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

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.

1 INTRODUCTION
The C/C++11 standards added a weak memory model with support for low-level atomics operations [11, 34] that allows experts to craft efficient concurrent data structures that scale better or provide stronger liveness guarantees than lock-based data structures. The potential benefits of atomics can lure both experts and novice developers to use them. However, writing correct concurrent code using these atomics operations is extremely difficult.

Simply executing concurrent code is not an effective approach to testing. Exposing concurrency bugs often requires executing a specific path that might only occur when the program is heavily loaded during deployment, executed on a specific processor, or compiled with a specific compiler. Some prior work helps record and replay buggy executions [45]. Debuggers like Symbiosis [43] and Cortex [44] 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.

1.1 Comparison to Prior Work on Testing C/C++11
Prior work on data race detectors for C/C++11 such as tsan11 [40] and tsan11rec [41] require hb ∪ rf ∪ mo ∪ 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 [49] 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. Eagerly selecting the modification order limits the choices of stores that a load can read from and thus limits the information available to algorithms. 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++ ATOMICS

In this section, we present general background on the C/C++ memory model and then discuss the fragment of the C/C++ memory model that C11Tester supports. The C and C++ standards were extended in 2011 to include a weak memory model that provides precise guarantees about the behavior of both the compiler and the underlying processor. The standards divide memory locations into two types: normal types, which are accessed using normal memory primitives; and atomic types, which are accessed using atomic memory primitives. The standards forbid data races on normal memory types and allow arbitrary accesses to atomic memory types. Accesses to atomic memory types have an optional memory_order argument that explicitly specifies the ordering constraints. Any operation on an atomic object will have one of six memory orders, each of which falls into one or more of the following categories. Like all other tools for the C/C++ memory model, compilers, and work on formalization to our knowledge, C11Tester does not support the consume memory order and thus we omit consume in our presentation.

- **seq-cst**: memory_order_seq_cst — strongest memory ordering, there exists a total order of all operations with this memory ordering. Loads that are seq_cst either read from the last store in the seq_cst order or from some store that is not part of seq_cst total order.

- **release**: memory_order_release, memory_order_acq_rel, and memory_order_seq_cst — when a load-acquire reads from a store-release, it establishes a happens-before relation between the store and the load. Release sequences generalize this notion to allow intervening RMW operations to not break synchronization.

- **acquire**: memory_order_acquire, memory_order_acq_rel, and memory_order_seq_cst — may form release/acquire synchronization.

- **relaxed**: memory_order_relaxed — weakest memory ordering. The only constraints for relaxed memory operations are a per-location total order, the modification order, that is equivalent to cache coherence.

The C/C++ memory model expresses program behavior in the form of binary relations or orderings. We briefly summarize the relations:

- **Sequenced-Before**: The evaluation order within a program establishes an intra-thread sequenced-before (sb) relation—a strict preorder of the atomic operations over the execution of a single thread.

- **Reads-From**: The reads-from (rf) relation consists of store-load pairs $(X, Y)$ such that $Y$ takes its value from $X$. In the C/C++ memory model, this relation is non-trivial, as a given load operation may read from one of many potential stores in the execution.

- **Synchronizes-With**: The synchronizes-with (sw) relation captures the synchronization that occurs when certain atomic operations interact across threads.

- **Happens-Before**: In the absence of memory operations with the consume memory ordering, the happens-before (hb) relation is the transitive closure of the union of the sequenced-before and the synchronizes-with relations.

- **Sequentially Consistent**: All operations that declare the memory_order_seq_cst memory order have a total ordering (sc) in the program execution.

- **Modification Order**: Each atomic object in a program has an associated modification order (mo)—a total order of all
stores to that object—which informally represents an ordering in which those stores may be observed by the rest of the program.

2.1 Example

To explore some of the key concepts of the memory-ordering operations provided by the C/C++ memory model, consider the example in Figure 2, assuming that two independent threads execute the methods threadA() and threadB(). This example uses the C++ syntax for atomics; shared, concurrently-accessed variables are given an atomic type, whose loads and stores are marked with an explicit memory_order governing their inter-thread ordering and visibility properties. In the example, the memory operations are specified to have the relaxed memory ordering, which is the weakest ordering in the C/C++ memory model and allows memory operations to different locations to be reordered.

In this example, a few simple interleavings of threadA() and threadB() show that we may see executions in which \( r_1 = 0 \) and \( r_2 = 1 \), or \( r_1 = 0 \land r_2 = 1 \), but it is somewhat counter-intuitive that we may also see \( r_1 = 1 \land r_2 = 0 \), in which the first load statement sees the second store but the second load statement does not see the first store. While this latter behavior cannot occur under a sequentially-consistent execution of this program, it is, in fact, allowed by the relaxed memory ordering used in the example (and achieved by compiler or processor reorderings).

Now, consider a modification of the same example, where the store and load on variable \( y \) (Line 5 and Line 8) now use memory_order_release and memory_order_acquire, respectively, so that when the load-acquire reads from the store-release, they form a release/acquire synchronization pair. Then in any execution where \( r_1 = 1 \) and thus the load-acquire statement (Line 8) reads from the store-release statement (Line 5), the synchronization between the store-release and the load-acquire forms an ordering between threadB() and threadA()—particularly, that the actions in threadB() after the acquire must observe the effects of the actions in threadA() before the release. In the terminology of the C/C++ memory model, we say that all actions in threadA() sequenced before the release happen before all actions in threadB() sequenced after the acquire.

So when \( r_1 = 1 \), threadB() must see \( r_2 = 1 \). In summary, this modified example allows only three of the four previously-described behaviors: \( \{ r_1 = 0 \land r_2 = 1 \} \), \( \{ r_1 = r_2 = 0 \} \), and \( \{ r_1 = 0 \land r_2 = 1 \} \).

2.2 C11Tester’s 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. [10]:

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 [17]. 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 \( \text{hb} \cup \text{sc} \cup \text{rf} \cup \text{mo} \) is acyclic: Supporting load buffering or out-of-thin-air executions is extremely difficult and the existing approaches introduce high overheads in dynamic tools [20, 47, 48]. 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., \( \text{hb} \cup \text{sc} \cup \text{rf} \), is acyclic [59]. 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 [13, 15, 16, 49].

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 the Appendix. Our fragment of the C/C++ memory model is larger than that of tsan11 and tsan11rec [40, 41]. The tsan11 and tsan11rec tools add a very strong restriction to the C/C++ memory model that requires that \( \text{hb} \cup \text{sc} \cup \text{rf} \cup \text{mo} \) 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 [29]. 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 \( \text{enabled}(s) \) 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 \( \text{next}(s, p) \) is the next transition in thread \( p \) at state \( s \).

