idea behind the smarts in this fuzzer.

We assume that programmers tend to think about programs in a way that matches a linear notion of time. This ends up being the sequentially consistent model of concurrency. That would tend to imply that when the behavior of a program violates sequential consistency, bugs are more likely to occur because this would be a behavior that the programmer is less likely to have thought about. Any time there's a cycle in the memory model graph (the union of happens-before, coherence-order, and sequential consistency), we are violating sequentially consistent behavior.

To this end, we group actions into strongly connected components in the memory model graph. We then digest these components into *states* which abstractly describe the current state of the execution. We maintain a component graph to keep the cost of computing new components minimized.

Actions are digested into digest actions. The only information preserved from the original action is the position of the action in the code, and the positions of the members of it's neighbors in the happens-before, coherence-ordered-before, and SC memory model graphs. Digest actions do not record duplicate positions within a given set of neighbors, so if two neighbors of an action that both happened before the action and shared the same position in the code would become only one entry in the digest action's happens-before set, for example.

Digest states are created from the strongly connected components in the memory model graph. They're just collections of the digest actions in those components. No distinguishing information about the threads or variables involved are kept. This way, if the actions occur again with different threads and different variables but the same structure, the component will digest to the same state. Digest also states do not record duplicate digest actions. This means that, if two actions in a component digest to the same digest action, only one digest action is recorded in the digest state for both actions. This actually helps collapse certain patterns into the same state, like components that include a string of read operations from a loop that spins on a flag. Those read operations would generally collapse into a number digest actions in the digest state that depends largely on the number of write operations in the code base that were available to be read from, rather than the number of reads performed. This reduction of state also means that digest states have a maximum size based on the set of digest actions that can be derived from the actions within a single component. This set will be at most as large as the number of actions that can occur within a single component, and would generally be smaller. This will help keep the digest state explosion under control, even in the context of a program that uses a lot of different kinds of concurrent operations on the same data structure.

The digest graph itself is a record of the abstract state machine that the program implements, to the degree that the fuzzer has explored it. The states are the digest states, but the transitions are nontrivial, because there are so many decision points in the process of getting from one state to the next.

Our digest states are always the states that occur right before some thread mentioned by the state is about to perform a read. The program might take many paths after a given action or state is reached, so we differentiate our state transitions based on what read is going to be performed next. The fuzzer distinguishes these reads by the equality of their digests. The first decision point in the transition is the read that will occur next. As such, our graph is a map from states to the read decision point structure.

Once a read has been chosen by the program, the fuzzer needs to decide what write that read will read from. Our read decision point structure is thus a map from reads to write decision point structures.

Once a write has been chosen, the program may end up in a number of different states. Which state is reached will depend on a bunch of other state in the program and the control-flow structure which we don't actually know. The final part of our transition structure, the write decision point structure, is thus a map from writes to collections of destination states.

When the fuzzer is exploring the execution, it extends the digest graph with the states and decision point structures as it discovers the corresponding components and decision points. When enumerating the reads-from set of a new read operation, all writes discovered in that reads-from set are digested and added as possible choices (if they weren't added previously) for the write decision point, even though they may not be picked. These choices end up pointing to empty destination sets, marking parts of the program's behavior that the fuzzer hasn't explored yet.

By assigning weights to the part of the graph, we can bias the selection of which writes to read from, and which threads to step next. We assign a high weight to the empty destination sets. These are the source weights. We propagate those weights backward through the graph, including the transition elements. Each section averages the weights of the next set of elements, and then multiplies with a decay factor to encourage the fuzzer to explore unexplored states greedily.

6.7 Bug Detection

The fuzzer checks for race conditions. A race condition happens when any two operations on the same variable, at least one of which is a write, are not related by happens-before. The exception is atomic variables, for which atomic reads and writes may occur without being related by happens before without issue. An atomic init on an atomic variable should happen before all atomic reads and writes to that variable, however. If it does not, then it is undefined behavior.

