C11Tester: A Race Detector for C/C++ Atomics

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

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

ABSTRACT

Writing correct concurrent code that uses atomics under the C/C++ memory model is extremely difficult. We present C11Tester, a race detector for the C/C++ memory model that can explore executions in a larger fragment of the C/C++ memory model than previous race detector tools. Relative to previous work, C11Tester's larger fragment includes behaviors that are exhibited by ARM processors. C11Tester uses a new constraint-based algorithm to implement modification order that is optimized to allow C11Tester to make decisions in terms of application-visible behaviors. We evaluate C11Tester on several benchmark applications, and compare C11Tester's performance to both tsan11rec, the state of the art tool that controls scheduling for C/C++; and tsan11, the state of the art tool that does not control scheduling.

CCS CONCEPTS

• **Software and its engineering → Concurrent programming languages**.

KEYWORDS

data races, concurrency, C++11, memory models

ACM Reference Format:
Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 17 pages.
https://doi.org/10.1145/3445814.3446711

1 INTRODUCTION

The C/C++11 standards added a weak memory model with support for low-level atomics operations [9, 31] that allows experts to craft efficient concurrent data structures that scale better or provide stronger liveness guarantees than lock-based data structures. 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, or compiled with a specific compiler. Some prior work helps record and replay buggy executions [43]. Debuggers like Symbiosis [41] and Cortex [42] focus on sequential consistency and test programs by modifying thread scheduling of given initial executions. However, both the thread scheduling and relaxed behavior of C/C++ atomics are sources of nondeterminism in a C/C++ programs that use atomics. Thus, it is necessary to develop tools to help test for concurrency bugs. We present the C11Tester tool for testing C/C++ programs that use atomics.

Figure 1 presents an overview of the C11Tester system. C11Tester is implemented as a dynamically linked library together with an LLVM compiler pass, which instruments atomic operations, non-atomic accesses to shared memory locations, and fence operations with function calls into the C11Tester dynamic library. The C++ and pthread library functions are overridden by the C11Tester library—C11Tester implements its own threading library using fibers to precisely control the scheduling of each thread. The C11Tester library implements a race detector and C11Tester reports any races or assertion violations that it discovers.

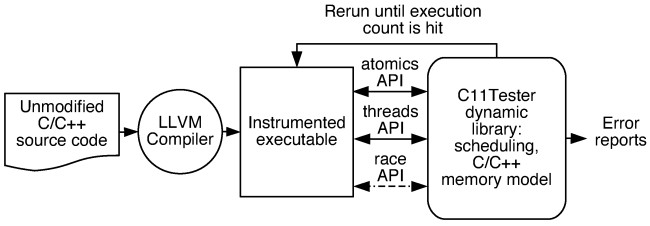

Figure 1: C11Tester system overview

The C/C++ memory model defines the *modification order* relation to totally order all atomic stores to a memory location. This relation captures the notion of cache coherence. The modification order is not directly observable by the program execution — it is only observed indirectly through its effects on program visible behaviors such as the values returned by loads. Under the C/C++ memory model, modification order cannot be extended to be a total order over all stores that is consistent with the happens-before relation.

This paper presents a new technique for scaling a constraint-based treatment of the modification order relation to long executions. *This technique allows C11Tester to support a larger fragment of the C/C++ memory model than previous race detectors.* In particular, this technique can handle the full range of modification orders that are permitted by the C/C++ memory model.

Constraint-based modification order delays decisions about the modification order until the decisions have observable effects on the program's behavior. For example, when an algorithm decides which store a load will read from, C11Tester adds the corresponding constraints to the modification order. This approach allows testing algorithms to focus on program visible behaviors such as

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446711

the value a load reads and does not require them to eagerly decide the modification order.

Fibers provide a more efficient means to control thread schedules than kernel threads. However, C/C++ programs commonly make use of thread local storage (TLS) and fibers do not directly support TLS. This paper presents a new technique, thread context borrowing, that allows fiber-based scheduling to support thread local storage without incurring dependencies on TLS implementation details that can vary across different library versions.

1.1 Comparison to Prior Work on Testing C/C++11

Prior work on data race detectors for C/C++11 such as tsan11 [37] and tsan11rec [38] require $hb \cup rf \cup mo \cup sc$ be acyclic and thus miss potentially 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 found examples of bugs that C11Tester can detect but tsan11 and tsan11rec miss due to the set of $hb \cup rf$ edges orders writes in the modification order.

C11Tester's constraint-based approach to modification order supports a larger fragment of the C/C++ memory model than tsan11 and tsan11rec. C11Tester adds minor constraints to the C/C++ memory model to forbid out-of-thin-air (OOTA) executions for relaxed atomics. Furthermore, these constraints appear to incur minimal overheads on existing ARM processors [46] while x86 and PowerPC processors already implement these constraints.

1.2 Contributions

This paper makes the following contributions:

- **Scalable Concurrency Testing Tool:** It presents a tool for the C/C++ memory model that can test full programs.
- **Supports a Larger Fragment of the C/C++ Memory Model:** It presents a tool that supports a larger fragment of the C/C++ memory model than previous tools.
- **Constraint-Based Modification Order:** The modification order relation is not directly visible to the application, instead it constrains the behaviors of visible relations such as the reads-from relation. We develop a scalable constraint-based approach to modeling the modification order relation that allows algorithms to ignore the modification order relation and focus on program visible behaviors.
- **Support for Limiting Memory Usage:** The size of the C/C++ execution graph and execution trace grows as the program executes and thus limits the length of executions that a testing tool can support. Naively freeing portions of the graph can cause a tool to produce executions that are forbidden by the memory model. We present techniques that can limit the memory usage of C11Tester while ensuring that C11Tester only produces executions that are allowed by the C/C++ memory model.
- **Fiber-based Support for Thread Local Storage:** Fibers are the most efficient way to control the scheduling of the application under test, but supporting thread local storage with fibers is problematic. We develop a novel approach for borrowing the context of a kernel thread to support thread local storage.
- **Evaluation:** We evaluate C11Tester on several applications and compare against both tsan11 and tsan11rec. We show that C11Tester can find bugs that tsan11 and tsan11rec miss. We present a performance comparison with both tsan11 and tsan11rec.

2 C/C++ MEMORY MODEL FRAGMENT

We next describe the fragment of the C/C++ memory model that C11Tester supports. Our memory model has the following changes based on the formalization of Batty *et al.* [8]:

1) Use the C/C++20 release sequence definition: Since the original C/C++11 memory model, the definition of release sequences has been weakened [15]. This change is part of the C/C++20 standard [1]. C11Tester uses the newly weakened definition. The new definition of release sequences does not allow `memory_order_relaxed` stores by the thread that originally performed the `memory_order_release` store that heads the release sequence to appear in the release sequence.

2) Add $hb \cup sc \cup rf$ is acyclic: Supporting load buffering or out-of-thin-air executions is extremely difficult and the existing approaches introduce high overheads in dynamic tools [17, 44, 45]. Thus, we prohibit out-of-thin-air executions with a similar assumption made by much work on the C/C++ memory model — we add the constraint that the union of happens-before, sequential consistency, and reads-from relations, *i.e.*, $hb \cup sc \cup rf$, is acyclic [54].[1] This feature of the C/C++ memory model is known to be generally problematic and similar solutions have been proposed to fix the C/C++ memory model [11, 13, 14, 46].

3) Strengthen consume atomics to acquire: No compilers support the consume access mode. Instead, all compilers strengthen consume atomics to acquire.

We formalize the above changes in Section A.1 of our technical report [40]. Our fragment of the C/C++ memory model is larger than that of tsan11 and tsan11rec [37, 38]. The tsan11 and tsan11rec tools add a very strong restriction to the C/C++ memory model that requires that $hb \cup sc \cup rf \cup mo$ be acyclic.

3 C11TESTER OVERVIEW

We present our algorithm in this section. In our presentation, we adapt some terminology and symbols from stateless model checking [26]. We denote the initial state with s_0. We associate every state transition t taken by thread p with the dynamic operation that affected the transition. We use $enabled(s)$ to denote the set of all threads that are enabled in state s (threads can be disabled when waiting on a mutex, condition variable, or when completed). We say that $next(s, p)$ is the next transition in thread p at state s.

Figure 2 presents pseudocode for C11Tester's exploration algorithm. C11Tester calls Explore multiple times—each time generates one program execution. The thread schedule does not uniquely define the behavior of C/C++ atomics, due to the weak behaviors of C/C++ atomics. Therefore, we split the exploration into two components: (1) selecting the next thread to execute and (2) selecting the behavior of that thread's next operation. C11Tester has a pluggable framework for testing algorithms—C11Tester generates a set of legal choices for the next thread and behavior, and then

[1] The C/C++11 memory model already requires that $hb \cup sc$ is acyclic.

```
1: procedure EXPLORE
2:     s := s_0
3:     while enabled(s) is not empty do
4:         Select p from enabled(s)
5:         t := next(s, p)
6:         behaviors(t) := {Initial behaviors}
7:         Select a behavior b from behaviors(t)
8:         s := Execute(s, t, b)
9:     end while
10: end procedure
```

Figure 2: Pseudocode for C11Tester's Algorithm

the plugin selects the next thread and behavior. The default plugin implements a random strategy.

Scheduling. Thread scheduling decisions are made at each atomic operation, threading operation, or synchronization operation (such as locking a mutex). Every time a thread finishes a visible operation, the next thread to execute is randomly selected from the set of enabled threads. However, when a thread performs several consecutive stores with memory order release or relaxed, the scheduler executes these stores consecutively without interruption from other threads. Executing these stores consecutively does not limit the set of possible executions and provides C11Tester with more stores to select from when deciding which store a load should read from.

Transition Behaviors. The source of multiple behaviors for a given schedule arises from the reads-from relation—in C/C++, loads can read from stores besides just the "last" store to an atomic object.

We use the concept of a *may-read-from* set, which is an overapproximation of the stores that a given atomic load may read from that just considers constraints from the happens-before relation. The *may-read-from* set for a load Y is constructed as:

$$\textit{may-read-from}(Y) = \{X \in \textit{stores}(Y) \mid \neg(Y \overset{hb}{\rightarrow} X) \wedge (\nexists Z \in \textit{stores}(Y) \,.\, X \overset{hb}{\rightarrow} Z \overset{hb}{\rightarrow} Y)\},$$

where $\textit{stores}(Y)$ denotes the set of all stores to the same object from which Y reads. C11Tester selects a store from the *may-read-from* set. C11Tester then checks that establishing this *rf* relation does not violate constraints imposed by the modification order, as described in Section 4. If the given selection is not allowed, C11Tester repeats the selection process. C11Tester delays the modification order check until after a selection is made to optimize for performance.

4 MEMORY MODEL SUPPORT

In this section, we present how C11Tester efficiently supports key aspects of the C/C++ memory model.

CDSChecker [44] initially introduced the technique of using a constraint-based treatment of modification order to remove redundancy from the search space it explores. There are essentially two types of constraints on the modification order: (1) that a store s_A is modification ordered before a store s_B and (2) that a store s_A immediately precedes an RMW r_B in the modification order.

CDSChecker models these constraints using a *modification order graph*. Two types of edges correspond to these two types of constraints. Edges only exist between two nodes if they both represent memory accesses to the same location. There is a cycle in the modification order graph if and only if the graph corresponds to an unsatisfiable set of constraints. Otherwise, a topological sort of the graph (with the additional constraint that an RMW node immediately follows the store that it reads from) yields a modification order that is consistent with the observed program behavior. CDSChecker used depth first search to check for cycles in the graph. CDSChecker would add edges to the modification order graph to determine whether a given reads-from edge was plausible — if the edge made the set of constraints unsatisfiable, CDSChecker would rollback the changes that the edge made to the graph.