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

Figure 3: Pseudocode for C11Tester’s Algorithm

\(^{1}\)The C/C++11 memory model already requires that \( \text{hb} \cup \text{sc} \) is acyclic.
atomic<int> x(0);
void threadB () {
    x.store(2, memory_order_relaxed);
    x.store(1, memory_order_relaxed);
}
void threadA () {
    r1 = x.load(memory_order_relaxed);
}

Figure 4: Bias of a Purely Randomized Algorithm

Figure 3 presents pseudocode for C11Tester’s exploration algorithm. C11Tester calls EXPLORE multiple times—each time generates one program execution. Recall from Section 2 that the thread schedule does not uniquely define the behavior 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 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. This decision also reduces bias in comparison to a purely randomized algorithm.

For example, in Figure 4, under a purely randomized algorithm, the probability that r1 = 1 is much greater than that of r1 = 2, because in order for r1 = 2, the scheduler must schedule threadA() twice before threadB() is scheduled. However, under C11Tester’s strategy, once threadA is scheduled to run, both stores at line 4 and line 5 will be performed consecutively. So when the load is encountered, the may-read-from set (defined in the paragraphs below) either only contains the initial store at line 1 or contains all three stores. Thus, r1 is equally likely to read 1 or 2.

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:

\[ \text{may-read-from}(Y) = \{ X \in \text{stores}(Y) \mid \neg((Y \overset{hb}{\rightarrow} X) \land (Z \overset{hb}{\rightarrow} Y)) \}, \]

where \( \text{stores}(Y) \) 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 [47] 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 \( rb \) 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. If 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, with the additional constraint that \( A \overset{rmw}{\rightarrow} B \). 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.
4.2 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\(^\text{[39]}\) 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 atomic store. The initial *mo-graph* clock vector \(\bot_{CV_A}\) 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:

\[
\bot_{CV_A} = \lambda t. \text{if } t = t_A \text{ then } s_A \text{ else 0}
\]

\[
CV_1 \cup CV_2 \leq \lambda t. \text{max}(CV_1(t), CV_2(t)),
\]

\[
CV_1 \leq CV_2 \leq \lambda t. CV_1(t) \leq CV_2(t).
\]

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 \(A \rightarrow_B \) 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 6 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 node *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 7 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*.\(^{2}\)

\(^{2}\) Events in each thread consist of atomic operations, thread creation and join, mutex lock and unlock, and other synchronization operations.
This theorem states that we can solely rely on mo-graph clock vectors to compute reachability between nodes in mo-graph. We present the theorem and its proof in Section 5.

Figure 7: Helper method for adding a set of edges to the mo-graph

Figure 6: Pseudocode for Updating 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 [47, 48]. 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.

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 X₀. Forming a new rf relation may only cause edges to be created from other nodes to the node representing the store X₀. We denote this set of "other nodes" as ReadPriorSet(X₀) and compute it using the ReadPriorSet procedure in Figure 13. 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(X₀) is reachable from X₀. If so, then having load Y read from store X₀ will introduce a cycle in the mo-graph, so we discard X₀ 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 CORRECTNESS OF MO-GRAth

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 \( G_0 \xrightarrow{mo} G_1 \xrightarrow{mo} \cdots \xrightarrow{mo} G_n \) be a path in a modification order graph \( G \), such that \( CV_{G_0} \leq \ldots \leq CV_{G_n} \). Then if any new edge \( E \) is added to \( G \) using procedures in Figure 6, it holds that

\[
CV'_{G_0} \leq \ldots \leq CV'_{G_n}
\]

for the updated clock vectors. We define \( CV'_{G_i} := CV_{G_i} \) if the values of \( CV_{G_i} \) are not actually updated.

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

By the definition of the union operator, each slot in clock vectors is monotonically increasing when the \( \text{MERGE} \) procedure is called. By the structure of procedure \( \text{AddRMWEdge} \)‘s algorithm, a node \( X \) is added to \( Q \) if and only if this node’s clock vector is updated by the \( \text{MERGE} \) procedure.

Let’s assume that adding the new edge \( E \) updates any of \( CV_{G_0}, \ldots, CV_{G_n} \). 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

\[
CV'_{G_0} \leq \ldots \leq CV'_{G_i}.
\]

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

**Case 1:** Suppose \( CV'_{i} \leq CV_{i} \) 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 \( \text{ADDEDGE} \) procedure, and it holds trivially that

\[
CV'_{j_0} \leq \ldots \leq CV'_{n}.
\]

By line 14 to line 24 in the \( \text{ADDEDGE} \) procedure, we have

\[
CV'_{k} = CV_{k} \cup CV_{k-1}',
\]

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 (5.2), we have

\[
CV'_{0} \leq \ldots \leq CV_{i-1} \leq CV'_{i-1}.
\]

Together with inequality (5.3), only need to show that \( CV'_{j-1} \leq CV_{j-1}' \) to complete the proof.

If \( j_0 = i+1 \), then we are done, because by assumption \( CV'_{i} \leq CV_{i} \). If \( j_0 > i+1 \), then \( CV'_{i} \leq CV_{i} \) and \( CV'_{i+1} \leq CV_{i} \) imply that \( CV'_{i+1} = CV_{i+1} \cup CV_{i} \leq CV_{i+1} \). Based on equation (5.4), we can deduce in a similar way that \( CV'_{i+1} \leq \ldots \leq CV'_{j-1} \leq CV_{j-1} \).

**Case 2:** Suppose \( CV_{j} \not\leq CV_{i} \) for all \( j \in \{i+1\ldots, n\} \). Then by line 14 to line 24 in the \( \text{ADDEDGE} \) procedure, all nodes \( \{C_{j_1}, \ldots, C_n\} \) are added to \( Q \) in the \( \text{ADDEDGE} \) 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 (5.2), we have \( CV'_{0} \leq \ldots \leq CV_{n} \).

Now suppose the newly added edge \( E \) is a rmw edge. If \( E : X \rightarrow_{rmw} C_i \) where \( i \in \{0\ldots n\} \) and \( X \) is some node not in path \( P \), then the path \( P \) remains unchanged and \( \text{AddEDGE}(X,C_i) \) is called. Then the above proof shows that inequality (5.1) holds. If \( E : C_i \rightarrow_{rmw} X \), then \( C_i \rightarrow_{mo} C_{i+1} \) is migrated to \( X \rightarrow_{mo} C_{i+1} \) by line 3 to line 7 in the \( \text{AddRMWEdge} \) procedure, and \( C_i \rightarrow_{mo} X \) is added.