To check for race conditions, the fuzzer keeps track of the most recent write to any non-atomic variable. It also keeps track of all reads since that write. When applying a new non-atomic read, the fuzzer checks if the most recent write happens before the new read. If not, there's a race condition. When applying a new non-atomic write, the fuzzer checks if all reads since the most recent write happen before the new write. If not, there's a race condition. If there are no reads since the most recent write, then the fuzzer checks if the most recent write happens before the new write. If not, there's a race condition.

When applying an atomic operation, the fuzzer checks to see if a non-atomic write has been applied to that variable. If so, then that write is an init write, and should happen before the atomic operation. The fuzzer checks for this, and if this check fails, then there is a race condition. The fuzzer checks this any time the init write is in the reads-from set of any atomic read applied to that variable, and any time a write is applied to that variable.

Chapter 7

Instrumentation Strategy

In order to fuzz a program, we have to intercept all the concurrency primitives. These would be all atomic actions, all basic thread actions, all accesses of non-atomic variables that are shared between threads, all mutex actions, and all condition variable actions. We then have to take control of the program's execution so that we can manipulate the schedule, and perform all of those those actions on the program's behalf, according to our fuzzing strategy.

7.1 Compiler Pass Intercept

To intercept all of the basic actions, we decided to write a compiler pass that would modify the program for us. This approach allows us to instrument programs without having to modify their source code at all. The only work we have to do to instrument a program is modify the build scripts.

In addition to the concurrency primitives, CDS fuzzer intercepts the main function. The pass renames main so that the loader will use the main function provided by the fuzzer itself, and replaces all atomic operations with function calls to functions exported by the fuzzer as a shared library. The other operations are already implemented as function calls into the pthread library, so we don't need to do anything for those in the pass.

7.2 The Shared Library Approach

The fuzzer itself is a shared library that we preload. The fuzzer exports symbols for all the operations listed above. The fuzzer also captures the thread, mutex, and condition variable actions by implementing the pthread API. The fuzzer's shared object file is preloaded when running the target application so that it can intercept all of these shared library calls.

7.3 Threads as real Threads

When creating threads, we had to decide if we were going to use real threads or lightweight user-space threads. User-space threads have a performance advantage, but cannot handle thread-local storage. It seemed likely to the author that real programs would use thread-local storage, so it was decided to use real threads.

To take control of these threads, some of the instrumented functions will yield the application thread to the fuzzer's scheduling thread. When we create a thread, we end up wrapping the function the program wants to execute with a function that contains the machinery to perform these yields. The scheduler only lets one of the program's threads run at a time, and only until it yields to the scheduler. The scheduler picks which thread runs next based on the fuzzing strategy.

7.4 Forking for Execution Isolation

To have a smart fuzzer, we need to be able to run the execution many times while retaining some information about each run. This means that some part of the fuzzer needs to be persistent across executions of the program. This also means that the parts that aren't persistent need to be reset after every execution. Since we also want to support running multiple executions simultaneously, we end up using process forking as our mechanism for starting and resetting executions.

The entry point of the program is main, and main is provided by the fuzzer. The program's main function has been renamed and will be called by the fuzzer when appropriate. This means that the fuzzer has a chance to perform some setup before the program's entry point is called. The fuzzer processes command-line arguments, and sets up a shared memory region for the digest graph before starting any executions. To run executions, the fuzzer forks, and the child process starts up the execution. The parent process doesn't ever perform an execution of the program. It just manages the child processes and shared resources. This way, the parent process is always in the start state of an execution, and can keep forking off new executions indefinitely.

Chapter 8

Evaluation

Significant problems were found in our choice of instrumentation strategy. As such, we are able to provide a limited evaluation of the fuzzer's implementation of the memory model, but are not able to provide an evaluation against real programs, nor are we able to meaningfully compare our tool against other tools.

8.1 Proof of Concept

The demo programs that we use to demonstrate that this fuzzer is actually a concurrent fuzzer capable of finding bugs in programs using C++ atomics are taken from the benchmarks used by CDSChecker [27, 28]. The exception is the SPSC queue shown earlier, which was inspired by the CDSChecker benchmark, but significantly simplified so that it would be suitable for being presented as the motivating example earlier. All other programs are essentially unmodified.