This approach works well for model checking where the graphs are small—the fundamental scalability limits of model checking ensure that the executions always contain a very small number of stores. *This approach is infeasible when executions (and thus the modification order graphs) can contain millions of atomic stores, because the graph traversals become extremely expensive.*

4.1 Modification Order Graph

We next describe the modification order graph in more detail. We represent modification order (*mo*) as a set of constraints, built as a constraint graph, namely the modification order graph (*mo-graph*). A node in the *mo-graph* represents a single store or RMW in the execution. There are two types of edges in the graph. An *mo* edge from node A to node B represents the constraint $A \overset{mo}{\rightarrow} B$. A *rmw* edge from node A to node B represents the constraint that A must immediately precede B or formally that: $A \overset{mo}{\rightarrow} B$ and $\forall C. C \neq A \wedge C \neq B \Rightarrow (A \overset{mo}{\rightarrow} C \Rightarrow B \overset{mo}{\rightarrow} C) \wedge (C \overset{mo}{\rightarrow} B \Rightarrow C \overset{mo}{\rightarrow} A)$.

C11Tester must only ensure that there exists some *mo* that satisfies the set of constraints, or equivalently an acyclic *mo-graph*. C11Tester dynamically adds edges to *mo-graph* when new *rf* and *hb* relations are formed. We briefly summarize the properties of *mo* as implications [44] in Figure 3. C11Tester maintains a per-thread list of atomic memory accesses to each memory location. Whenever a new atomic load or store is executed, C11Tester uses this list to evaluate the implications in Figure 3 as well as additional implications for fences.

4.2 Clock Vectors

Due to the high cost of graph traversals for large graphs, graph traversals are not a feasible implementation approach for C11Tester. We next describe how we adapt clock vectors [36] to efficiently compute reachability in the *mo-graph* and scale the constraint-based modification order approach to large executions. We associate a clock vector with each node in the *mo-graph*. *It is important to note that our use of clock vectors in the mo-graph is not to track the happens-before relation. Instead we use clock vectors to efficiently compute reachability between nodes in the mo-graph. Thus, our mo-graph clock vectors model a partial order that contains the current set of ordering constraints on the modification order.*

Each event E [2] in C11Tester has a unique sequence number s_E. Sequence numbers are a global counter of events across all threads, which is incremented by one at each event. We denote the thread that executed E as t_E. Each node in the *mo-graph* represents an

[2]Events in each thread consist of atomic operations, thread creation and join, mutex lock and unlock, and other synchronization operations.

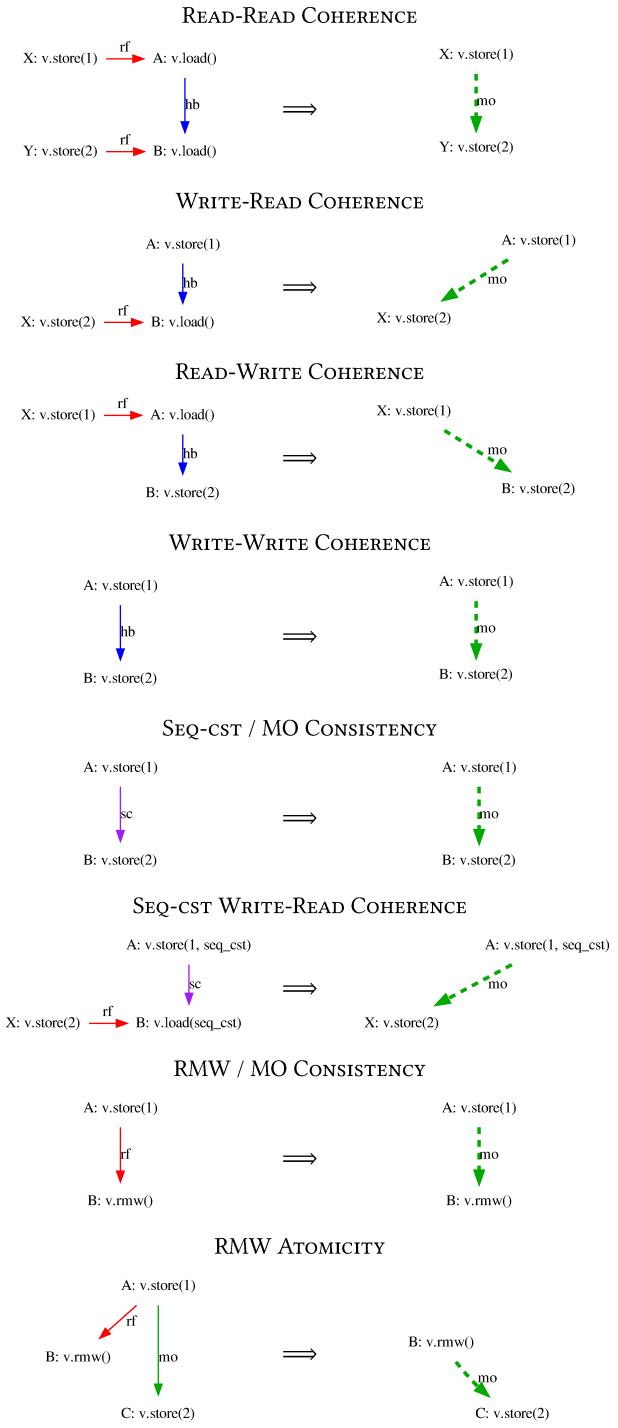

Figure 3: Modification order implications. On the left side of each implication, A, B, C, X, and Y must be distinct.

atomic store. The initial *mo-graph* clock vector $\perp_{CV_A}$ associated with the node representing an atomic store A, the union operator $\cup$, and the comparison operator $\leq$ for *mo-graph* clock vectors are defined as follows:

$$\perp_{CV_A} = \lambda t.\ \text{if } t == t_A \text{ then } s_A \text{ else } 0,$$
$$CV_1 \cup CV_2 \triangleq \lambda t.max(CV_1(t), CV_2(t)),$$
$$CV_1 \leq CV_2 \triangleq \forall t.CV_1(t) \leq CV_2(t).$$

Note that two mo-graph clock vectors can only be compared if their associated nodes represent atomic stores to the same memory location.

The *mo-graph* clock vectors are updated when new *mo* relations are formed. For example, if $A \xrightarrow{mo} B$ is a newly formed *mo* relation, then the node B's *mo-graph* clock vector is merged with that of node A, *i.e.*, $CV_B := CV_A \cup CV_B$. If CV_B is updated by this merge, the change in CV_B must be propagated to all nodes reachable from B using the union operator.

Figure 4 presents pseudocode for updating the modification order graph. The Merge procedure merges the *mo-graph* clock vector of the `src` node into the `dst` node and returns true if the `dst` *mo-graph* clock vector changed. The AddEdge procedure adds a new modification order edge to the graph. It first compares *mo-graph* clock vectors to check if the edge is redundant and if so drops the edge update. Recall that RMW operations are ordered immediately after the stores that they read from. To implement this, AddEdge checks to see if the `from` node has a `rmw` edge, and if so, follows the `rmw` edge. AddEdge finally adds the relevant edge, and then propagates any changes in the *mo-graph* clock vectors. The AddRMWEdge procedure has two parameters, where the `rmw` node reads from the `from` node. It first adds an `rmw` edge and then migrates any outgoing edges from the source of the edge to the `rmw` node. Finally, it calls the AddEdge procedure to add a normal modification order edge and to propagate *mo-graph* clock vector changes.

Figure 5 presents pseudocode for the helper method AddEdges that adds a set of edges to the *mo-graph*. The parameter *set* is a set of atomic stores or RMWs, and S is an atomic store or RMW. The `GetNode` method converts an atomic action to the corresponding node in the *mo-graph*. If such node does not exist yet, then the method will create a new node in the *mo-graph*.

Theorem 1 guarantees the soundness of our use of *mo-graph* clock vectors. We present the theorem and its proof in Section 9. This theorem states that we can solely rely on *mo-graph* clock vectors to compute reachability between nodes in *mo-graph*.

4.3 Eliminating Rollback in *Mo-graph*

Prior work on constraint-based modification order utilized rollback when it was determined that a given reads-from relation was not feasible [44, 45]. C11Tester may also hit such infeasible executions because the *may-read-from* set defined in Section 3 is an overapproximation of the set of stores that a load can read from. To determine precisely whether a load can read from a store, a naive approach is to add edges to the *mo-graph* and then utilize rollback if adding these edges introduces cycles in the *mo-graph*. However, the addition of clock vectors and clock vector propagation makes rollback much more expensive. It is thus critical that C11Tester avoids the need for rollback. We now discuss how C11Tester avoids rollback.

```
1:  procedure MERGE(Node dst, Node src)
2:      if src.cv ≤ dst.cv then
3:          return false
4:      end if
5:      dst.cv := dst.cv ∪ src.cv
6:      return true
7:  end procedure
1:  procedure ADDEDGE(Node from, Node to)
2:      mustAddEdge := (from.rmw == to ∨ from.tid == to.tid)
3:      if from.cv ≤ to.cv ∧¬ mustAddEdge then
4:          return
5:      end if
6:      while from.rmw ≠ null do
7:          next := from.rmw
8:          if next == to then
9:              break
10:         end if
11:         from := next
12:     end while
13:     from.edges := from.edges ∪ to
14:     if MERGE(to, from) then
15:         Q := { to }
16:         while Q is not empty do
17:             node := remove item from Q
18:             for each dst in node.edges do
19:                 if MERGE(dst, node) then
20:                     Q := Q ∪ dst
21:                 end if
22:             end for
23:         end while
24:     end if
25: end procedure
1:  procedure ADDRMWEDGE(Node from, Node rmw)
2:      from.rmw := rmw
3:      for each dst in from.edges do
4:          if dst ≠ rmw then
5:              rmw.edges := rmw.edges ∪ dst
6:          end if
7:      end for
8:      from.edges := ∅
9:      ADDEDGE(from, rmw)
10: end procedure
```

Figure 4: Pseudocode for Updating *mo-graph*

```
1:  procedure ADDEDGES(set, S)
2:      n_S := GetNode(S)
3:      for each e in set do
4:          n_e := GetNode(e)
5:          ADDEDGE(n_e, n_S)
6:      end for
7:  end procedure
```

Figure 5: Helper method for adding a set of edges to the *mo-graph*

The *mo-graph* is updated whenever a new atomic store, atomic load, or atomic RMW is encountered. Processing a new atomic store, atomic load, or atomic RMW can potentially add multiple edges to the *mo-graph*. We next analyze each case to understand how to avoid rollback:

- **Atomic Store:** Since an atomic load can only read from past stores, a newly created store node in *mo-graph* has no outgoing edges. By the properties of *mo*, only incoming edges from other nodes to this new node will be created. Hence, a new store node cannot introduce any cycles.
- **Atomic Load:** Consider a new atomic load Y that reads from a store X_0. Forming a new *rf* relation may only cause edges to be created from other nodes to the node representing the store X_0. We denote this set of "other nodes" as *ReadPriorSet*(X_0) and compute it using the READPRIORSET procedure in Figure 11. Lines 6, 7, and 8 in the READPRIORSET procedure consider statements 5, 4, and 6 in Section 29.3 of the C++11 standard. Line 9 in the procedure considers write-read and read-read coherences. Therefore, the set returned by the READPRIORSET procedure captures the set of stores from where new *mo* relations are to be formed if the *rf* relation is established.
  Before forming the *rf* relation, C11Tester checks whether any node in *ReadPriorSet*(X_0) is reachable from X_0. If so, then having load Y read from store X_0 will introduce a cycle in the *mo-graph*, so we discard X_0 and try another store. While it is possible for a cycle to contain two or more edges in the set of newly created edges, this also implies that there is a cycle with one edge (since all edges have the same destination).
- **Atomic RMWs:** An atomic RMW is similar to both a load and store, but with the constraint that it must be immediately modification ordered after the store it reads from. We implement this by moving modification order edges from the store it reads from to the RMW. Thus, the same checks used by the load suffice to check for cycles for atomic RMWs.

Thus, C11Tester first computes a set of edges that reading from a given store would add to the *mo-graph*. Then for each edge, it checks the *mo-graph* clock vectors to see if the destination of the edge can reach the source of the edge. If none of the edges would create a cycle, it adds all of the edges to the *mo-graph* using the ADDEDGE and ADDRMWEDGE procedures.