If \( X \) is not in path \( P \), then path \( P \) becomes

\[
C_0 \rightarrow_{mo} \ldots \rightarrow_{mo} C_i \rightarrow_{mo} X \rightarrow_{mo} C_{i+1} \rightarrow_{mo} \ldots \rightarrow_{mo} C_n.
\]

Since \( \text{AddEDGE}(C_i,X) \) is called, the same proof in the case without \( \text{rmw} \) 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.

\[\square\]

Let \( \bar{x} = (x_1, x_2, \ldots, x_n) \). We define the projection function \( U_i \) that extracts the \( i \)-th position of \( \bar{x} \) as \( U_i(\bar{x}) = x_i \) 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 \( S = \{A_1, A_2, \ldots, A_n\} \) be the sequence of stores performed by thread \( i \) with sequence numbers \( \{s_1, s_2, \ldots\} \), respectively. Suppose that there is a point in time in a terminating execution such that the first store \( A_n \) in the sequence with \( U_i(CV_{A_n}) > s_A \) appears. Sequence numbers are strictly increasing and by the \( \text{MERGE} \) 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 \( s_A \), \( CV_{A_n} \) 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 \( U_i(CV_{A_n}) \) is unique for all stores \( A_j \) in the sequence \( S \) other than \( A_n \). Then \( \perp_{CV_{A_n}} \geq \perp_{CV_{A_n}} \). 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 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, we will eventually deduce that \( X \) is modification ordered after some thread \( i \). Hence, we would have a cycle in \( G \), a contradiction.

\[\square\]

**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 \( A \rightarrow_{mo} C_1 \rightarrow_{mo} \ldots \rightarrow_{mo} C_n \rightarrow_{mo} B \) be the shortest path \( P \) from \( A \) to \( B \) in graph \( G \). To simplify notation, \( X \rightarrow_{mo} Y \) is abbreviated as \( X \rightarrow Y \) in the following.

As the \( \text{AddRMWEdge} \) procedure calls the \( \text{ADDEDGE} \) procedure to create an \( mo \) edge, we can assume that all the \( mo \) edges in \( P \) are created by directly calling \( \text{ADDEDGE} \).
**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 $\text{AddEdge}(A, B)$, $CV_B$ was merged with $CV_A$ in line 14 of the $\text{AddEdge}$ 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 $CV_{C_1} \leq CV_A$. When $C_1 \rightarrow B$ was formed, $CV_B$ was merged with $CV_{C_1}$ and $CV_{C_1} \leq CV_A$. 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

$$CV_{C_0} \leq \ldots \leq CV_{C_i},$$  

$$CV_{C_{i+1}} \leq \ldots \leq CV_{C_k+1} \leq CV_{B}.$$  

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 $CV_{C_k} \leq CV_{C_{i+1}}$, and

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

\[\square\]  

**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 $CV_B = (\ldots, s_B, \ldots)$ where both $s_A$ and $s_B$ are in the $i^{th}$ position. By assumption, we have $0 < s_A \leq s_B$.

Since $B$ is not performed by thread $i$, we have $U_i(\cup CV_B) = 0$. 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.  

\[\square\]

### 6 OPERATIONAL MODEL

We present our operational model with respect to the tsan11 [40] core language described by the grammar in Figure 8. A program is a sequence of statements. $\text{LocNA}$ and $\text{LocCA}$ 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, writing a new value, joining a thread via its thread handle, and atomic statements. The symbol $e$ denotes an empty statement. Atomic statements denoted by $\text{StmtA}$ include atomic loads, store, rmws, fences. An rmw takes a function, $F$, to implement rmw operations, such as atomic_fetch_and_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.

**Figure 8: Syntax for our core language**

6.1 Happens-Before Clock Vectors

We next discuss the various happens-before clock vectors that C11Tester uses to implement happens-before relations. Figure 9 presents our algorithm for updating clock vectors used to track happens-before relations for atomic loads, stores, rmws, and fences. The union operator $\cup$ 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 $CV_t$, and release- and acquire-fence clock vectors $F_{rel}^t$ and $F_{acq}^t$. The algorithm also records a reads-from clock vector $R_{t}$ 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 \(C, RF, p_{rf}, p_{raw}\) and \(RF\) 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 [40]. 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 \(C_t\) and a second \(reads-from\) clock vector \(RF_S\) 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 \(RF_A\) as the union of: (1) the store \(B\)’s \(reads-from\) clock vector \(RF_B\) and (2) the RMW \(A\)’s \(current\) thread clock vector \(C_{t_A}\) if \(A\) is a release. When a load-acquire \(A\) reads from a store-release or RMW, C11Tester computes the load-acquire’s \(current\) thread clock vector as the union of: (1) the load-acquire’s \(current\) thread clock vector \(C_{t_A}\) 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 released stores into store-releases, but the happens-before relation is established at the fence-release. C11Tester maintains a release fence clock vector \(F^rf\) for each thread and uses this clock vector when computing the clock vector for release sequences. Acquire fences effectively make previous released 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 \(F^acq\) 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.

### 6.2 Formal Operational Model

Figure 10 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 8. The initial state of the system has empty mappings \(ALocs\) and \(NALocs\), and \(FenceInfo\), only one thread representing the main function, and an empty \(mo-graph\).

### 6.3 Operational Semantics

Figures 11 to 13 present state transitions and related algorithms for our operational model. A system under evaluation is a triple of the form \((\Sigma, ss, T)\), where \(\Sigma\) 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 \(\Sigma\) to become outdated. However, the updated \(T\) will replace the old copy in \(\Sigma\) when the thread switching function \(\delta\) is called at the end of each atomic statement. The \(mo-graph\) is a data structure in \(State\) and represented as \(\Sigma.M\). The \(mo-graph\) has methods \text{merge}, \text{addEdge}, \text{addRMWEdge}, and \text{addEdges} described in Figure 6 and Figure 7.

Figure 11 shows semantics for atomic statements. Every time an atomic statement is encountered, a corresponding \text{LoadElem},

---

**Figure 9:** Semantics for tracking happens-before clock vectors for atomic loads, stores, RMWs, and fences. An RMW also triggers a load rule initially.
StoreElem, RMWElem, or FenceElem is created with the sequence number auto-assigned. The process of assigning sequence numbers are omitted in Figure 11. Function calls [LOAD], [STORE], and [FENCE] invokes the corresponding inference rules for updating clock vectors described in Figure 9 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], 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 12, 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 selected store does not create a cycle in the mo-graph. Then, the mo-graph is updated using the procedure AndRMWEdge, and the atomic RMW rules is finally finished by invoking an atomic load.