SPSC Queue: This is the example from earlier. Both the version shown and the fixed version are run.

Chase-Lev Deque: An implementation of the Chase-Lev deque [21] using many relaxed operations. There is a bug in this program that occurs when the steal operation overlaps with a push operation that resizes the deque. Both the buggy and fixed versions are run. One thread pushes 3 values and takes 2 off of the queue, while another thread concurrently steals a value. Since the initial size is 2, pushing 3 values can trigger a resize.

Barrier: A simple synchronizing barrier [10]. All but the last thread spin on a global flag waiting for the last thread to reach the barrier. A writer thread writes to a shared non-atomic variable, and then waits on the barrier. A number of readers wait on the barrier, and then read the shared non-atomic variable. In our case, we've configured the number of readers to be 10, making it quite a bit larger than the other demo programs. CDSChcecker configures the number of readers to be 1 for comparison.

MCS Lock This is an implementation of the MCS Lock [9, 25]. Two threads are created. Each acquires and releases the lock twice. One threads loads from a non-atomic variable in the first critical section, and stores to the non-atomic variable in the second critical section. The other threads stores to the non-atomic variable in the first critical section, and loads from the non-atomic variable in the second critical section.

All programs were run with a 30 second time budget and a concurrency level of 1 on an Intel(R) Xeon(R) CPU E3-1245 v6 running Ubuntu 18.04. We record the number of executions completed, and note the bugs found by the fuzzer.

Program	Number of Executions	Found Bug?
SPSC Queue (Buggy)	6026	Yes
SPSC Queue (Fixed)	4735	No
Chase-Lev Deque (Buggy)	1603	Yes
Chase-Lev Deque (Fixed)	1737	No
Barrier	393	No
MCS Lock	1773	No

Table 8.1: Results of running the demo programs.

As we can see in table 8.1, the fuzzer completes a number of executions within the time budget for each of these programs. There are enough executions to explore a significant portion of the programs' potential behaviors, given the programs' small size. It is worth noting that we don't really care about execution count with this fuzzer so much as we care about the number of different behaviors that are explored. A long running program that performs many of the same kind of operation on a concurrent data structure would be just as good as a small program that executes only a few, as the fuzzer is learning from each individual data structure operation. These programs all execute less than a dozen operations against their concurrent data structures, so more executions are required to explore all possible behaviors of their data structures.

The fuzzer also found the bugs in the programs that contained them, and did not find bugs in the programs that did not contain them. This shows that the fuzzer can actually find bugs in a useful manner.

SPSC Queue (both versions) MCS Lock were modified to take an argument for the number of iterations run. The programs are the same as before, except that each thread will perform the whole set of actions described above the specified number of times. The programs were then run with iteration counts from 10 to 100 in 10 iteration intervals. The other parameters were the same as above. The results can be seen in figure 8.1 and figure 8.2.