5 OPERATIONAL MODEL

We present our operational model with respect to the tsan11 [37] core language described by the grammar in Figure 6. A program is a sequence of statements. `LocNA` and `LocA` denote disjoint sets of non-atomic and atomic memory locations. A statement can be one of these forms: an `if` statement, assigning the result of an expression to a non-atomic location, forking a new thread, joining a thread via its thread handle, and atomic statements. The symbol ϵ denotes an empty statement. Atomic statements denoted by `StmtA` include atomic loads, store, `RMW`s, and fences. An `RMW` takes a functor, `F`, to implement `RMW` operations, such as `atomic_fetch_add`. We omit loops for simplicity and leave the details of an expression unspecified. We omit lock and unlock operations because they can be implemented with atomic statements.

5.1 Happens-Before Clock Vectors

We next discuss the various happens-before clock vectors that C11Tester uses to implement happens-before relations. Figure 7 presents our algorithm for updating clock vectors used to track

```
Prog  ::= Stmt ; ϵ
Stmt  ::= Stmt ; Stmt
        | if (LocNA) {Stmt} else {Stmt}
        | LocNA := Expr
        | LocNA = Fork(Prog)
        | Join(LocNA)
        | StmtA
        | ϵ
StmtA ::= LocNA = Load(LocA, MO)
        | Store(LocNA, LocA, MO)
        | RMW(LocA, MO, F)
        | Fence(MO)
MO    ::= relaxed | release | acquire | rel_acq
        | seq_cst
Expr  ::= <literal> | LocNA | Expr op Expr
```

Figure 6: Syntax for our core language

States:

$Tid \triangleq \mathbb{Z}$ $\quad Seq \triangleq \mathbb{Z}$ $\quad \mathbb{C} : Tid \to CV$

$\mathbb{F}^{rel} : Tid \to CV$ $\quad \mathbb{RF} : Seq \to CV$ $\quad \mathbb{F}^{acq} : Tid \to CV$

[RELEASE STORE]

$$\frac{\mathbb{RF}' = \mathbb{RF}[s := \mathbb{C}_t]}{\left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right) \Rightarrow^{store_{rel}(s,t)} \left(\mathbb{C}, \mathbb{RF}', \mathbb{F}^{rel}, \mathbb{F}^{acq}\right)}$$

[RELAXED STORE]

$$\frac{\mathbb{RF}' = \mathbb{RF}[s := \mathbb{F}^{rel}_t]}{\left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right) \Rightarrow^{store_{rlx}(s,t)} \left(\mathbb{C}, \mathbb{RF}', \mathbb{F}^{rel}, \mathbb{F}^{acq}\right)}$$

[RELEASE RMW]

$$\frac{\mathbb{RF}' = \mathbb{RF}[s := \mathbb{C}_t \cup \mathbb{RF}_{s'}]}{\left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right) \Rightarrow^{rmw_{rel}(s,t),rf(s',t')} \left(\mathbb{C}, \mathbb{RF}', \mathbb{F}^{rel}, \mathbb{F}^{acq}\right)}$$

[RELAXED RMW]

$$\frac{\mathbb{RF}' = \mathbb{RF}[s := \mathbb{F}^{rel}_t \cup \mathbb{RF}_{s'}]}{\left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right) \Rightarrow^{rmw_{rlx}(s,t),rf(s',t')} \left(\mathbb{C}, \mathbb{RF}', \mathbb{F}^{rel}, \mathbb{F}^{acq}\right)}$$

[ACQUIRE LOAD]

$$\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \cup \mathbb{RF}_{s'}]}{\left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right) \Rightarrow^{load_{acq}(s,t),rf(s',t')} \left(\mathbb{C}', \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right)}$$

[RELAXED LOAD]

$$\frac{\mathbb{F}^{acq'} = \mathbb{C}[t := \mathbb{F}^{acq}_t \cup \mathbb{RF}_{s'}]}{\left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right) \Rightarrow^{load_{rlx}(s,t),rf(s',t')} \left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq'}\right)}$$

[RELEASE FENCE]

$$\frac{\mathbb{F}^{rel'} = \mathbb{F}^{rel}[t := \mathbb{C}_t]}{\left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right) \Rightarrow^{fence_{rel}(t)} \left(\mathbb{C}', \mathbb{RF}, \mathbb{F}^{rel'}, \mathbb{F}^{acq}\right)}$$

[ACQUIRE FENCE]

$$\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \cup \mathbb{F}^{acq}_t]}{\left(\mathbb{C}, \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right) \Rightarrow^{fence_{acq}(t)} \left(\mathbb{C}', \mathbb{RF}, \mathbb{F}^{rel}, \mathbb{F}^{acq}\right)}$$

Figure 7: Semantics for tracking happens-before clock vectors for atomic loads, stores, RMWs, and fences. An RMW also triggers a load rule initially.

happens-before relations for atomic loads, stores, RMWs, and fences. The union operator ∪ between clock vectors is defined the same way as in Section 4.2.

For each thread t, the algorithm maintains the thread's own clock vector $\mathbb{C}_t$, and release- and acquire-fence clock vectors $\mathbb{F}^{rel}_t$ and $\mathbb{F}^{acq}_t$. The algorithm also records a reads-from clock vector $\mathbb{RF}_s$ for each atomic store and RMW. Recall that the sequence number is a global counter of events across all threads, and thus uniquely identifies an event. We use $\mathbb{C}$, $\mathbb{F}^{rel}$, $\mathbb{F}^{acq}$ and $\mathbb{RF}$ to denote these clock vectors across all threads, and atomic stores and RMWs. The rules for atomic loads and RMWs also require the stores or RMWs that are read from to be specified, which are denoted as *rf*.

Release Sequences. The 2011 standard used a complicated definition of release sequences that allowed the possibility of relaxed writes blocking release sequences [37]. The 2020 standard simplifies and weakens the definition of release sequences. In a recently approved draft [1], a store-release heads a release sequence and an RMW is part of the release sequence if and only if it reads from a store or RMW that is part of the release sequence. A load-acquire synchronizes with a store-release S if the load reads from a store or RMW in the release sequence headed by S.

We first discuss C11Tester's treatment of release sequences in the absence of fences. C11Tester uses two clock vectors for store/RMW operations: both the current thread clock vector $\mathbb{C}_t$ and a second *reads-from* clock vector $\mathbb{RF}_S$ that tracks the happens-before relation for all release sequences that the RMW/store S is part of. For a normal store release, these two clock vectors are the same. When a relaxed or release RMW A reads from another store B, C11Tester computes the RMW's reads-from clock vector $\mathbb{RF}_A$ as the union of: (1) the store B's reads-from clock vector $\mathbb{RF}_B$ and (2) the RMW A's current thread clock vector $\mathbb{C}_{t_A}$ if A is a release. When a load-acquire A reads from a store-release or RMW, C11Tester computes the load-acquire's new thread clock vector as the union of: (1) the load-acquire's current thread clock vector $\mathbb{C}_{t_A}$ and (2) the store release/RMW's reads-from clock vector.

Fences. The C/C++ memory model also contains fences. Fences can have one of four different memory orders: acquire, release, acq_rel, and seq_cst. Release fences effectively make later relaxed stores into store-releases, but the happens-before relation is established at the fence-release. C11Tester maintains a release fence clock vector $\mathbb{F}^{rel}_t$ for each thread and uses this clock vector when computing the clock vector for release sequences. Acquire fences effectively make previous relaxed loads into load-acquires, but the happens-before relation starts at the fence. When a relaxed load reads from a release sequence, C11Tester updates the per-thread acquire-fence clock vector $\mathbb{F}^{acq}_t$. When C11Tester processes an acquire fence, it uses $\mathbb{F}^{acq}_t$ to update the thread's clock vector $\mathbb{C}_t$. Seq_cst fences constrain the interactions between sequentially consistent atomics and non-sequentially consistent atomics. The behavior of seq_cst fences can be represented as rules for generating modification order constraints [8]. C11Tester maintains a list of all seq_cst fences for each thread so that C11Tester can quickly locate the relevant fence instructions. It then generates the relevant modification order edges to implement the fence semantics.

5.2 Formal Operational Model

Figure 8 formalizes the operational state of a program. The state of system *State* consists of the list of *ThrState*, the mapping *ALocs* from memory locations to atomic information, the mapping *NALocs* from

memory locations to values stored at non-atomic locations, the mapping *FenceInfo*, and the *mo-graph* described in Section 4. *ALocInfo* records the list of atomic loads, stores, and RMWs performed at a given atomic location. *FenceInfo* records the list of fences performed by each thread. *Prog* is a program described by the grammar in Figure 6. The initial state of the system has empty mappings *ALocs* and *NALocs*, and *FenceInfo*, only one thread representing the main function, and an empty *mo-graph*.

$$
\begin{aligned}
Tid &\triangleq \mathbb{Z} \quad Epoch \triangleq \mathbb{Z} \quad Val \triangleq \mathbb{Z} \quad Seq \triangleq \mathbb{Z} \\
CV &\triangleq Tid \rightarrow Epoch \\
ThrState &\triangleq (t : Tid) \times (\mathbb{C} : CV) \times (\mathbb{F}^{\{rel,\, acq\}} : CV) \times (\mathbb{RF} : Seq \rightarrow CV) \\
&\quad \times (P : Prog) \\
StoreElem &\triangleq (t : Tid) \times (s : Seq) \times (a : LocA) \times (mo : MemoryOrder) \\
&\quad \times (v : Val) \\
LoadElem &\triangleq (t : Tid) \times (s : Seq) \times (a : LocA) \times (mo : MemoryOrder) \\
&\quad \times (rf : StoreElem) \\
RMWElem &\triangleq (t : Tid) \times (s : Seq) \times (a : LocA) \times (mo : MemoryOrder) \\
&\quad \times (rf : StoreElem \text{ or } RMWElem) \times (v : Val) \\
FenceElem &\triangleq (t : Tid) \times (s : Seq) \times (mo : MemoryOrder) \\
ALocInfo &\triangleq (StoreElem \text{ or } LoadElem \text{ or } RMWElem) \text{ list} \\
FenceInfo &\triangleq Tid \rightarrow FenceElem \text{ list} \\
ALocs &\triangleq LocA \rightarrow ALocInfo \\
NALocs &\triangleq LocNA \rightarrow Val \\
State &\triangleq ThrState \text{ list} \times ALocs \times NALocs \\
&\quad \times FenceInfo \times (M : mo\text{-}graph)
\end{aligned}
$$

Figure 8: Operational State

5.3 Operational Semantics

Figures 9 to 11 present state transitions and related algorithms for our operational model. A system under evaluation is a triple of the form (Σ, *ss*, *T*), where Σ represents the state of the system *State*, *ss* is the program being executed, and *T* represents *ThrState* of the thread currently running the program. The current thread only updates its own state *T* when the program *ss* executes, which causes the copy of *T* in Σ to become outdated. However, the updated *T* will replace the old copy in Σ when the thread switching function δ is called at the end of each atomic statement. The *mo-graph* is a data structure in *State* and represented as Σ.*M*. The *mo-graph* has methods Merge, AddEdge, AddRMWEdge, and AddEdges described in Figure 4 and Figure 5.

Figure 9 shows semantics for atomic statements. Every time an atomic statement is encountered, a corresponding *LoadElem*, *StoreElem*, *RMWElem*, or *FenceElem* is created with the sequence number auto-assigned. The process of assigning sequence numbers are omitted in Figure 9. Function calls [LOAD], [STORE], [RMW], and [FENCE] invokes the corresponding inference rules for updating clock vectors described in Figure 7 based on the type of atomic statements and the memory orders. Atomic statements with `seq_cst` or `acq_rel` memory orderings invoke both acquire and release clock vector rules if they apply. [LOAD], [STORE], [RMW], and [FENCE] take the current state of the system, the current atomic element, and the state of the current thread as arguments, pass necessary input into the inference rules for updating clock vectors, and finally return the updated state of the current thread.