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

\[ \text{READ} \]
\[ \text{LOAD}(t, a) \] returns the list of stores and RMWs performed by thread \( t \) at location \( a \);
\[ \text{LOAD}_{\text{stores}}(t, a) \] returns the list of loads, stores, and RMWs performed by thread \( t \) at location \( a \);
\[ \text{lastlist}(t) \] returns the element with the largest sequence number in the list, excluding null elements;
\[ \text{get_write}(A) \] returns \( A \) if \( A \) is an atomic store or RMW and returns \( A \) otherwise.

![Figure 10: Operational State](image)

**Figure 10: Operational State**

StoreElem, RMWElem, or FenceElem is created with the sequence number auto-assigned. The process of assigning sequence numbers are omitted in Figure 11. Function calls [LOAD], [STORE], [RMW], and [FENCE] invokes the corresponding inference rules for updating clock vectors described in Figure 9 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 12, 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 selected store does not create a cycle in the mo-graph. Then, the mo-graph is updated using the procedure AndRMWEdge, and the atomic RMW rules is finally finished by invoking an atomic load.

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

\[ \text{FENCE} \]
\[ \text{LoadElem} \]
\[ \text{FenceInfo} \]
\[ \text{LoadElem}(t, a) \] returns the list of stores and RMWs performed by thread \( t \) at location \( a \);
\[ \text{FenceInfo}(t, a) \] returns the list of stores and RMWs performed by thread \( t \) at location \( a \);
\[ \text{order} \]
\[ \text{last_sc_fence}(t) \] returns the last seq_cst fence in thread \( t \);
\[ \text{last_sc_store}(a, S) \] returns the last seq_cst store performed at location \( a \) and is different from \( S \);
\[ \text{sc_fences}(t) \] returns the list of seq_cst fences performed by thread \( t \);
\[ \text{sc_stores}(t, a) \] returns the list of seq_cst stores and RMWs performed by thread \( t \) at location \( a \);
\[ \text{seq_cst} \]
\[ \text{seq_cst}(t) \] returns the list of stores and RMWs performed by thread \( t \) at location \( a \);
\[ \text{seq_cst}(t, a) \] returns the list of loads, stores, and RMWs performed by thread \( t \) at location \( a \);
\[ \text{seq_cst}(t) \] returns the list of loads, stores, and RMWs performed by thread \( t \) at location \( a \);
\[ \text{seq_cst}(t, a) \] returns the list of loads, stores, and RMWs performed by thread \( t \) at location \( a \);
\[ \text{seq_cst}(t) \] returns the list of loads, stores, and RMWs performed by thread \( t \) at location \( a \);
\[ \text{seq_cst}(t, a) \] returns the list of loads, stores, and RMWs performed by thread \( t \) at location \( a \);
We next present several aspects of the C11Tester implementation. We make our axiomatic model precise and prove the equivalence of our operational and axiomatic models in Section A of the Appendix.

### 7 IMPLEMENTATION

We next present several aspects of the C11Tester implementation. Section 7.1 presents C11Tester’s support for limiting memory usage. Section 7.2 presents C11Tester’s support for mixed mode accesses to a memory location. Section 7.3 discusses the overheads of different approaches to controlling thread schedules. Section 7.4 describes how C11Tester implements thread local storage with fiber-based scheduling. Section 7.5 presents C11Tester’s support for static initializers. Section 7.6 presents C11Tester’s support for repeated execution.

#### 7.1 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 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 $A_t \rightarrow_{SB} L_t$. If action $S$ is a store that either happens before $A_t$ or $A_t$ then any new loads from the same memory location must either read from $S$ or some store that is modification ordered after $S$. 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:

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

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.

7.2 Supporting Mixed Access Modes

The C/C++ memory model is silent on the semantics of how atomic accesses interact with non-atomics accesses to the same memory location. Researchers have recognized this as a serious limitation of the standard [8]. It is necessary to handle mixtures of atomics and non-atomics in C11Tester for three reasons: (1) atomic_init is implemented in the header files as a non-atomic store and may race with concurrent atomic accesses to the same memory location, (2) memory can be reused in C/C++, e.g., via malloc and free, and the new use may use a memory location for a different purpose, and (3) C/C++ programs may use non-atomic accesses to copy memory that contains atomics, e.g., via realloc or memcpy. C11Tester thus supports non-atomic operations that access the same memory location as an atomic as they must be tolerated provided that the accesses are ordered by the happens-before relation. If the accesses conflict and are not ordered by happens-before, C11Tester reports a data race.

Handling non-atomic stores poses a challenge. For performance reasons, it is important to implement non-atomic stores as simple writes to memory. But if a non-atomic store is later read by an atomic load, then C11Tester must include that non-atomic store in the modification order graph and other internal data structures. The challenge is that by the time C11Tester observes the atomic load, it has lost information about the non-atomic store.

C11Tester uses a FastTrack [27]-like approach to race detection. It maintains a 64-bit shadow word for each byte of memory. The shadow word either contains 25-bit read and write clocks and 6-bit read and write thread identifiers or a reference to an expanded access record. We use one bit in the shadow word to record whether the last store to the address was from a non-atomic or an atomic store. If C11Tester performs an atomic access to a memory location that was last written to by a non-atomic store, C11Tester creates a special non-atomic access record and adds the access to the modification order graph.

Many applications also contain legacy libraries that use pre-C/C++11 atomic operations such as LLVM intrinsics and volatile accesses. C11Tester supports converting such volatile accesses into atomic accesses (with a user specific memory order) to allow code that incorporates legacy libraries to execute.

7.3 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 [57] 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. Figure 14 reports the results of these experiments. We ran each experiment in two configurations: (1) in the all-core configuration the microbenchmark could use all 4 hardware cores and (2) in the single-core configuration the microbenchmark was pinned to a single hardware thread.

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 fences 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. We report context switches with this system call in the “w/ tls” entries.

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

7.4 Thread Context Borrowing

A major challenge with implementing fibers is supporting thread local storage. The specification for thread local storage on x86-64 [23] 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.

7.5 Static Initializers

Static initializers in C++ can and do create threads, perform atomic operations, and call arbitrary functions in the C++ and pthread libraries. C11Tester guards access to itself with initialization checks. In the first call to a C11Tester routine, C11Tester initializes itself and converts the current application thread into a fiber context. It then takes control of the execution and controls the remainder of the program execution. This allows C11Tester to support programs that perform arbitrary operations in their static initializers.

7.6 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.

8 EVALUATION

We compare C11Tester with both tsan11rec, a race detector that supports controlled execution [41] and tsan11 [40], 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 [47].