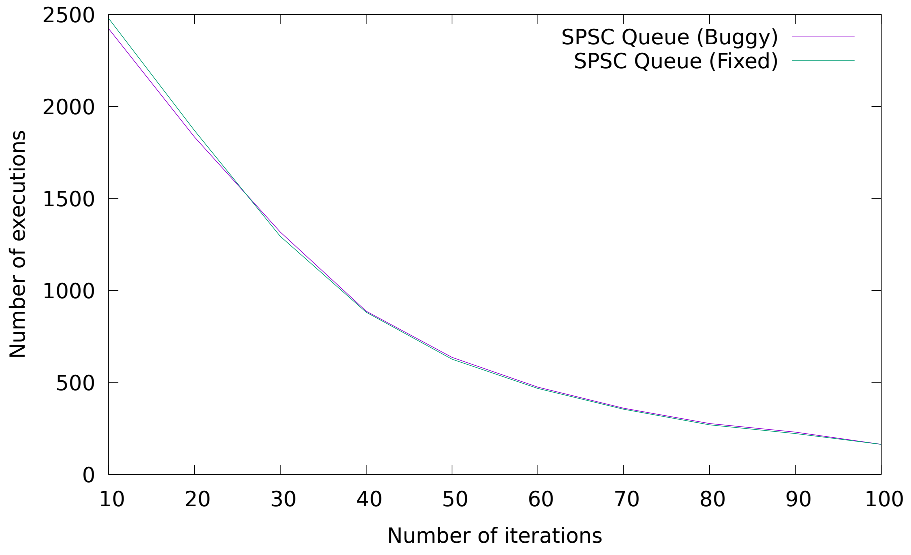

Figure 8.1: Scaling of executions for the SPSC Queue demo

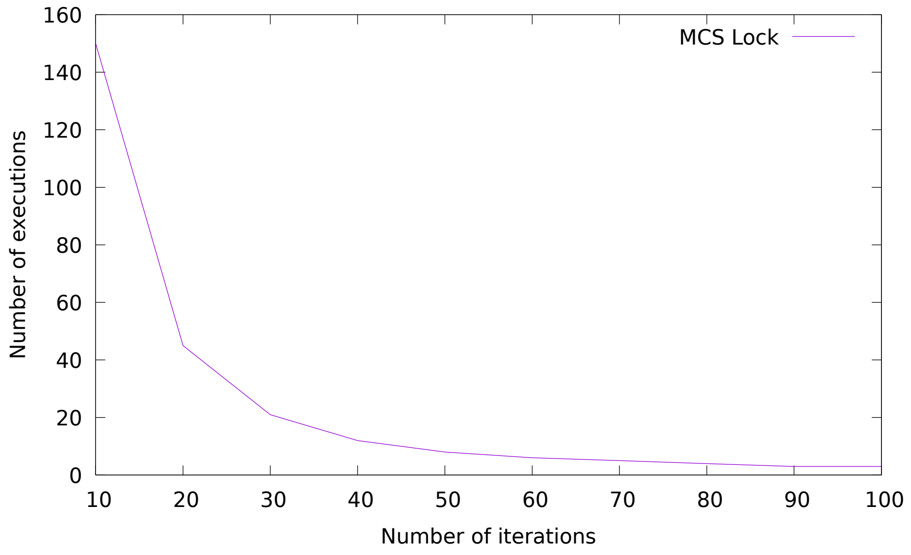

Figure 8.2: Scaling of executions for the MCS Lock demo

The results both show a sublinear decline in executions as iterations increases. This is likely due to the reduced significant overhead of starting up new processes. Larger executions have less of that overhead per unit of code fuzzed, so as we make the individual executions larger, the fuzzer gets proportionately better at executing them. What we do not see is a superlinear decline in the number of executions as iterations increases, meaning that the memory model logic in the fuzzer isn't scaling poorly as executions get larger.

8.2 Real-World Programs

Approximately one year after we began this project, we started to search for benchmarks to use to evaluate our fuzzer. We ended up searching through thousands of projects on GitHub over the course of an entire quarter.

There were extremely large projects that used atomics, like web browsers and operating systems. There were also core libraries and toolkits that used atomics, like PulseAudio [2] and OpenJDK openjdk. There were programs that weren't self-contained at all, like those that used the network extensively. There are libraries that use atomics, like Boost [3] and Folly [13]. These are all good examples of real code that uses atomics like we wanted, but are not readily adapted for use as a benchmark in an academic setting.

Then there were programs that were easier to compile to run, but did not use atomics in the way that we wanted. There were programs that were little more than test drivers for concurrent data structure implementations. There were programs that used atomics for trivial purposes, like atomic counters or flags.

In the end, we were only able to find four programs that seemed suitable as benchmarks.

8.2.1 GDAX Orderbook

GDAX Orderbook [5] is a library that maintains a highly available local copy of the GDAX orderbook that is continuously synchronized through the GDAX cryptotrading API. Little more than glue between LibCDS and the GDAX Cryptotrading API, the demo program for this library continuously scans over the orderbook with multiple threads to look for changes in the market.

GDAX Orderbook uses lock-free data structures from LibCDS. This isn't necessarily a problem for evaluating our fuzzer on applications, but it doesn't align with our vision of the target program.