For atomic loads and RMWs, the store that is read from is randomly selected from the *may-read-from* set computed using the algorithm BuildMayReadFrom presented in Figure 10, and the store must satisfy the constraint that the second return value of ReadPriorSet is true, *i.e.*, having the load reading from the selected store does not create a cycle in the *mo-graph*. The atomic RMW rule first triggers an atomic load rule, and the store/RMW *S* that is read from is recorded in the *rf* field of the *RMWElem*. Then, the *mo-graph* is updated using the procedure AddRMWEdge, and the atomic RMW rules is finally finished by invoking an atomic store rule. Both atomic load and atomic store rules call the helper method AddEdges in Figure 5 to add edges to the *mo-graph*.

Figure 11 presents the procedures ReadPriorSet and WritePriorSet which compute the set of atomic actions (*mo-graph* nodes) from where new *mo* edges will be formed.

We use the following helper functions in Figure 10 and Figure 11:

- *last_sc_fence*(*t*) returns the last `seq_cst` fence in thread *t*;
- *last_sc_store*(*a*, *S*) returns the last `seq_cst` store performed at location *a* and is different from *S*;
- *sc_fences*(*t*) returns the list of `seq_cst` fences performed by thread *t*;
- *sc_stores*(*t*, *a*) returns the list of `seq_cst` stores and RMWs performed by thread *t* at location *a*;
- *stores*(*t*, *a*) returns the list of stores and RMWs performed by thread *t* at location *a*;
- *loads_stores*(*t*, *a*) returns the list of loads, stores, and RMWs performed by thread *t* at location *a*;
- *last*(list) returns the element with the largest sequence number in the list, excluding null elements;
- *get_write*(*A*) returns *A* if *A* is an atomic store or RMW and returns *A.rf* if *A* is an atomic load.

All the above functions return null if the result does not exist.

5.4 Equivalent to Axiomatic Model

We make our axiomatic model precise and prove the equivalence of our operational and axiomatic models in Section A of our technical report [40].

6 IMPLEMENTATION

We next present several aspects of the C11Tester implementation.

6.1 Pruning the Execution Graph

While keeping the complete C/C++ execution graph and execution trace is feasible for short executions and can help with debugging, for longer executions their size eventually becomes too large to store in memory. Naively pruning the execution trace to retain the most recent actions is not safe—an older store S_A to an atomic location X in the trace can be modification ordered after a later store S_B to X in the trace. If a thread has already read from S_A, it cannot read from S_B because it is modification ordered before S_A. Naively pruning S_A from execution graph without also removing S_B might erroneously produce an invalid execution in which a thread reads from S_A and then S_B.

C11Tester supports two approaches to limiting memory usage: (1) a conservative mode that limits the size of the execution graph

[ATOMIC LOAD]

$$\frac{\begin{array}{c}(\Sigma, T) \rightarrow_{load} (\Sigma, T') \\ L.t = T'.t \quad L.a = a \quad L.mo = mo \quad S \in \textsc{BuildMayReadFrom}(L) \\ L.rf = S \\ (pset, ret) = \textsc{ReadPriorSet}(L, S) \quad ret == True \quad T'' = [\text{LOAD}](\Sigma, L, T') \\ \Sigma' = \Sigma[M := \Sigma.M.\textsc{AddEdges}(pset, S)] \\ \Sigma'' = \Sigma'[NALocs := \Sigma'.NALocs[l := S.v]] \\ \Sigma''' = \Sigma''[ALocs := \Sigma''.ALocs(a).\text{pushback}(L)]\end{array}}{(\Sigma, l = \texttt{Load}(a, mo); ss, T) \Rightarrow (\Sigma'', \delta; ss, T'')}$$

[ATOMIC STORE]

$$\frac{\begin{array}{c}(\Sigma, T) \rightarrow_{store} (\Sigma', T) \quad S.t = T.t \\ S.a = a \quad S.mo = mo \quad S.v = \Sigma'.NALocs(l) \quad pset = \textsc{WritePriorSet}(S) \\ T' = [\text{STORE}](\Sigma', S, T) \\ \Sigma'' = \Sigma'[M := \Sigma'.M.\textsc{AddEdges}(pset, S)] \\ \Sigma''' = \Sigma''[ALocs := \Sigma''.ALocs(a).\text{pushback}(S)]\end{array}}{(\Sigma, \texttt{Store}(l, a, mo); ss, T) \Rightarrow (\Sigma''', \delta; ss, T')}$$

[ATOMIC RMW]

$$\frac{\begin{array}{c}(\Sigma, T) \rightarrow_{rmw} (\Sigma', T') \quad R.t = T'.t \quad R.a = a \quad R.mo = mo \\ (\Sigma', l = \texttt{Load}(a, mo), T') \rightarrow (\Sigma'', ss, T'') \\ R.rf = S \quad T''' = [\text{RMW}](\Sigma'', R, T'') \\ \Sigma''' = \Sigma''[M := \Sigma''.M.\textsc{AddRMWEdge}(\texttt{GetNode}(R.rf), \texttt{GetNode}(R))] \\ \Sigma'''' = \Sigma'''[ALocs := \Sigma'''.ALocs(a).\text{pushback}(R)]\end{array}}{\begin{array}{c}(\Sigma, \texttt{RMW}(a, mo, F); ss, T) \Rightarrow \\ (\Sigma'''', l = F(l); R.v = \Sigma''''.NALocs(l); \texttt{Store}(l, a, mo); \delta; ss, T''')\end{array}}$$

[ATOMIC FENCE]

$$\frac{\begin{array}{c}F.t = T.t \quad F.mo = mo \\ T' = [\text{FENCE}](\Sigma, F, T) \quad \Sigma' = \Sigma[FenceInfo := \Sigma.FenceInfo(t).\text{pushback}(F)]\end{array}}{(\Sigma, \texttt{Fence}(mo); ss, T) \Rightarrow (\Sigma', \delta; ss, T')}$$

Figure 9: Semantics for atomic statements

```
1:  procedure BuildMayReadFrom(L)
2:      ret := ∅
3:      if L.mo == seq_cst then
4:          S := last_sc_store(L.a, L)
5:      end if
6:      for all threads t do
7:          stores := stores(t, L.a)
8:          base := {X ∈ stores | ¬(X -hb-> L) ∨ (X -hb-> L ∧ (∄Y ∈ stores . X -sb-> Y -hb-> L))}
9:          if L.mo == seq_cst ∧ S ≠ null then
10:             base := base \ {X ∈ stores | X -sc-> S ∨ X -hb-> S}
11:         end if
12:         ret := ret ∪ base
13:     end for
14:     if L is rmw then
15:         ret := {X ∈ ret | no rmw has read from X}
16:     end if
17:     return ret
18: end procedure
```

Figure 10: Pseudocode for computing *may-read-from* sets

with the constraint that C11Tester must retain the ability to generate all possible executions and (2) an aggressive mode that can potentially reduce the set of executions that C11Tester can produce.

Conservative Mode. The key idea behind the conservative mode is to compute a set of older stores that can no longer be read by any thread and thus can be safely removed from the execution graph. The basic idea is to compute the latest action A_t for each thread t such that for the last action $L_{t'}$ in every other thread t', we have $A_t \xrightarrow{hb} L_{t'}$. If action S is a store that either happens before A_t or is A_t, then any new loads from the same memory location must either read from S or some store that is modification ordered after S.

```
1:  procedure WritePriorSet(S)
2:      priorset := ∅; F_S := last_sc_fence(S.t); is_sc_store := ( S.mo == seq_cst )
3:      if is_sc_store then
4:          add last_sc_store(S.a, S) to priorset
5:      end if
6:      for all threads t do
7:          F_t := last_sc_fence(t)
8:          F_b := last({F ∈ sc_fences(t) | F_S ≠ null ∧ F -sc-> F_S})
9:          S_1 := last({X ∈ stores(t, S.a) | is_sc_store ∧ F_t ≠ null ∧ X -sb-> F_t})
10:         S_2 := last({X ∈ sc_stores(t, S.a) | F_S ≠ null ∧ X -sc-> F_S})
11:         S_3 := last({X ∈ stores(t, S.a) | F_b ≠ null ∧ X -sb-> F_b})
12:         S_4 := last({X ∈ load_stores(t, S.a) | X -hb-> S})
13:         add get_write(last({S_1, S_2, S_3, S_4})) to priorset
14:     end for
15:     return priorset
16: end procedure
1:  procedure ReadPriorSet(L, S)
2:      priorset := ∅; F_L := last_sc_fence(L.t); is_sc_load := ( L.mo == seq_cst )
3:      for all threads t do
4:          F_t := last_sc_fence(t)
5:          F_b := last({F ∈ sc_fences(t) | F_L ≠ null ∧ F -sc-> F_L})
6:          S_1 := last({X ∈ stores(t, L.a) | is_sc_load ∧ F_t ≠ null ∧ X -sb-> F_t})
7:          S_2 := last({X ∈ sc_stores(t, L.a) | F_L ≠ null ∧ X -sc-> F_L})
8:          S_3 := last({X ∈ stores(t, L.a) | F_b ≠ null ∧ X -sb-> F_b})
9:          S_4 := last({X ∈ load_stores(t, L.a) | X -hb-> L})
10:         A := get_write(last({S_1, S_2, S_3, S_4}))
11:         if A ≠ S then
12:             add A to priorset
13:         end if
14:     end for
15:     for each e in priorset do
16:         if e is reachable from S in mo-graph then
17:             return (∅, false)
18:         end if
19:     end for
20:     return (priorset, true)
21: end procedure
```

Figure 11: Pseudocode for computing *priorsets* for atomic stores and loads

Thus any store S_{old} that is modification ordered before the store S can no longer be read from by any thread and can be safely pruned.

C11Tester efficiently computes a clock vector CV_{min} to identify such actions A_t for each thread by using the intersection operator, $\cap$, to combine the clock vectors of all running threads. We define the intersection operator $\cap$ as follows:

$$CV_1 \cap CV_2 \triangleq \lambda t.min(CV_1(t), CV_2(t)).$$

C11Tester then searches for stores that happen before these operations. It then uses the *mo-graph* to identify old stores to prune. Finally, it prunes these stores and any loads that read from them.

Aggressive Mode. If a thread fails to synchronize with other threads, this can prevent C11Tester from freeing much of the execution graph or execution trace as such a thread can potentially read from older stores in the execution trace and thus prevent freeing those stores. In the aggressive mode, the user provides a window of the trace that C11Tester attempts to keep in the graph. Simply deleting all memory operations before that window is not sound as newer (with respect to the trace) memory operations may be modification ordered before older memory operations. Thus removing older memory operations could cause C11Tester to erroneously allow loads to read from stores they should not.

For a store S outside of this window, C11Tester attempts to remove all stores modification ordered before S. Such stores can in some cases be inside of the window that C11Tester attempts to preserve, but they must also be removed. C11Tester then removes any loads that read from the removed stores.

Fences. Release fences that happen before actions whose sequence numbers correspond to components of CV_{min} are not necessary to keep since every running thread has already synchronized with a later point in the respective thread's execution. Thus such release fences can be safely removed.

After an acquire fence is executed, its effect is summarized in the clock vector of subsequent actions in the same thread. Thus acquire fences can be safely removed.

Sequentially consistent fences that happen before CV_{min} are no longer necessary since the happens-before relation will enforce the same orderings. Thus, such sequentially consistent fences can be safely removed.

6.2 Race Detection

C11Tester uses a FastTrack [24]-like approach to race detection with the per-thread happens-before clock vectors from the operational semantics. It uses the standard C/C++ definition of races. Section 7.2 of our technical report [40] discusses this in more detail.

6.3 Scheduling

There are two general techniques for controlling the schedule for executing threads. The first technique is to map application threads to kernel threads and then use synchronization constructs to control which thread takes a step. The second technique is to simulate application threads with user threads or fibers that are all mapped to one kernel thread. While there is a proposal for user-space control of thread scheduling that provides very low latency context switches, unfortunately it still has not been implemented in the mainline Linux kernel [53] after six years.