We were not able to build tsan11rec and tsan11 directly on our machine due to dependencies on legacy versions of software. Nevertheless, we compiled tsan11rec and tsan11 inside two docker containers whose base images were both Ubuntu 14.04 LTS. The tsan11rec-instrumented benchmarks were compiled with Clang v4.0 revision 286346, the tsan11-instrumented benchmarks were compiled with Clang v3.9 revision 375507, and the C11Tester-instrumented benchmarks were compiled with Clang v8.0 revision 346999.

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.

8.1 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 \( \text{hb} \cup \text{rf} \cup \text{mo} \cup \text{sc} \).

Seglock. We took the seqlock implementation from Figure 5 of Hans Boehm’s MSPC 12 paper [14], 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.

8.2 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 [67], a low-latency C++ logging library; Mabain [22], a key-value store library; Silo [55, 56], a multicore in-memory storage engine; and the Firefox JavaScript engine.
To make our results as reproducible as possible, we tested the JavaScript engine using the offline version of JSBench v2013.1.

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 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 (\texttt{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 15 summarizes 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.

Table 3 presents the number of atomic operations and normal accesses to shared memory locations executed by C11Tester for each benchmark. As the compiler pass of C11Tester was adapted from the LLVM ThreadSanitizer pass, the number of atomic operations and normal accesses to shared memory executed by tsan11 and tsan11rec should be relatively similar, except for the two throughput-based benchmarks — Silo and GDAX, as the amount of work depends on how fast a tool is.

**Silo.** Silo [55, 56] is an in-memory database that is designed for performance and scalability for modern multicore machines. The test driver we used is \texttt{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 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

---

1. https://ftp.mozilla.org/pub/firefox/releases/50.0.1/source/
2. 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.

| Test          | Single-core Configuration | All-core Configuration | Measurement          |
|---------------|---------------------------|------------------------|----------------------|
|               | C11Tester                 | tsan11rec              | tsan11               | Throughput (ops/sec) |
| Silo          | 15287 (0.45%)             | 436 (2.52%)            | 5496 (4.54%)         | 30279 (1.17%)        |
| GDX           | 2953 (1.80%)              | 693 (0.97%)            | 5982 (0.12%)         | 6076 (0.04%)         |
| Mabain        | 577 (0.25%)               | 593 (0.98%)            | 15790 (0.12%)        | 5226 (0.04%)         |
| Iris          | 8.95 (1.46%)              | 31.31 (0.89%)          | 4.873 (1.64%)        | 8.68 (0.22%)         |
| JSBench       | 1835 (0.26%)              | 2522 (1.41%)           | 867.8 (0.21%)        | 836 (0.35%)          |

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  |
| barrier       | 4ms       | 76.6%     | 19ms   | 36.4%  |
| chase-lev-deque| 2ms       | 94.6%     | 7ms    | 8.0%   |
| dekker-fences | 2ms       | 21.6%     | 10ms   | 41.4%  |
| linuxrwlocks  | 2ms       | 86.2%     | 10ms   | 53.4%  |
| mcs-lock      | 3ms       | 89.4%     | 11ms   | 71.4%  |
| mpmc-queue    | 4ms       | 59.4%     | 10ms   | 58.2%  |
| ms-queue      | 4ms       | 100.0%    | 136ms  | 100.0% |
| Average       |           | 75.4%     |        | 51.3%  |

Table 3: The number of atomic operations (including synchronization operations such as mutex and condition variable operations) and normal accesses to shared memory locations executed in each benchmark by C11Tester.

| Test         | Silo | GDX     | Mabain | Iris | JSBench |
|--------------|------|---------|--------|------|---------|
| # normal memory accesses | 63.7M | 408.3M  | 77.1M  | 35.1M | 5747M   |
| # atomic operations    | 11.3M | 44.9M   | 2.98M  | 4.8M | 8.02M   |

Mabain, Mabain is a lightweight key-value store library [22]. 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.

GDX. GDX [7] implements an in-memory copy of the order book for the GDX cryptocurrency exchange using a lock-free skip list with garbage collection from the libcds library [35]. The original GDX fetches data from a server, but we have recorded input data from a previous run and modified GDX to read local data. All tools reported data races in GDX.

In our experiment, GDX 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 [67] 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_1fringbuffer.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

work on model checking for sequential consistency has developed
with rates higher than tsan11rec in 4 benchmarks and tsan11 in 5
were added to 6 of these benchmarks to induce some variability
which contains 25 JavaScript benchmarks, sampled from real-world
To assess the ability of C11Tester to discover data races, we also
JavaScript shell with Thread Sanitizer given by the developers of
Firefox. 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 arith-
metric mean of all 25 benchmarks over 10 runs, and then took the
geometric means of the 25 arithmetic mean, as reported in Table 1.

8.3 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 [40]. We replicated the
same timing strategy used in [40] 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.

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

Model Checkers. In the context of weak hardware memory mod-
elns, researchers have developed stateful model checkers [33, 38, 50].
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++
current data structures [47, 48]. 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, 36, 37]. 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 [12], and Cpp-
Mem [10] are intended to help understand the behaviors of memory
models and do not scale to real-world data structures.

CHESS [46] is designed to find and reproduce concurrency bugs
in C, C++, and C#. It systematically explores thread interleavings.
However, it can miss concurrency bugs for C/C++ as it does not
reorder memory operations. Line-Up [19] extends CHESS to check
for linearization. Like CHESS, it can miss bugs that are exposed by
reordering of memory operations. The Inspect tool combines
stateless model checking and stateful model checking to model
check C and C++ code [61–64]. The Inspect tool checks code using
the sequential consistency model rather than the weaker C/C++
memory model and therefore may miss concurrency bugs arising
from reordered memory operations.

Dynamic Partial Order Reduction [29] and Optimal Dynamic
Parti onal Order Reduction [2] seek to make stateless model checking
more efficient by skipping equivalent executions. Maximal causal
reduction [30] further refines the technique with the insight that it
is only necessary to explore executions in which threads read differ-
ent values. Recent work has extended these algorithms to handle the
TSO and PSO memory models [3, 32, 65]. SATCheck further devel-
ops partial order reduction with the insight that it is only necessary
to explore executions that exhibit new behaviors [21]. CheckFence
checks concurrent code by translating it into SAT [18]. Despite
these advances, model checking faces fundamental limitations that
prevent it from scaling to full applications.

Fuzzers. The Relacy race detector [60] explores thread interleav-
ings 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 or-
der; it cannot simulate (legal) executions in which the modification
order is inconsistent with the execution order.