The code has bugs in it that are easy to spot just by looking through the code. This makes it a poor candidate for fuzzing, since you would generally use a fuzzer to find bugs that couldn't be found by means that required less investment - like code review.

The code is unmaintained, and was last updated over a year ago. Many of its dependencies have been updated since then, and it takes some work to get this code base to compile on a modern system.

8.2.2 Silo

Silo [34] is a highly concurrent transactional database inspired by Masstree [23].

Silo creates threads and performs atomic actions during static initialization. While we can introduce hacks that handle some cases, we cannot support this behavior in the general case.

Silo uses jemalloc by default. "jemalloc is a general purpose malloc(3) implementation that emphasizes fragmentation avoidance and scalable concurrency support." [17] jemalloc uses locks internally. Unfortunately, because of the way that linking works, when an instrumented program uses an alternate malloc implementation as a shared library, our fuzzer also ends up using that malloc implementation. Because of this sharing, there is no good way to properly handle jemalloc, since we can't fuzz ourselves, but we also cannot easily determine if a lock call is coming from jemalloc versus some other source. It would also be dangerous to only instrument lock and unlock calls in jemalloc that come from the application, as they may be attempting to interact with lock and unlock calls from within the fuzzer itself. As such, we cannot actually fuzz any program that uses a custom malloc implementation that contains locks with our approach to instrumentation. While in this case we could easily compile the project against a less problematic malloc implementation, the fact that we cannot fuzz programs using concurrent malloc implementations is a real problem, as jemalloc isn't the only concurrent malloc library out there, and they represent a category of concurrent code that really needs to be correct. Our tool is ostensibly designed to help developers of such code find bugs in their code so that they can me more confident about its correctness, and because of the instrumentation strategy we've chosen, a whole category of such code simply cannot benefit from our work.

Silo is also unmaintained. It was last updated in 2014. C++ atomics were still relatively new, and not much concurrent code had been ported from the old volatile-based idioms to the new standardized ones. Silo included code that made heavy use of the old volatiles-as-atomics idioms from the pre-C++11 era.

8.2.3 Mabain

Mabain [11] is a highly concurrent key-value store implementation that supports access from multiple processes.

Mabain forks off child processes. This does not violate any assumption that we had made, nor does it break the fuzzer. However, we cannot fuzz the interactions between processes effectively, because each process is seen as a separate entity, independent from other processes.

8.2.4 Iris

Iris [37] is a concurrent logging library that uses a lock-free queue to communicate between the client threads and the background thread doing the real work.

Iris creates threads and performs atomic actions during static initialization. Like with Silo, we can introduce hacks that handle some cases, but we cannot support this behavior in the general case.

8.2.5 Prevalence of Target Applications

Of the four benchmarks we chose, only two are actively maintained and also implement their own data structures. It would seem that, if one were to really run benchmarks that were representative of the code that is out there, they should be prepared to run programs that do not readily fit the mold for a self-contained benchmark. Concurrent libraries, web browsers particularly seem like good candidates, since they can be most easily adapted to running in a self-contained manner, though this requires some significant effort. These codebases would be even worse about using language features in ways that are pathological to our instrumentation strategy.

8.3 Linker Issues

In addition to real programs largely failing to satisfy our criteria as suitable benchmarks, we also ran into issues with our tools.

While getting a compiler pass to rename the main function of a program is easy, getting the linker to actually resolve the symbol in the shared library against the renamed function is another story. The behavior of the compiler in this regard changes from environment to environment, and version to version. In one environment, you will be required to preload the binary like it is a shared library in order for the symbol to resolve. In another, this will be unnecessary. In one environment, you can resolve the symbol at load-time. In another, you have to resolve it at run-time using the dynamic loader. In yet another, resolution of this symbol will fail entirely. Moreover, compiling different programs within the same environment might exhibit different linker behaviors. The exact cause of all of this variation is still unknown, but the variation itself strongly suggests that resolving symbols in libraries to symbols in binaries is not a well-supported feature.