We implemented a microbenchmark on x86 to measure the context switch costs for several implementations of these two techniques. Our microbenchmark starts two threads or fibers and measures the time to switch between these threads. The experiment results are available in our technical report [40].

For the kernel threads, we implemented four approaches to context switches. The first approach uses standard pthread condition variables and was generally the slowest approach. The second approach uses Linux futexes and is a little faster. The next two approaches use spinning to wait. Simply spinning is very fast if every thread has its own core. As soon as two threads have to share a core, this approach becomes 10,000× slower than the other approaches because it has to wait for a scheduling epoch to occur to switch contexts. We also implemented a version that adds a yield call. This hurts performance if both threads run on their own core, but significantly helps performance if threads share a core. But in general, spinning is problematic as idle threads keep cores busy.

For the fiber-based approaches, we used both `swapcontext` and `setjmp` to implement fibers. `Swapcontext` is significantly slower than `setjmp` because it makes a system call to update the signal mask. An issue with these approaches is that neither call updates the register that points to thread local storage. Updating this register requires a system call, and this slows down both fiber approaches.

For practical implementation strategies, the fiber-based approach is faster than kernel threads. Thus, C11Tester uses fibers implemented via `swapcontext` to simulate application threads.

6.4 Thread Context Borrowing

A major challenge with implementing fibers is supporting thread local storage. The specification for thread local storage on x86-64 [20] is complicated and leaves many important details implementation-defined and these details vary across different versions of the standard library. Generating a correct thread local storage region for each thread is a significant effort as it requires continually updating C11Tester code to support the current set of library implementation strategies. This is complicated by the fact that creating the thread local storage may involve calling initializers and freeing the thread local storage may involve calling destructors.

Instead, C11Tester implements a technique for borrowing the thread context including the thread local storage from a kernel thread. The idea is that for each fiber C11Tester creates a real kernel thread and the fiber borrows the kernel thread's entire context including its thread local storage.

C11Tester implements thread context borrowing by first creating and locking a mutex to protect the thread context and then creating a new kernel thread to serve as a lending thread that lends its context to C11Tester. The lending thread then creates a fiber context and switches to the fiber context. The fiber context then transfers the lending thread's context along with its thread local storage to the C11Tester. Finally, the fiber context grabs the context mutex to wait for the C11Tester to return its context. Once the application thread is finished, C11Tester returns the thread context to the lending thread by releasing the context mutex. The lending thread then switches back to its original context, frees its fiber context, and then exits. Migrating thread local storage on x86 requires a system call to change the *fs* register. C11Tester implements thread context borrowing for x86, but the basic idea should work for any architecture.

6.5 Repeated Execution

C11Tester supports repeatedly executing the same benchmark to find hard-to-trigger bugs. It can be desirable for testing algorithms to maintain state between executions to attempt to explore different program behaviors across different executions. C11Tester maintains its internal state across executions of the application under test and resets the application's state between executions.

C11Tester uses fork-based snapshots to restore the application to its initial state. C11Tester uses the `mmap` library call to map a shared memory region to store its internal state. The data in this shared memory region persists across different executions. This state allows C11Tester to report data races only once as opposed to reporting the same race on each execution. It also allows for the creation of smart plugins that explore different behaviors across different executions.

7 EVALUATION

We compare C11Tester with both tsan11rec, a race detector that supports controlled execution [38] and tsan11 [37], a race detector that relies on the operating system scheduler to control the scheduling of threads. We ran our experiments on an Ubuntu Linux 18.04 LTS machine with a 6 core Intel Core i7-8700K CPU and 64GB RAM. We first evaluated the above tools on buggy implementations of seqlock and reader-writer lock to check whether all three tools can detect the injected bugs. Then we evaluated the three tools on both a set of five applications that make extensive use of C/C++ atomics and the data structure benchmarks used to evaluate CDSChecker previously [44].

The way these three tools support multi-threading differs significantly. C11Tester sequentializes thread executions and only allows one thread to execute at a single time, tsan11 allows multiple threads to execute in parallel, while tsan11rec falls in between—it sequentializes visible operations (such as atomics, thread operations, and synchronization operations) and runs invisible operations in parallel. The closest tool to compare C11Tester with is tsan11rec because both C11Tester and tsan11rec support controlled scheduling, while results for tsan11 are also presented for completeness. Although both tsan11 and tsan11rec execute all or some operations in parallel, we present a best effort comparison in the following.

7.1 Benchmarks with Injected Bugs

We have injected bugs into two commonly used data structures and verified that both tsan11 and tsan11rec miss these bugs due to the restrictions of their memory models and that the buggy executions contained cycles in $hb \cup rf \cup mo \cup sc$.

Seqlock. We took the seqlock implementation from Figure 5 of Hans Boehm's MSPC 12 paper [12], made the writer correctly use release atomics for the data field stores, and injected a bug by weakening atomics that initially increment the counter to relaxed memory ordering.

Reader-Writer Lock. We also implemented a broken reader-writer lock where the write-lock operation incorrectly uses relaxed atomics. The test case uses the read-lock to protect reads from atomic variables and the write-lock to protect writes to atomic variables.

C11Tester was able to detect the injected bugs in the broken seqlock and reader-writer lock with bug detection rates of 28.8% and 55.3%, respectively, in 1,000 runs. However, tsan11 and tsan11rec failed to detect the bugs in 10,000 runs.

7.2 Real-World Applications

Ideally, we would evaluate the tools against real world applications that make extensive use of C/C++ atomics. However, to our knowledge, no such standard benchmark suite exists so far. So we gathered our benchmarks through searching for benchmarks evaluated in previous work as well as concurrent programs on GitHub.

The five large applications that we have gathered include: GDAX [7], an in-memory copy of the order book of the GDAX cryptocurrency exchange; Iris [57], a low-latency C++ logging library; Mabain [19], a key-value store library; Silo [51, 52], a multi-core in-memory storage engine; and the Firefox JavaScript engine release 50.0.1.[3] To make our results as reproducible as possible, we tested the JavaScript engine using the offline version of JSBench v2013.1. [48] [4]

As the three tools supported multi-threading in different ways, to make a fair comparison, we ran each experiment on application benchmarks in both the all-core configuration, where all hardware cores could be utilized, and the single-core configuration, where the tools were restricted to running on a single CPU using the Linux command `taskset`. As it is always trivial to parallelize testing by running several copies of a tool in parallel, the rationale behind the single-core experiment is to compare the total CPU time used to execute a benchmark or the equivalent throughput under different tools. However, to understand the performance benefits of parallelism for the other tools, we also ran experiments in the all-core configuration. The performance of C11Tester does not vary much in two configurations, because C11Tester only schedules one thread to run at a time.

Table 1 summarizes the average and relative standard deviation (in parentheses) of execution time or throughput for each of the five benchmarks in the single-core and all-core configurations. Table 1 reports wall-clock time for Iris and Mabain. The throughput of Silo is the aggregate throughput (`agg_throughput`) reported by Silo, and the unit is ops/sec, *i.e.*, the number of database operations performed per second. The throughput of GDAX is the number of iterations which the entire data set is iterated over in 120s. The time and relative standard deviation reported for JSBench are the statistics reported by the python script in JSBench over 10 runs. For the other four benchmarks, the average and relative standard deviation of the time and throughput are calculated over 10 runs.

C11Tester is slower than tsan11 in all benchmarks except Silo in the single-core configuration. C11Tester is faster than tsan11rec in all benchmarks except JSBench in the all-core configuration.

Figure 12 summarizes speedups compared to tsan11 on the single-core configuration for each tool under both configurations, which are derived from data in Table 1. Tsan11 on the single-core configuration is set as the baseline and is omitted from Figure 12.

Based on the results in Figure 12, we further calculated the geometric mean of the speedup over the five benchmarks for each tool under both configurations. According to the geometric means, C11Tester is 14.9× and 11.1× faster than tsan11rec in the single-core configuration and all-core configuration, respectively. C11Tester is 1.6× and 3.1× slower than tsan11 in the single-core configuration and all-core configuration, respectively.

Silo. Silo [51, 52] is an in-memory database that is designed for performance and scalability for modern multicore machines. The test driver we used is `dbtest.cc`. We ran the driver for 30 seconds each run with option `"-t 5"`, *i.e.*, 5 threads in parallel.

In the first part of the experiment, Silo was compiled with invariant checking turned on. C11Tester found executions in which invariants were violated. We found that it was because Silo used volatiles with gcc intrinsic atomics to implement a spinlock and assumed stronger behaviors from volatiles than C11Tester's default handling of volatiles as relaxed atomics. The bug disappeared when we handled volatile loads and stores as load-acquire and

[3] https://ftp.mozilla.org/pub/firefox/releases/50.0.1/source/
[4] https://plg.uwaterloo.ca/~dynjs/jsbench/

Table 1: Performance results for application benchmarks in the single-core and all-core configurations. The results are averaged over 10 runs. Relative standard deviation is reported in parentheses. Larger throughputs are better for throughput-based measurements, smaller times are better for time-based measurements.

	Single-core Configuration			All-core Configuration			
Test	**C11Tester**	**tsan11rec**	**tsan11**	**C11Tester**	**tsan11rec**	**tsan11**	**Measurement**
Silo	15267 (0.45%)	436 (2.52%)	5496 (4.54%)	15297 (1.17%)	438.3 (0.59%)	46688 (1.68%)	Throughput (ops/sec)
GDAX	2953 (1.80%)	69.3 (0.97%)	15700 (0.12%)	2946 (1.64%)	49.4 (1.04%)	53362 (11.4%)	Throughput (# of iterations)
Mabain	5.77 (0.25%)	593.4 (0.98%)	3.513 (1.15%)	5.69 (0.04%)	441.6 (0.69%)	7.00 (0.22%)	Time (in s)
Iris	8.95 (1.46%)	31.31 (0.89%)	4.873 (1.64%)	8.86 (0.22%)	17.20 (1.05%)	2.725 (4.07%)	Time (in s)
JSBench	1835 (0.26%)	2522 (1.41%)	867.8 (0.21%)	1836 (0.35%)	970.7 (0.68%)	781.9 (0.61%)	Time (in ms)

Table 2: Performance results for data structure benchmarks. The time column gives the time taken to execute the test case once, averaged over 500 runs. The rate column gives the percentage of executions in which the data race is detected among 500 runs.

Test	**C11Tester**		**tsan11rec**		**tsan11**	
	Time	rate	Time	rate	Time	rate
barrier	4ms	76.6%	19ms	36.4%	12ms	0.0 %
chase-lev-deque	2ms	94.6%	7ms	0.0 %	3ms	0.0 %
dekker-fences	2ms	21.6%	10ms	41.4%	5ms	53.2 %
linuxrwlocks	2ms	86.2%	10ms	53.4%	5ms	1.6 %
mcs-lock	3ms	89.4%	11ms	71.4%	14ms	0.8 %
mpmc-queue	4ms	59.4%	10ms	58.2%	5ms	0.4 %
ms-queue	4ms	100.0%	136ms	100.0%	9ms	100.0%
Average		75.4%		51.5%		22.3%

store-release atomics. Volatile variables were commonly used to implement atomic memory accesses before C/C++11. However, this usage of volatile is technically incorrect, because the C++ standard provides no guarantee when volatiles are mixed with atomics, and weaker behaviors for volatiles can be exhibited by ARM processors.

We ran both tsan11rec and tsan11 on Silo for 100 runs with 30s each run. Tsan11rec was not able to reproduce the weak behaviors that C11Tester discovered, while tsan11 could reproduce the weak behaviors 35% of the time. Tsan11rec and tsan11 both found racy accesses on volatile variables that were used to implement a spin lock. C11Tester did not report an error message for the volatile races because C11Tester intentionally elides race warnings for races involving volatiles and atomic accesses or races involving volatiles and volatiles because volatiles are in practice still commonly used to implement atomics.

When measuring performance for Silo, we turned off invariant checking. We measured performances in terms of aggregate throughput reported by Silo. C11Tester is faster than tsan11 in the single-core configuration, because reporting data races caused significant overhead for tsan11 in the case of Silo.