Industry tools like the IBM ConTest tool support testing con-
current software. They work by injecting noise into the execution
schedule [58]. While such tools may increase the likelihood of find-
ing races, they do not precisely control the schedule. They also do
not handle weak memory models like the C/C++ memory model.
Adversarial memory increases the likelihood of observing weak
memory system behaviors for the purpose of testing [28]. In the
context of Java, prescient memory can simulate some of the weak
behaviors allowed by the Java memory model [20]. Prescient mem-
ory 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 [52]. Concurr is a DSL designed to help reproduce concurre-
cy bugs [24]. Developers write code in a DSL to help guide
Concurr to a bug reproducing schedule. CalFuzzer more uniformly
samples non-equivalent thread interleavings by using techniques in-
spired by partial order reduction [54]. 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 [25–
27, 31, 42]. These tools typically verify that all accesses to shared
data are protected by a locking discipline. They miss higher-level se-
notropic races that occur when the locks allow unexpected orderings
that produce incorrect results.
We have presented C11Tester, which implements a novel approach to controlling scheduling in C/C++11 memory model [40]. 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 lets developers 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.

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.

REFERENCES

[1] N4849: Working draft, standard for programming language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf, Jan 2020.
[2] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagounas. Optimal dynamic partial order reduction. In Proceedings of the 2014 Symposium on Principles of Programming Languages, pages 373–384, 2014.
[3] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagounas. 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, pages 353–367, 2015.
[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagounas. Optimal stateless model checking for reads-from equivalence under sequential consistency. Proceedings of ACM on Programming Languages, 3(OOPSLA):150:1–150:29, October 2019.
[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. Proceedings of the ACM on Programming Languages, 2(OOPSLA):135:1–135:29, October 2018.
[6] Jade Algaze, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Transactions on Programming Languages and Systems, 36(2):7:1–7:4, July 2014.
[7] F. Eugene Ansoon. gdxv-orderbook-hpp. https://github.com/eruGenea/gdxv-orderbook-hpp, June 2011.
[8] Mark Batty, Kayvan Memarian, Knydylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In European Symposium on Programming Languages and Systems, pages 283–307. Springer, 2015.
[9] Mark Batty, Scott Owens, Sumit Sarkar, Peter Sewell, and Tjark Weber. https://www.cl.cam.ac.uk/~pes20/cpp/cmm.pdf, 2011.
[10] Mark Batty, Scott Owens, Sumit Sarkar, Peter Sewell, and Tjark Weber. Mathemati-
ing C++ concurrency. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2011.
[11] Pete Becker. ISO/IEC 14882:2011, Information technology – programming lan-
guages – C++, 2011.
[12] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Sumit Sarkar. Nitpicking C++ concurrency. In Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming, pages 113–124, 2011.
[13] Hans Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air re-
sults. In Proceedings of ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, pages 7–16, June 2014.
[14] Hans-J. Boehm. Can selocks get along with programming language memory models? In Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, pages 12–20, June 2012.
[15] Hans-J. Boehm. N3786: Prohibiting “out of thin air” results in C++14. http://www.

ACM Transactions on Programming Languages and Systems, 36(2):7:1–7:4, July 2014.

Conference ’17, July 2017, Washington, DC, USA.
A PROOF OF EQUIVALENCE BETWEEN OPERATIONAL AND AXIOMATIC MODELS

We first present the formalization of our axiomatic model. We then show how to lift a trace produced by our operational model to an axiomatic-style execution and prove that lifting the set of traces produced by our operational model gives rise to executions that exactly match the executions allowed by our restricted axiomatic model. We refer to the axiomatic model that is based on the C++11 memory model but incorporates the first and the third changes described in Section A.1 below as the modified C++11 memory model. We refer to the axiomatic model presented in Section A.1 as the restricted axiomatic model or our axiomatic model. Our axiomatic model is stronger than the modified C++11 memory model.

We also introduce some notations in this Section. Let $P$ be a program written in our language described in Figure 8 of the paper. Let $Consistent(P)$ denote the set of executions allowed by the modified C++11 memory model, $rConsistent(P)$ denote the set of executions allowed by our axiomatic memory model, and traces($P$) denote the set of traces produced by our operational model. We use $\sigma$ to denote an individual trace, which is a finite sequence of state transitions, i.e.,

\[ \sigma = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_m \]

The set of axiomatic-style executions obtained by lifting a trace $\sigma$ is denoted as $\text{lift}(\sigma)$. Lifting a trace gives rise to a set of executions because the extension of the $mo$-graph is not unique, as explained in Section A.2. We write $\text{lift}(\sigma)$ when we wish to refer to a single execution in $\text{lift}(\sigma)$.

A.1 Restricted Axiomatic Model

We present the formalization of our axiomatic model by making following changes to the formalization of Batty et al. [9, 10]:

1) Use the C/C++20 release sequence definition: This corresponds to changing the definition of “rs_element” (in Section 3.6 of their formalization) by dropping the “same_thread a rs_head” term.

2) Add $hb \cup sc \cup rf$ is acyclic: This is implemented by adding the following to their formalization in Section 3:

- Add the following definition:
  
  "%acquire_hb_sc_rf actions hb sc rf = irrefl actions tc (hb \cup sc \cup rf)"

- Add the following term to the conjunct in Section 3.11:
  
  "%acyclic_hb_sc_rf Xo.actions hb Xw.sc Xw.rf &"

3) Strengthen consume atomics to acquire: This is implemented with the following two changes in Section 2.1:

- Make is_consume always false.
- Change the MO_CONSUME case for is_acquire a to "is_read a V is_fence a".

Therefore, the set of executions allowed by our restricted axiomatic model can be expressed as:

\[ rConsistent(P) = Consistent(P) \land \text{acyclic}(hb \cup sc \cup rf) \]

Since we do not consider the consume memory ordering, the $hb$ relation is the transitive closure of $sb$ and $sw$. In the simple language described in Figure 8, an $sw$ edge is either an additional synchronizes with (asw) edge, an $rf$ edge, or a combination of $sb$ and $rf$ edges (release-acquire synchronization involving fences). Therefore, we have

\[ sb \cup asw \subseteq hb \subseteq sb \cup asw \cup rf, \]

and we can deduce that

\[ sb \cup asw \cup sc \cup rf \subseteq hb \cup sc \cup rf \subseteq sb \cup asw \cup sc \cup rf, \]

which implies that $hb \cup sc \cup rf = sb \cup asw \cup sc \cup rf$. Therefore,

\[ rConsistent(P) = Consistent(P) \land \text{acyclic}(sb \cup asw \cup rf \cup sc) \]

A.2 Lifting Traces

We need to extend our operational states with auxiliary labels in order to track events. We define a label as $Label \triangleq \{1, 2, 3, ... \} \cup \{\bot\}$. We extend $ThrState$ with a last sequentially before ($lsb$) label and a last additional synchronizes with (lasw) label to track $sb$ and $asw$ relations, and $State$ with a last sequentially consistent (lsc) label to track $sc$ relations. The $lasw$ label stores the last instruction the parent thread performs before forking a new thread. Each load, store, or RMW element will have an event label representing its unique event id.