This linker issue had to be hacked around in the demo programs in order to run them. This involved modification of the programs' source code to pass information into the fuzzer that we don't intend to be necessary in a finished version of the tool.

8.4 Limitations of the Evaluation

Based on the problems we ran into, we need to rethink the instrumentation strategy. The author suspects that there are significant improvements that could be made to the heuristic that guides the executions, just based on work that has already been done in the field. Also, the implementation has received virtually no optimization. Between these things, any interesting metrics we could measure must be taken with a gain of salt. Because of this, it would be unfair to compare this tool to other tools, since the numbers could all change significantly when the above issues are addressed.

Chapter 9

Additional Behaviors Observed in the Real World

In addition to violated assumptions, there were a few details about how real programs work that we had to take extra care to properly support.

9.1 Static Initialization

Statically initialized variables may not be written to explicitly by the program before they're read from. This is because the compiler just compiles the constants into the appropriate data section, and it is loaded already initialized. This also means that we needed to support atomic reads from variables that hadn't been explicitly initialized or written to.

Special writes were used for this purpose. Every time the fuzzer found a new variable, a special write was created to represent the static value that the variable had when the program was loaded into memory. This special write was assumed to happen before and be modification ordered before all other writes to the variable.

9.2 Unbounded Loops Without Atomics

A common novice move when writing concurrent software is to use non-atomics like they are atomics. This leads to patterns like busy-wait loops that exit only when the only operation in the loop - a non-atomic load - returns a specific value. Besides being a guaranteed race condition that we'd like to report, this is a trap for any fuzzer which isn't ready for it. Our fuzzer would get stuck in loops like this, because the threads would not yield to the scheduler unless they executed an atomic read, a lock acquire, or a condition variable wait. We modified the fuzzer instrumentation to yield to the scheduler after a certain number of non-atomics had been executed since the last yield to fix this problem.

9.3 Atomic Initialization as a Non-Atomic Write

Atomic initialization is implemented with a non-atomic write. This means that when we perform an atomic read, we have to check to see if the variable was initialized via a non-atomic operation. Moreover, that non-atomic write then also needs to be able to participate in the modification order for that atomic variable.

This isn't terribly difficult, but it needs to be done. In our case, we end up upgrading the non-atomic write to an intermediate write type that can participate in the modification order graph, but isn't an atomic operation. We also ensure that all atomic operations on that variable happen after the initialization write.

9.4 Memory Re-use

Because programs use real memory, memory can be re-used. This means that the fuzzer might see a variable that was used as an atomic get used as a non-atomic, and vice versa. You might also see multiple atomic initializations on the same variable.

Nothing special was needed to support this, except allow for the possibility. Essentially, the fuzzer cannot assume that there will be a single atomic initialization for each variable, or that atomic initialization will be the only non-atomic operations performed on a variable.

9.5 Forking and Shared Memory

Mabain would fork off child processes and sometimes communicate with them via shared memory. This did not break our fuzzer as the scheduler was forked as well, but we didn't have any mechanism to manipulate the relative timing of accesses to the shared memory between processes.

It might be nice to separate the scheduler from the target program's process so that we could treat forked processes in a manner similar to threads and fuzz the whole process tree as a unit.

9.6 Limitations of the Linker

Because the load-time linker will always resolve all uses of a symbol to one instance, some code is inevitably going to be shared between the target program and the instrumentation code unless extreme care is taken to avoid it. In our case, we made use of C++ standard libraries, and some of the code internal to those libraries was provided by the instrumented program, in practice. This meant that the fuzzer ended up calling into itself sometimes. To fix this, we had to add a flag that indicated when the fuzzer's core scheduler thread or a thread from the target program was running. The fuzzer would switch the behavior of instrumented calls depending on the value of the flag, so that we wouldn't end up erroneously instrumenting ourselves.

Chapter 10

Conclusions and Future Work

Additional work is required before an effective tool for finding concurrency bugs in the C++ memory model can be produced. The remainder of this section is written for the benefit of those brave souls who would continue this work.