Mabain. Mabain is a lightweight key-value store library [19]. Mabain contains a few test drivers that insert key-value pairs concurrently into the Mabain system—we used `mb_multi_thread_insert_test.cpp`. All tools discovered an application bug that caused assertions in the test driver to fail, although tsan11 required us to set a different number of threads than our standard test harness to detect it. For performance measurements, we turned off assertions in the test driver. All tools found data races in Mabain.

The application bug is as follows. The test driver has one asynchronous writer and a few workers. The workers and the writer communicate via a shared queue protected by a lock. The writer consumes jobs (insertion into the database) in the queue and insert values into the Mabain database, while the workers submit jobs into the queue. When workers finish submitting all jobs into the queue, the writer is stopped. However, there is no check to make sure that all jobs in the queue have been cleared before the writer is stopped. Thus, after the writer is stopped, some values may not be found in the Mabain database, causing assertion failures.

The time reported in Table 1 was measured for inserting 100,000 key-value pairs into the Mabain system.

GDAX. GDAX [7] implements an in-memory copy of the order book for the GDAX cryptocurrency exchange using a lock-free skip list with garbage collection from the libcds library [32]. The original GDAX fetches data from a server, but we have recorded input data from a previous run and modified GDAX to read local data. All tools reported data races in GDAX.

In our experiment, GDAX was run for 120s each time, during which 5 threads kept iterating over the data set. We counted the number of iterations the data set was iterated over by each tool in each run and computed statistics based on 10 runs.

Iris. Iris [57] is a low latency asynchronous C++ logging library that buffers data using lock-free circular queues. The test driver we used to measure performance was `test_lfringbuffer.cpp`, in which there is one producer and one consumer. To make the test driver finish in a timely manner, we reduced the number of ITERATIONS to 1 million in the test driver. All tools reported data races in Iris.

Firefox JavaScript Engine. We compiled the Firefox JavaScript engine release 50.0.1 following the instructions for building the JavaScript shell with Thread Sanitizer given by the developers of Firefox. [5] We tested the JavaScript engine with the JSBench suite, which contains 25 JavaScript benchmarks, sampled from real-world applications. The Python script of JSBench first calculated the arithmetic mean of all 25 benchmarks over 10 runs, and then took the geometric means of the 25 arithmetic mean, as reported in Table 1.

[5] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Thread_Sanitizer

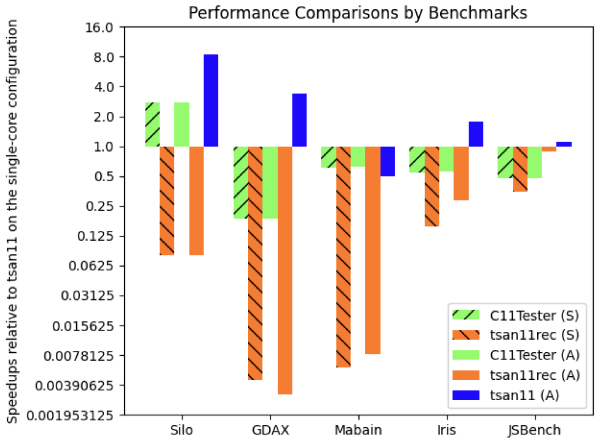

Figure 12: Speedups compared to tsan11 on the single-core configuration for all three tools under both configurations, derived from Table 1. The performance results of tsan11 on the single-core configuration is set as the baseline and is omitted in the Figure. The larger values the faster the tools are. The "(S)" label stands for the single-core configuration, and "(A)" stands for the all-core configuration.

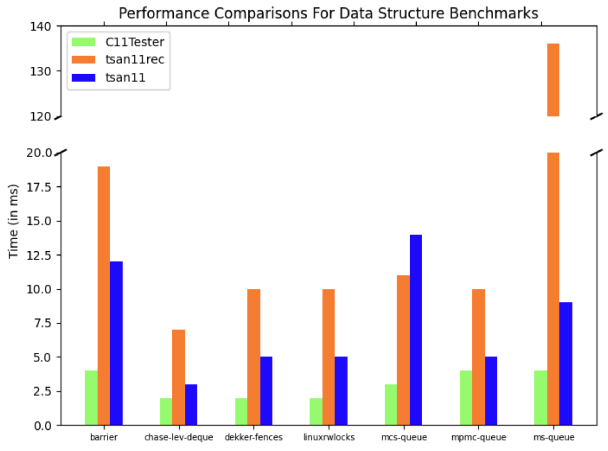

Figure 13: Performance comparisons for data structure benchmarks, based on data in table 2.

7.3 Data Structure Benchmarks

To assess the ability of C11Tester to discover data races, we also used the data structure benchmarks that were originally used to evaluate CDSChecker and subsequently modified to evaluate tsan11 and tsan11rec. We used the version of the benchmarks available at https://github.com/mc-imperial/tsan11. Note that sleep statements were added to 6 of these benchmarks to induce some variability in the schedules explored by the tsan11 [37]. We replicated the same timing strategy used in [37] and reported times that were the sum of the user time and system time measured by the `time` command. Due to differences in the implementation of the sleep statement, sleep time is partially included in C11Tester's user time and thus we removed the sleep statements for C11Tester to make the comparison fair. We executed the benchmarks in the all-core configuration to ensure that we did not put tsan11 at a disadvantage since it does not control the thread schedule.

Table 2 summarizes the experiment results for the data structure benchmarks. The times reported in Table 2 were averaged over 500 runs, and the rate columns report data races detection rates based on 500 runs. Out of 7 benchmarks, C11Tester detects data races with rates higher than tsan11rec in 4 benchmarks and tsan11 in 5 benchmarks. Tsan11 and tsan11rec did not detect races in chase-lev-deque, but C11Tester did. All three tools always detected races in ms-queue.

8 RELATED WORK

Related work falls into three categories: model checkers, fuzzers, and race detectors.

Model Checkers. In the context of weak hardware memory models, researchers have developed stateful model checkers [30, 35, 47]. Stateful model checkers however are limited by the state explosion problem and have the general problem of comparing abstractly equivalent but concretely different program states.

Stateless model checkers have been developed for the C/C++ memory model. CDSChecker can model check real-world C/C++ concurrent data structures [44, 45]. More recent work has led to the development of other model checking tools that can efficiently check fragments of the C/C++ memory model [5, 33, 34]. Recent work on model checking for sequential consistency has developed partial order reduction techniques that only explore all reads-from relations and do not need to explore all sequentially consistent orderings [4]. Other tools such as Herd [6], Nitpick [10], and Cpp-Mem [8] are intended to help understand the behaviors of memory models and do not scale to real-world data structures.

Dynamic Partial Order Reduction [26] and Optimal Dynamic Partial Order Reduction [2] seek to make stateless model checking more efficient by skipping equivalent executions. Maximal causal reduction [27] further refines the technique with the insight that it is only necessary to explore executions in which threads read different values. Recent work has extended these algorithms to handle the TSO and PSO memory models [3, 29, 56]. SATCheck further develops partial order reduction with the insight that it is only necessary to explore executions that exhibit new behaviors [18]. CheckFence checks concurrent code by translating it into SAT [16]. Despite these advances, model checking faces fundamental limitations that prevent it from scaling to full applications.

Fuzzers. The Relacy race detector [55] explores thread interleavings and memory operation reorderings for C++ code. The Relacy race detector has several limitations that cause it to miss executions allowed by the C/C++ memory model. Relacy imposes an execution order on the program under test in which it executes the program. Relacy then derives the modification order from the execution order; it cannot simulate (legal) executions in which the modification order is inconsistent with the execution order.

Adversarial memory increases the likelihood of observing weak memory system behaviors for the purpose of testing [25]. In the context of Java, prescient memory can simulate some of the weak

behaviors allowed by the Java memory model [17]. Prescient memory however requires that the entire application be amenable to deterministic record and replay and uses a single profiling run to generate future values limiting the executions it can discover.

Concutest-JUnit extends JUnit with checks for concurrent unit tests and support for perturbing schedules using randomized waits [49]. Concurrit is a DSL designed to help reproduce concurrency bugs [21]. Developers write code in a DSL to help guide Concurrit to a bug reproducing schedule. CalFuzzer more uniformly samples non-equivalent thread interleavings by using techniques inspired by partial order reduction [50]. These approaches are largely orthogonal to C11Tester.

Race Detectors. Several tools have been designed to detect data races in code that uses standard lock-based concurrency control [22–24, 28, 39]. These tools typically verify that all accesses to shared data are protected by a locking discipline. They miss higher-level semantic races that occur when the locks allow unexpected orderings that produce incorrect results.

9 CORRECTNESS OF *MO-GRAPH*

To prove the correctness of *mo-graphs*, we first prove three Lemmas and then prove Theorem 1. Lemma 1 and Lemma 2 characterize some important properties of *mo-graph* clock vectors. Lemma 3 proves one direction in Theorem 1. *Mo-graph* clock vectors are simply referred to as clock vectors in the following context.

Lemma 1. *Let $C_0 \xrightarrow{mo} C_1 \xrightarrow{mo} \ldots \xrightarrow{mo} C_n$ be a path in a modification order graph G, such that $CV_{C_0} \leq \ldots \leq CV_{C_n}$. Then if any new edge E is added to G using procedures in Figure 4, it holds that*

$$CV'_{C_0} \leq \ldots \leq CV'_{C_n} \tag{9.1}$$

for the updated clock vectors. We define $CV'_{C_i} := CV_{C_i}$ if the values of CV_{C_i} are not actually updated.

Proof. To simplify notation, we define $CV_i := CV_{C_i}$ for all $i \in \{0\ldots, n\}$. Let's first consider the case where no `rmw` edge is added, *i.e.*, the AddRMWEdge procedure is not called.

By the definition of the union operator, each slot in clock vectors is monotonically increasing when the Merge procedure is called. By the structure of procedure AddEdge's algorithm, a node X is added to Q if and only if this node's clock vector is updated by the Merge procedure.

Let's assume that adding the new edge E updates any of $CV_0, \ldots, CV_n$. Otherwise, it is trivial. Let i be the smallest integer in $\{0, \ldots, n\}$ such that CV_i is updated. Then $CV'_k = CV_k$ for all $k \in I := \{0, \ldots, i-1\}$, and we have

$$CV'_0 \leq \ldots \leq CV'_i. \tag{9.2}$$

If $i = 0$, then we take $I = \emptyset$. There are two cases.

Case 1: Suppose $CV'_i \leq CV_j$ for some $j \in \{i+1, \ldots, n\}$, let j_0 be the smallest such integer. Then $CV'_k = CV_k$ for all $k \in \{j_0, \ldots, n\}$, as nodes $\{C_{j_0}, \ldots, C_n\}$ will not be added to Q in the AddEdge procedure, and it holds trivially that

$$CV'_{j_0} \leq \ldots \leq CV'_n. \tag{9.3}$$

By line 14 to line 24 in the AddEdge procedure, we have

$$CV'_k = CV_k \cup CV'_{k-1}, \tag{9.4}$$

for all $k \in S := \{i+1, \ldots, j_0 - 1\}$. If j_0 happens to be $i+1$, then take $S = \emptyset$. And we have for all $k \in S$, $CV'_{k-1} \leq CV'_k$. Then combining with inequality (9.2), we have

$$CV'_0 \leq \ldots \leq CV_i \leq \ldots \leq CV'_{j_0-1}.$$

Together with inequality (9.3), we only need to show that $CV'_{j_0-1} \leq CV'_{j_0}$ to complete the proof.

If $j_0 = i+1$, then we are done, because by assumption $CV'_i \leq CV_{j_0} = CV'_{j_0}$. If $j_0 > i+1$, then $CV'_i \leq CV_{j_0}$ and $CV_{i+1} \leq CV_{j_0}$ imply that $CV'_{i+1} = CV_{i+1} \cup CV'_i \leq CV_{j_0} = CV'_{j_0}$. Based on equation (9.4), we can deduce in a similar way that $CV'_{i+2} \leq \ldots \leq CV'_{j_0-1} \leq CV'_{j_0}$.