The Load, Store, RMW, and Fence in Figure 8 correspond to atomic load, store, RMW, and fence events in an execution. Loads from and stores to LocNA correspond to non-atomic reads and writes in an execution. The event labels inside $LoadElem$, $StoreElem$, $RMWElem$, and $FenceElem$ will match the event ids of their corresponding events in the execution.

We will describe how to lift traces in the following. The instructions referred to below are the ones that create events in an execution.

When a thread $T$ performs an instruction and $T.lsb \neq \bot$, an $sb$ edge is created from $T.lsb$ to the current instruction. Similarly, when a seq_cst instruction is performed and $T.lsc \neq \bot$, an $sc$ edge is created from $T.lsc$ to the current seq_cst instruction. The $rf$ edges can be created by inspecting traces and checking the $rf$ fields of $LoadElems$ and $RMWElems$.

The $asw$ edges can be created in two ways: a) When a thread $T$ performs a Fork instruction, creating a new thread $T'$, the new thread $T'$ stores $T.lsb$ in the field $T'.lasw$. Then when $T'$ performs an instruction and $T'.lasw \neq \bot$, an $asw$ edge is created; b) When thread $T'$ has finished, the parent thread $T$ performs a Join instruction with the thread id of $T'$. If $T'.lsb \neq \bot$, an $asw$ edge is created when $T$ performs the next instruction.

The $mo$-graph of a trace models $mo$ relations in an execution. However, the $mo$-graph at a memory location may sometimes only capture a partial order over all stores to the location. We know that a partial order can always be extended to a total order. Therefore, we can perform a topological sort of the $mo$-graph at each memory location, and extend the obtained $mo$ relations to total orders if necessary. Then, we can create $mo$ edges in the lifted trace based on the extended $mo$ relations. Since linear extensions of a partial order are not unique, lifting a trace may give rise up multiple axiomatic-style executions.

Since the operational model requires atomic loads to read from stores that have been processed by the operational model, and the $sb$, $sc$, and $asw$ edges are constructed to be consistent with the program order in the lifting process, we have that $(sb \cup asw \cup rf \cup sc)$ is acyclic. We summarize this relation as the Lemma below.
Lemma 4. Let \( P \) be an arbitrary program and \( \sigma \in \text{traces}(P) \). Then for any \( E \in \text{lift}(\sigma) \), the union of \( \text{sh}, \text{asw}, \text{sc}, \) and \( \text{rf} \) edges in \( E \) is acyclic.

### A.3 Equivalence of Axiomatic and Operational Models

Our goal is to show that for an arbitrary program \( P \), the set of executions allowable by our restricted axiomatic model is equivalent to the set of executions we get by lifting traces that our operational model can produce, i.e.,

\[
\forall P \forall E \forall E \in \text{rConsistent}(P) \Leftrightarrow \exists \sigma \in \text{traces}(P). E \in \text{lift}(\sigma).
\]

Definition 1. Let \( P \) be an arbitrary program and \( E \) be an axiomatic-style execution of \( P \). We define \( E \) as the execution that only contains \( \text{sh}, \text{asw}, \text{sc}, \) and \( \text{rf} \) edges in \( E \) together with events in \( E \).

Given an execution \( E \in \text{rConsistent}(P) \) that consists of \( n \) events, \( E \) is a DAG, which can be topologically sorted to give an ordering, \( e_1, ..., e_n \), that is consistent with the order that events are added to \( E \) as the program is running.

\[
e_1: v.\text{store}(1) \xrightarrow{\text{rf}} e_3: v.\text{fetch}_\text{add}(1)
\]

\[
e_2: v.\text{store}(3)
\]

![Figure 17: Example where the partial execution graph \( E_2 \) misses mo edges](image)

Based on the topological sort, we define the partial execution graph \( E_i \) of \( E \) as the execution that consists of the first \( i \) events, together with \( \text{sh}, \text{asw}, \text{sc}, \text{rf} \), and \( \text{mo} \) edges such that the sources and destinations of included relations are events in \( E_i \), where \( 0 \leq i \leq n \). \( E_0 \) is defined as the empty execution.

Since we do not consider \( \text{mo} \) edges in the topological sort, some partial execution graph \( E_i \) may contain events where there exists modification ordering between them in \( E \) but the \( \text{mo} \) edges are missing in \( E_i \). For example, in Figure 17, \( \{e_1, e_2, e_3\} \) is a valid topological sort of \( E \), and we also have \( e_1 \xrightarrow{\text{mo}} e_3 \xrightarrow{\text{mo}} e_2 \), but no \( \text{mo} \) edge is present in the partial execution graph \( E_2 \). To deal with this issue, we define the modification order at an atomic location \( M \) in partial execution graph \( E_i \) as a total order \( S^M_i \) over \( E_i \) that modifies \( M \) such that \( S^M_i \) is consistent with the modification order at location \( M \) in the complete execution graph \( E \). For atomic modifications \( X \) and \( Y \) in \( S^M_i \), if \( X \text{ precedes } Y \), then we write \( X \xrightarrow{\text{sm}} Y \).

The equivalence proof between axiomatic and operational models contains two directions, and we will break it down into Lemma 5 and Lemma 6. In the proofs of the two lemmas below, for a store event \( e_j \) in \( E \), there is a corresponding node in the \text{mo-graph} of the equivalent trace in the operational model. Although \( e_j \) is technically an event in an axiomatic execution, we sometimes abuse the notation and use \( e_j \) to refer to the corresponding node in the \text{mo-graph} of the equivalent trace. If the node corresponding to \( e_j \) is modification ordered before the node corresponding to \( e_i \) in a \text{mo-graph}, then we may say \( e_j \xrightarrow{\text{mo}} e_i \) exists in the \text{mo-graph}.

In the forward direction (Lemma 5), the proof strategy is to apply induction on the construction of partial execution graphs. More specifically, if \( E_i \) is a partial execution graph of \( E \) such that there exists at least one trace \( \sigma_i \) where \( E_i \in \text{lift}(\sigma_i) \), then when \( E_i \) is extended to \( E_{i+1} \), we can construct a trace \( \sigma_{i+1} \) that is an extension of \( \sigma_i \) such that \( E_{i+1} \in \text{lift}(\sigma_{i+1}) \).