10.1 Rethinking the Instrumentation Strategy

Based on what we observed, there are things that programs do that anyone writing a concurrent fuzzer for C++ programs should be aware of.

- Programs can create, use, and destroy threads, mutexes, and condition variables at any time.
- Programs can perform atomic actions at any time.
- Programs will sometimes use thread-local storage.

- Compilers will statically initialize variables, so the fuzzer can't rely on a write always being executed before a read to a given variable.
- Memory management implementations will re-use memory, so don't assume that a variable will always be the same thing. Prepare to see a mixture of atomic and non-atomic actions on the same variable, including multiple atomic initializations.
- Compilers compile atomic initialization to non-atomic writes, or even to a constant loaded into the data region where there's no write at all in the case of static initialization.
- Some programs will fork and communicate via shared memory regions.
- Programs will replace core libraries with alternate implementations, making linking against them tricky. Either write the instrumentation code in such a way that it uses *zero* external libraries, or don't put it in a shared library linked against the main program at all. A combination of both will likely be required in order to write effective and maintainable software.
- Resolving symbols in a library against exported symbols in a binary appears to be tricky business. The author generally recommends avoiding this, as it would most likely require custom linking scripts to properly do reliably.

Because any of the operations the fuzzer cares about can happen long before main is called and things are properly set up, we need a strategy that has the fuzzer up and running before any part of the target program is running. Because the target program can swap out libraries that the fuzzer would ordinarily depend on, and because those libraries might also be the code we want to fuzz, we need the scheduler and the target program to be in separate processes. Because fuzzer parameters like level of concurrency need to be processed before any execution is started, the fuzzer needs to be able to process its command-line arguments before the instrumented program is ever run. This also implies putting the fuzzer core and the target program in separate processes. Having the scheduler and the target program in separate processes also makes it easier to handle programs that fork and use locks and such in shared memory. The locks will ultimately be handled inside of the scheduler, so no special handling will be required to make that work, so long as forking is handled in a way that resembles thread creation.

Putting the scheduler and the target program in separate processes means that the target program needs to have the operations and function calls of interest instrumented with a very thin layer of code that communicates with and even yields to the scheduler process. Generally, this would be achieved via shared memory. Since the scheduler is in a separate process from the instrumented program, there's no reason that it can't just be a thread within the the fuzzer core for a fuzzer which would run multiple executions at once.

Essentially, the fuzzer core process starts up. It initializes everything that it needs, including shared memory to communicate with the actual instrumented programs. Then the fuzzer forks and exec's the instrumented program. The instrumented program should be able to establish a connection to the shared memory as soon as any part of the instrumentation is executed. This step is the hard part, because the instrumentation can't assume that much of anything has been set up, and it doesn't have any real means of getting information from the scheduler yet, like where the shared memory is located. Even the memory allocator isn't guaranteed to be set up properly. Communication between the scheduler and the instrumented program would have to rely only on mechanisms which exist and work even when nothing inside of the program has been properly set up yet. System calls to access the environment or open files with fixed names are likely the only portable way.

Once a connection is established, however, things are much easier. Communicating with the fuzzer can be done with a simple event queue and the ability to block pending a decision from the scheduler. Even though control is regularly being passed across a process boundary, there shouldn't really be any real performance penalty as compared to the approach used in my current implementation, since my implementation is already passing control between threads.

Exactly how the instrumentation is injected into the program is ultimately up to the tool author. There are two approaches that are reasonable. Despite the difficulties that we ran into, if the instrumentation to communicate with the scheduler is simple enough, the shared library preloading approach would work. Alternately, it might be easier to just modify the compiler pass to generate all the instrumentation code directly, avoiding any potential linking and code-sharing issues entirely.

10.2 Supporting Extremely Large Programs

As of now, the fuzzer accumulates a memory model graph for the entire duration of an execution. This ultimately limits the size of an execution, since an execution that runs for too long will end up consuming all of the available memory. In order to run the fuzzer on truly long-running programs, the memory model implementation will need to be modified to keep the amount of memory used bounded somehow.