Case 2: Suppose $CV_i \not\leq CV_j$ for all $j \in \{i+1, \ldots, n\}$. Then by line 14 to line 24 in the AddEdge procedure, all nodes $\{C_i, \ldots, C_n\}$ are added to Q in the AddEdge procedure, and $CV'_k = CV_k \cup CV'_{k-1}$ for all $k \in S := \{i+1, \ldots, n\}$. This recursive formula guarantees that for all $k \in S$, $CV'_{k-1} \leq CV'_k$. Therefore, combining with inequality (9.2), we have $CV'_0 \leq \ldots \leq CV'_n$.

Now suppose the newly added edge E is a `rmw` edge. If $E : X \xrightarrow{rmw} C_i$ where $i \in \{0, \ldots, n\}$ and X is some node not in path P, then the path P remains unchanged and AddEdge(X,C_i) is called. Then the above proof shows that inequality (9.1) holds. If $E : C_i \xrightarrow{rmw} X$, then $C_i \xrightarrow{mo} C_{i+1}$ is migrated to $X \xrightarrow{mo} C_{i+1}$ by line 3 to line 7 in the AddRMWEdge procedure, and $C_i \xrightarrow{mo} X$ is added.

If X is not in path P, then path P becomes

$$C_0 \xrightarrow{mo} \ldots \xrightarrow{mo} C_i \xrightarrow{mo} X \xrightarrow{mo} C_{i+1} \xrightarrow{mo} \ldots \xrightarrow{mo} C_n.$$

Since AddEdge(C_i,X) is called, the same proof in the case without `rmw` edges applies. If X is in path P, then X can only be C_{i+1} and the path P remains unchanged. Otherwise, a cycle is created and this execution is invalid. In any case, the same proof applies. □

Let $\vec{x} = (x_1, x_2, \ldots, x_n)$. We define the projection function U_i that extracts the i^{th} position of $\vec{x}$ as $U_i(\vec{x}) = x_i$, where we assume $i \leq n$.

Lemma 2. *Let A be a store with sequence number s_A performed by thread i in an acyclic modification order graph G. Then $U_i(CV_A) = U_i(\perp_{CV_A}) = s_A$ throughout each execution that terminates.*

Proof. We will prove by contradiction. Let $S = \{A_1, A_2, \ldots\}$ be the sequence of stores performed by thread i with sequence numbers $\{s_1, s_2, \ldots\}$, respectively. Suppose that there is a point of time in a terminating execution such that the first store A_n in the sequence with $U_i(CV_{A_n}) > s_n$ appears. Sequence numbers are strictly increasing and by the Merge procedure, $U_i(CV_{A_n}) \in \{s_{n+1}, s_{n+2}, \ldots, \}$. Let $U_i(CV_{A_n}) = s_N$ for some $N > n$.

For $U_i(CV_{A_n})$ to increase to s_N from s_n, CV_{A_n} must be merged with the clock vector of some node X (*i.e.*, some store X) in G such that $U_i(CV_X) = s_N$. Such X is modification ordered before A_n.

If X is performed by thread i, then X has to be the store A_N, because $U_i(CV_{A_j})$ is unique for all stores A_j in the sequence S other than A_n. Then $\perp_{CV_X} \geq \perp_{CV_{A_n}}$. By the definition of initial values of clock vectors and sequence numbers, X happens after and is modification ordered after A_n. However, X is also modification ordered before A_n, and we have a cycle in G. This is a contradiction.

If X is not performed by thread i, then $U_i(\perp_{CV_X}) = 0$. For $U_i(CV_X)$ to be s_N, X must be modification ordered after by some store Y in G such that $U_i(CV_Y) = s_N$. If Y is done by thread i, then

the same argument in the last paragraph leads to a contradiction; otherwise, by repeating the same argument as in this paragraph finitely many times (there are only a finite number of stores in such a terminating execution), we would eventually deduce that X is modification ordered after some store by thread i. Hence, we would have a cycle in G, a contradiction.

□

Lemma 3. *Let A and B be two nodes that write to the same location in an acyclic modification order graph G. If B is reachable from A in G, then $CV_A \leq CV_B$.*

Proof. Suppose that B is reachable from A in G. Let $A \xrightarrow{mo} C_1 \xrightarrow{mo} \ldots \xrightarrow{mo} C_{n-1} \xrightarrow{mo} B$ be the shortest path P from A to B in graph G. To simplify notation, $X \xrightarrow{mo} Y$ is abbreviated as $X \rightarrow Y$ in the following. As the AddRMWEdge procedure calls the AddEdge procedure to create an *mo* edge, we can assume that all the *mo* edges in P are created by directly calling AddEdge.

Base Case 1: Suppose the path P has length 1, *i.e.*, A immediately precedes B. Then when the edge $A \rightarrow B$ was formed by calling AddEdge(A,B), CV_B was merged with CV_A in line 14 of the AddEdge procedure. In other words, $CV_B = CV_B \cup CV_A \geq CV_A$.

Base Case 2: Suppose the path P has length 2, *i.e.*, $A \rightarrow C_1 \rightarrow B$. There are two cases:

(a) If $A \rightarrow C_1$ was formed first, then $CV_A \leq CV_{C_1}$. When $C_1 \rightarrow B$ was formed, CV_B was merged with CV_{C_1} and $CV_{C_1} \leq CV_B$. According to Lemma 1, adding the edge $C_1 \rightarrow B$ or any edge not in path P (if any such edges were formed before $C_1 \rightarrow B$ was formed) to G would not break the inequality $CV_A \leq CV_{C_1}$. It follows that $CV_A \leq CV_{C_1} \leq CV_B$.

(b) If $C_1 \rightarrow B$ was formed first, then $CV_{C_1} \leq CV_B$. Based on Lemma 1, this inequality remains true when $A \rightarrow C_1$ was formed. Therefore $CV_A \leq CV_{C_1} \leq CV_B$.

Inductive Step: Suppose that B being reachable from A implies that $CV_A \leq CV_B$ for all paths with length k or less, for some $k > 2$. We want to prove that the same holds for paths with length $k+1$. Let P be a path from A to B with length $k+1$,

$$P : A = C_0 \rightarrow C_1 \rightarrow \ldots \rightarrow C_k \rightarrow C_{k+1} = B.$$

We denote A as C_0 and B as C_{k+1} in the following.

Let $E : C_i \rightarrow C_{i+1}$ be the last edge formed in path P, where $i \in \{0, \ldots, k\}$. Then before edge E was formed, the inductive hypothesis implies that $CV_{C_0} \leq \ldots \leq CV_{C_i}$ and $CV_{C_{i+1}} \leq \ldots \leq CV_{C_{k+1}}$, because both $C_0 \rightarrow \ldots \rightarrow C_i$ and $C_{i+1} \rightarrow \ldots \rightarrow C_{k+1}$ have length k or less. Lemma 1 guarantees that

$$CV_{C_0} \leq \ldots \leq CV_{C_i},$$
$$CV_{C_{i+1}} \leq \ldots \leq CV_{C_{k+1}}$$

remain true if any edge not in path P was added to G as well as the moment when E was formed. Therefore when the edge E was formed, we have $CV_{C_i} \leq CV_{C_{i+1}}$, and

$$CV_A = CV_{C_0} \leq \ldots \leq CV_{C_{k+1}} = CV_B.$$

□

Theorem 1. *Let A and B be two nodes that write to the same location in an acyclic modification order graph G for a terminating execution. Then $CV_A \leq CV_B$ iff B is reachable from A in G.*

Proof. Lemma 3 proves the backward direction, so we only need to prove the forward direction. Suppose that $CV_A \leq CV_B$. Let's first consider the situation where the graph G contain no `rmw` edges.

Case 1: A and B are two stores performed by the same thread with thread id i. Then it is either A happens before B or B happens before A. If A happens before B, then A precedes B in the modification order because A and B are performed by the same thread. Hence B is reachable from A in G. We want to show that the other case is impossible.

If B happens before A and hence precedes A in the modification order, then A is reachable from B. By Lemma 3, A being reachable from B implies that $CV_B \leq CV_A$. Since $CV_A \leq CV_B$ by assumption, we deduce that $CV_A = CV_B$. This is impossible according to Lemma 2, because each store has a unique sequence number and $U_i(CV_A) = s_A \neq s_B = U_i(CV_B)$, implying that $CV_A \neq CV_B$.

Case 2: A and B are two stores done by different threads. Suppose that A is performed by thread i. Let $CV_A = (\ldots, s_A, \ldots)$ and $CV_B = (\ldots, t_b, \ldots)$ where both s_A and t_b are in the i^{th} position. By assumption, we have $0 < s_A \leq t_b$.

Since B is not performed by thread i, we have $U_i(\perp_{CV_B}) = 0$. We can apply the same argument similar to the second, third and fourth paragraphs in the proof of Lemma 2 and deduce that B is modification ordered after A or some store sequenced after A. Since modification order is consistent with *sequenced-before* relation, if follows that B is reachable from A in graph G.

Now, consider the case where `rmw` edges are present. Adding a `rmw` edge from a node S to a node R first transfers to R all outgoing *mo* edges coming from S and then adds a normal *mo* edge from S to R. So, any updates in CV_S are propagated to all nodes that are reachable from S. Therefore, the above argument still applies. □

10 CONCLUSION

We have presented C11Tester, which implements a novel approach for efficiently testing C/C++11 programs. C11Tester supports a larger fragment of the C/C++ memory model than prior work while still delivering competitive performance to prior systems. C11Tester uses a constraint-based approach to the modification order that allows testing tools to make decisions about the modification order implicitly when they select the store that a load reads from. C11Tester includes a data race detector that can identify races. C11Tester supports controlled scheduling for C/C++11 at lower overhead than prior systems. Our evaluation shows that C11Tester can find bugs in all of our benchmark applications including bugs that were missed by other tools.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thorough and insightful comments. We are especially grateful to our shepherd Caroline Trippel for her feedback. We also thank Derek Yeh for his work on performance improvement for the C11Tester tool. This work is supported by the National Science Foundation grants CNS-1703598, OAC-1740210, and CCF-2006948.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains a `c11tester-vagrant` directory and a `tsan11-tsan11rec-docker` directory. The `c11tester-vagrant` directory is a vagrant repository that compiles source codes for C11Tester, LLVM, the companion compiler pass, and benchmarks for C11Tester. The `tsan11-tsan11rec-docker` directory contains benchmarks and a docker image with prebuilt LLVMs for tsan11 and tsan11rec. We had attempted to install tsan11 and tsan11rec in the same VM as C11Tester. However, tsan11rec became significantly slower and some benchmarks were even unrunnable under tsan11rec. So we had to build tsan11, tsan11rec, and benchmarks under the same environment as provided by their artifact documentations.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Testing/race detection algorithm for C/C++ memory model.
- **Program:** C11Tester.
- **Compilation:** Clang 8.0.0 for C11Tester, Clang 3.9.0 for tsan11, and Clang 4.0.0 for tsan11rec.
- **Transformations:** An LLVM pass.
- **Binary:** Modified LLVMs for tsan11 and tsan11rec are included in the docker image.
- **Run-time environment:** Ubuntu 18.04 for C11Tester and Ubuntu 14.04 for tsan11 and tsan11rec.
- **Hardware:** An Intel x86 machine with 6 cores.
- **Execution:** Automated via shell scripts.
- **Metrics:** Execution time, data race detection rate, assertion detection rate.
- **Output:** Numerical results printed in console.
- **Experiments:** GDAX, Iris, Silo, Mabain, the Javascript Engine of Firefox, a broken seqlock, a broken reader-writer lock, and some data structure benchmarks. We measure both the performance (execution time or throughput) and the ability to detect data races and assertions.
- **How much disk space required (approximately)?:** 10G for the VM that contains C11Tester and 15G for the docker container that contains tsan11 and tsan11rec.
- **How much time is needed to prepare workflow (approximately)?:** About 40 minutes for compilation.
- **How much time is needed to complete experiments (approximately)?:** 2 hours for C11Tester, 3 hours for tsan11, and 6.5 hours for tsan11rec.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GNU GPL v2.
- **Data licenses (if publicly available)?:** Varies depending on benchmark.
- **Workflow framework used?:** Vagrant & scripts are provided to automate the measurements.
- **Archived (provide DOI)?:** https://doi.org/10.1145/3410278

A.3 Description

A.3.1 How to Access. The artifact is available at: https://doi.org/10.1145/3410278

A.3.2 Hardware Dependencies. An Intel x86 CPUs with at least 6 cores and at least 40G RAM is required to reproduce results. The VM for C11Tester requires 40G RAM because one particular benchmark (GDAX) consumes 36G RAM under C11Tester. Experiments additionally require CPUs to have Intel VT-d support.