There are challenges to doing this. Variables that were last touched a long time ago might be touched again in the future, and the fuzzer needs to know what values are available to be read from by future reads. Essentially, any bounding of the memory model graph needs to be aware of what information might be required in the future. If it is unreasonable to simply prune off any graph structure that might not be needed anymore and a more aggressive approach becomes required, some accommodations or approximations need to be made for future reads.

These modifications will likely be necessary if one wants to run this tool on applications like web browsers.

Bibliography

[1] `http://libcds.sourceforge.net/doc/cds-api/index.html`. Accessed: 2019-11-20.

[2] `https://www.freedesktop.org/wiki/Software/PulseAudio`. Accessed: 2019-11-20.

[3] `https://www.boost.org/`. Accessed: 2019-11-20.

[4] P. Abdulla, M. F. Atig, B. Jonsson, and P. Ngo. Optimal stateless model checking under the release-acquire semantics. *Proceedings of the ACM on Programming Languages*, 2:1–29, 10 2018.

[5] F. E. Aumson. `https://github.com/feuGeneA/gdax-orderbook-hpp`. Accessed: 2019-11-04.

[6] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, and V. Štill. Divine 3.0 – an explicit-state model checker for multithreaded c & c++ programs. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 863–868, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing c++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM.

[8] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM.

[9] cbloom:blogger.com. `http://cbloomrants.blogspot.com/2011/07/07-18-11-mcs-list-based-lock_18.html`. Accessed: 2019-11-18.

[10] @chill:stackoverflow.com. `https://stackoverflow.com/questions/8115267/writing-a-spinning-thread-barrier-using-c11-atomics`. Accessed: 2019-11-18.

[11] @chxdeng:github.com. `https://github.com/chxdeng/mabain`. Accessed: 2019-11-04.

[12] T. Elmas, J. Burnim, G. Necula, and K. Sen. Concurrit: A domain specific language for reproducing concurrency bugs. *ACM SIGPLAN Notices*, 48, 06 2013.

[13] Facebook. `https://github.com/facebook/folly`. Accessed: 2019-11-20.

[14] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 286.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[15] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 244–254, New York, NY, USA, 2010. ACM.

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

[17] D. Goldblatt, D. Watson, Q. Wang, J. Evans, and Y. Zhang. `http://jemalloc.net/`. Accessed: 2019-11-04.

[18] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[19] B. Jonsson. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Computer Architecture News*, 36:65–71, 01 2008.

[20] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. Repairing sequential consistency in c/c++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. ACM.

[21] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 69–80, New York, NY, USA, 2013. ACM.

[22] C. Lidbury and A. 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.

[23] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[24] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. Elsevier Science Publishers, 10 1988.

[25] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 269–278. ACM, 1991.

[26] K. Nienhuis, K. Memarian, and P. Sewell. An operational semantics for c/c++11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 111–128, New York, NY, USA, 2016. ACM.

[27] B. Norris and B. Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 131–150, New York, NY, USA, 2013. ACM.

[28] B. Norris and B. Demsky. A practical approach for model checking c/c++11 code. *ACM Trans. Program. Lang. Syst.*, 38(3):10:1–10:51, May 2016.

[29] L. Otten. LaTeX template for thesis and dissertation documents at UC Irvine. `https://github.com/lotten/uci-thesis-latex/`, 2012.

[30] P. Ou and B. Demsky. Checking concurrent data structures under the c/c++11 memory model. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 45–59, New York, NY, USA, 2017. ACM.

[31] K. Sen. Effective random testing of concurrent programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 323–332, New York, NY, USA, 2007. ACM.

[32] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[33] V. Štill and J. Barnat. Model checking of c++ programs under the x86-tso memory model. In J. Sun and M. Sun, editors, *Formal Methods and Software Engineering*, pages 124–140, Cham, 2018. Springer International Publishing.

[34] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[35] D. Vyukov. `http://www.1024cores.net/home/relacy-race-detector`. Accessed: 2019-11-20.

[36] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 250–259, New York, NY, USA, 2015. ACM.

[37] X. Zhou. `https://github.com/zxjcarrot/iris`. Accessed: 2019-11-04.