A.3.3 Software Dependencies. C++ Compiler, CMake, Clang, LLVM Compiler Infrastructure, Docker, Vagrant, and VirtualBox.

A.4 Installation

First download the artifact and extract it. The extracted file contains two folders: `c11tester-vagrant` and `tsan11-tsan11rec-docker`.

```
$ cd c11tester-artifact
```

To build C11Tester and benchmarks using Vagrant:

```
$ cd c11tester-vagrant
$ vagrant up
```

The `tsan11-tsan11rec-docker` folder contains a docker image named `tsan11-tsan11rec-image.tar.gz` with prebuilt LLVMs for tsan11 and tsan11rec. For instructions on creating docker containers from the docker image, please see the README.md file in the `tsan11-tsan11rec-docker` repository.

To find the IP address of the container (assuming the container is named tsan11-tsan11rec-container):

```
$ docker inspect tsan11-tsan11rec-container
```

Then use `scp` to copy the `scripts` and `src` directories in the `tsan11-tsan11rec-docker` folder to the container (replace 172.17.0.2 by the container's IP address):

```
$ scp -i insecure_key -r scripts root@172.17.0.2:/data
$ scp -i insecure_key -r src root@172.17.0.2:/data
```

Logging into the container as root (replace 172.17.0.2 by the container's IP address):

```
$ ssh -i insecure_key root@172.17.0.2
```

After logging into the docker container, to build benchmarks for tsan11 and tsan11rec:

```
# ./data/scripts/setup.sh
```

A.5 Experiment Workflow

Scripts are provided to run experiments. To run experiments for C11Tester, logging into the Vagrant VM:

```
$ cd ~/c11tester-benchmarks
$ ./do_test_all.sh
```

To run experiments for tsan11, logging into the docker container:

```
# cd /data/tsan11-benchmarks
# ./do_test_all.sh
```

To run experiments for tsan11rec, inside the same docker container:

```
# cd /data/tsan11rec-benchmarks
# ./do_test_all.sh
```

A.6 Evaluation and Expected Results

Once the workflow is completed, the data race detection rates and assertion rates for data structure benchmarks are printed in the console.

For application benchmarks, the result for each benchmark is written to log files (such `gdax.log` and `silo.log`, etc). These log files are stored in `all-core/` and `single-core/` directories under the benchmark directories `c11tester-benchmarks`, `tsan11-benchmarks`, and `tsan11rec-benchmarks`.

The `do_test_all.sh` script also executes the python script `calculator.py` that prints out result summaries for all of five application benchmarks executed under both the all-core and single-core configurations. Each benchmark directory has this python script. If you wish to regenerate result summaries from log files using the python script, you can first go to one benchmark directory (we will use `c11tester-benchmarks` as an example here):

```
$ cd ~/c11tester-benchmarks
```

and type:

```
$ python calculator.py all-core
```

or

```
$ python calculator.py single-core
```

to print out result summaries for all of five application benchmarks executed under the all-core of single-core configuration.

A.7 Experiment Customization

In the benchmark directory, the two scripts

- `tsan11-missingbug/test.sh`
- `cdschecker_modified_benchmarks/test.sh`

can be customized to run different times by changing the shell variable `TOTAL_RUN`.

The `run.sh` and `app_assertion_test.sh` scripts in the benchmark directory accept an optional argument that specifies how many times the test programs are run. The default is 10 times. Besides that, you can also decide which test program are run by modifying the `TESTS` variable in these two scripts.

A.8 Notes

Tsan11 may occasionally get stuck when testing Silo in the single-core configuration. If this happens, we suggest to rerun Silo individually by customizing the `tsan11-benchmarks/run.sh` script.

REFERENCES

[1] 2020. N4849: Working Draft, Standard for ProgrammingLanguage C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf.

[2] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *Proceedings of the 2014 Symposium on Principles of Programming Languages*. 373–384. http://doi.acm.org/10.1145/2535838.2535845

[3] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless model checking for TSO and PSO. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 353–367. http://link.springer.com/chapter/10.1007%2F978-3-662-46681-0_28

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-from Equivalence Under Sequential Consistency. *Proceedings of ACM on Programming Languages* 3, OOPSLA, Article 150 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360576

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking Under the Release-acquire Semantics. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276505

[6] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems* 36, 2 (July 2014), 7:1–7:74. http://doi.acm.org/10.1145/2627752

[7] F. Eugene Aumson. 2018. gdax-orderbook-hpp. https://github.com/feuGeneA/gdax-orderbook-hpp.

[8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[9] Pete Becker. 2011. ISO/IEC 14882:2011, Information Technology – Programming Languages – C++.

[10] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. 2011. Nitpicking C++ Concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*. 113–124. http://doi.acm.org/10.1145/2003476.2003493

[11] Hans Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *Proceedings of ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. 7:1–7:6. http://doi.acm.org/10.1145/2618128.2618134

[12] Hans-J. Boehm. 2012. Can Seqlocks Get Along with Programming Language Memory Models?. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. 12–20. http://doi.acm.org/10.1145/2247684.2247688

[13] Hans-J. Boehm. 2013. N3786: Prohibiting "out of thin air" results in C++14. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm.

[14] Hans-J. Boehm, Mark Batty, Brian Demsky, Olivier Giroux, Paul McKenney, Peter Sewell, Francesco Zappa Nardelli, et al. 2013. N3710: Specifying the absence of "out of thin air" results (LWG2265). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html.

[15] Hans-J. Boehm, Olivier Giroux, and Viktor Vafeiades. 2018. P0982R0: Weaken Release Sequences. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0982r0.html.

[16] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *Proceedings of the 2007 Conference on Programming Language Design and Implementation*. 12–21. http://doi.acm.org/10.1145/1250734.1250737

[17] Man Cao, Jake Roemer, Aritra Sengupta, and Michael D. Bond. 2016. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. 99–110. http://doi.acm.org/10.1145/2926697.2926700

[18] Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-Directed Stateless Model Checking for SC and TSO. In *Proceedings of the 2015 Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 20–36. http://doi.acm.org/10.1145/2814270.2814297

[19] Changxue Deng. 2018. Mabain: A fast and light-weighted key-value store library. https://github.com/chxdeng/mabain.

[20] Ulrich Drepper. 2013. ELF Handling For Thread-Local Storage. https://akkadia.org/drepper/tls.pdf.

[21] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. 2013. CONCURRIT: A Domain Specific Language for Reproducing Concurrency Bugs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. 153–164. https://doi.org/10.1145/2491956.2462162

[22] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-Aware Java Runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–255. http://doi.acm.org/10.1145/1250734.1250762

[23] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. 237–252. http://doi.acm.org/10.1145/945445.945468

[24] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 121–133. http://doi.acm.org/10.1145/1542476.1542490

[25] Cormac Flanagan and Stephen N. Freund. 2010. Adversarial memory for detecting destructive races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 244–254. http://doi.acm.org/10.1145/1806596.1806625

[26] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 2005 Symposium on Principles of Programming Languages*. 110–121. http://doi.acm.org/10.1145/1040305.1040315

[27] Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *Proceedings of the 2015 Conference on Programming Language Design and Implementation*. 165–174. http://doi.acm.org/10.1145/2813885.2737975

[28] Jeff Huang, Patrick Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*. ACM, 337–348. https://doi.org/10.1145/2594291.2594315

[29] Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 447–461. http://doi.acm.org/10.1145/2983990.2984025

[30] Bengt Jonsson. 2009. State-space exploration for concurrent algorithms under weak memory orderings. *SIGARCH Computer Architecture News* 36, 5 (June 2009), 65–71. http://doi.acm.org/10.1145/1556444.1556453

[31] ISO JTC. 2011. ISO/IEC 9899:2011, Information Technology – Programming Languages – C.

[32] Max Khiszinsky. 2017. https://github.com/khizmax/libcds.

[33] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 17 (December 2017), 32 pages. https://doi.org/10.1145/3158105

[34] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. 96–110. https://doi.org/10.1145/3314221.3314609

[35] Michael Kuperstein, Martin Vechev, and Eran Yahav. 2010. Automatic inference of memory fences. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*. 111–120. http://dl.acm.org/citation.cfm?id=1998496.1998518

[36] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565.

[37] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic Race Detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 443–457. https://doi.org/10.1145/3009837.3009857

[38] Christopher Lidbury and Alastair F. Donaldson. 2019. Sparse Record and Replay with Controlled Scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. 576–593. https://doi.org/10.1145/3314221.3314635

[39] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. 2010. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 210–221. http://doi.acm.org/10.1145/1815961.1815987

[40] Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. Technical Report. (2021). arXiv:2102.07901 [cs.PL]

[41] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015. Concurrency Debugging with Differential Schedule Projections. *SIGPLAN Not.* 50, 6 (June 2015), 586–595. https://doi.org/10.1145/2813885.2737973

[42] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2016. Production-Guided Concurrency Debugging. *SIGPLAN Not.* 51, 8, Article 29 (Feb. 2016), 12 pages. https://doi.org/10.1145/3016078.2851149

[43] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. *SIGARCH Comput. Archit. News* 36, 3 (June 2008), 289–300. https://doi.org/10.1145/1394608.1382146

[44] Brian Norris and Brian Demsky. 2013. CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceedings of the 2013 Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 131–150. http://doi.acm.org/10.1145/2544173.2509514

[45] Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Transactions on Programming Languages and Systems* 38, 3 (May 2016), 10:1–10:51.

[46] Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-thin-air Results. *Proceedings of the ACM on Programming Languages Volume 2 Issue OOPSLA* 2, OOPSLA (Oct. 2018), 136:1–136:29. https://doi.org/10.1145/3276506

[47] Seungjoon Park and David L. Dill. 1999. An Executable Specification and Verifier for Relaxed Memory Order. *IEEE Trans. Comput.* 48, 2 (February 1999), 227–235. http://dx.doi.org/10.1109/12.752664

[48] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA âĂŹ11)*. Association for Computing Machinery, New York, NY, USA, 677âĂŞ694. https://doi.org/10.1145/2048066.2048119

[49] Mathias Guenter Ricken. 2011. *A Framework for Testing Concurrent Programs*. Ph.D. Dissertation. Houston, TX, USA. Advisor(s) Cartwright, Robert. AAI3463989.

[50] Koushik Sen. 2007. Effective Random Testing of Concurrent Programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. 323–332. https://doi.org/10.1145/1321631.1321679

[51] Stephen Tu, Wenting Zheng, and Eddie Kohler. 2015. Silo: Multicore in-memory storage engine. https://github.com/stephentu/silo.

[52] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 18–32. https://doi.org/10.1145/2517349.2522713

[53] Paul Turner. 2013. User-level threads...with threads. https://blog.linuxplumbersconf.org/2013/ocw/system/presentations/1653/original/LPC%20-%20User%20Threading.pdf#4.

[54] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 867–884.

[55] Dmitriy Vyukov. 2011. Relacy Race Detector. http://relacy.sourceforge.net/.

[56] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic Partial Order Reduction for Relaxed Memory Models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 250–259. http://doi.acm.org/10.1145/2737924.2737956

[57] Xinjing Zhou. 2015. Iris: A low latency asynchronous C++ logging library. https://github.com/zxjcarrot/iris.