Lemma 5. Let \( P \) be an arbitrary program, and \( E \in \text{rConsistent}(P) \) be an execution. Then there exists a trace \( \sigma \in \text{traces}(P) \) such that \( E \in \text{lift}(\sigma) \).

Proof. Let \( P \) be an arbitrary program, and \( E \in \text{rConsistent}(P) \) be an execution. Let \( e_1, ..., e_n \) be a topological sort of \( E \). We will prove by induction on the construction of partial execution graphs of \( E \) as described in our proof strategy.

Base case: When \( i = 0 \), \( E_0 \) is the empty execution. We can take the initial trace \( \sigma_0 \) that is the initial state of the program \( P \) without any transitions, and \( E_0 \in \text{lift}(\sigma_0) \). At this point the \text{mo-graph} is empty as well.

Inductive step: Suppose that for some \( i < n \), we have constructed the partial execution graph \( E_i \) and that there exists some trace \( \sigma_i \) such that \( E_i \in \text{lift}(\sigma_i) \). We will assume throughout the proof that the instructions used for the state transitions match the events being added. Also note that each visible instruction adds a context switch afterward, so there is always the ability to change to the required thread. We will extend \( E_i \) by adding the next event \( e_{i+1} \), and show that we can construct a \( \sigma_{i+1} \) that is the extension of \( \sigma_i \) such that \( E_{i+1} \in \text{lift}(\sigma_{i+1}) \). In the following, we will first analyze five different cases of incoming edges to \( e_j \), and then show that we can extend the \text{mo-graph} of \( \sigma_{i+1} \) to the modification orders in \( E_{i+1} \).

A.1. If \( e_{i+1} \) has no incoming edges, then \( e_{i+1} \) must be the event \( e_1 \), corresponding to the first visible instruction in the initial thread. Because if \( e_{i+1} \neq e_1 \), there must be an \text{sh} edge or an \text{asw} edge coming to \( e_{i+1} \). We can construct \( \sigma_{i+1} = \sigma_1 \) by letting the initial thread execute until the instruction corresponding to \( e_1 \) is executed.

A.2. If \( e_{i+1} \) has an incoming \text{sh} edge from some event \( e_j \), then from the last state of \( \sigma_i \), we must switch to the thread that executes \( e_j \) and continue until the instruction corresponding to \( e_{i+1} \) is executed, to obtain \( \sigma_{i+1} \).

A.3. If \( e_{i+1} \) has an incoming \text{asw} edge, then we have two scenarios: \( e_{i+1} \) is the first event in a newly created thread; or a thread is finishing and joining onto its parent thread, and \( e_{i+1} \) is an event in the parent thread. In any case, let \( e_j \) be the source of the \text{asw} edge, and \( t \) be the thread performing \( e_j \).

In the first case, there must be a \text{fork} instruction after \( e_j \) and before the next visible instruction in thread \( t \). The event \( e_j \) must also be the last event completed in thread \( t \), because otherwise, the source of the \text{asw} edge would be some other event than \( e_j \). To produce the trace \( \sigma_{i+1} \), we can switch to thread \( t \) and run the program until the \text{fork} is done, switch to the newly created thread, and run until \( e_{i+1} \) is done.

In the second case, the thread \( t \) is finishing, and there is no more visible instruction in thread \( t \). To produce the trace \( \sigma_{i+1} \), we can switch to thread \( t \) and run until \( t \) finishes, then switch to the
parent thread, and run until \( e_{i+1} \) is done. We must encounter a Join instruction before \( e_{i+1} \) is done, because otherwise the destination of the asw edge would be some event other than \( e_{i+1} \).

A.4. If \( e_{i+1} \) has an incoming \( sc \) edge from some event \( e_j \), then it is similar to the case of \( sb \). To obtain \( \sigma_{i+1} \), we will switch to the thread that performs \( e_{i+1} \) and continue until \( e_{i+1} \) is done. There shall be no \( seq_{-}cst \) events sequenced before \( e_{i+1} \) that has not yet been performed, because otherwise, there cannot be an \( sc \) edge from \( e_j \) to \( e_{i+1} \) in \( E_{i+1} \).

A.5. If \( e_{i+1} \) has an incoming \( rf \) edge from some event \( e_j \), then we need to show that it is valid for \( e_{i+1} \) to read from \( e_j \) in \( \sigma_{i+1} \). Let \( e_{i+1} \) be an RMW or atomic read at \( M \). We claim that \( e_j \) belongs to the set constructed by BuildMayReadFrom procedure when the operational model processes the instruction that creates \( e_{i+1} \). The for loop in the procedure considers the thread that performs \( e_j \). If event \( e_j \) does not happen before \( e_{i+1} \), then there cannot be any event \( e_k \) that modifies \( M \) and that \( e_j \) \( sb \rightarrow e_k \) \( hb \rightarrow e_{i+1} \), because in that case, Write-Read Coherence (CoWR) would forbid \( e_{i+1} \) from reading from \( e_j \) in \( E_{i+1} \). Therefore, \( e_j \) \( base \) at line 8. Now we will show that \( e_j \) is not removed at line 10 when \( e_{i+1} \) has \( seq_{-}cst \) memory ordering. If the last \( seq_{-}cst \) modification of \( M \) that precedes \( e_{i+1} \) in the total order of \( sc \), i.e., \( S \) at line 4, does not exist, then we are done. Assume such \( S \) exists. According to Section 29.3 statement 3 of the C++11 standard, \( e_{i+1} \) either reads from \( S \) or some non-\( seq_{-}cst \) modification of \( M \) that does not happen before \( S \). The fact that \( e_{i+1} \) reads from \( e_j \) in \( E_{i+1} \) implies that \( e_j \) is either \( S \) or a non-\( seq_{-}cst \) modification that does not happen before \( S \). Hence, \( e_j \) is not removed from \( base \) at line 10. If \( e_{i+1} \) is an RMW, then \( e_j \) has not been read by any other RMW, because no two RMWs can read from the same modification in \( E_{i+1} \). Hence, \( e_j \) is in the set returned from the BuildMayReadFrom procedure.

We still need to show that having \( e_{i+1} \) read from \( e_j \) does not create a cycle in the \( mo-graph \). The discussion in paragraph B.2 below about how an atomic load updates the \( mo-graph \) shows that having \( e_{i+1} \) read from \( e_j \), the updated \( mo-graph \) is still consistent with modification orders in \( E_{i+1} \). Thus, \( e_{i+1} \) reading from \( e_j \) does not create a cycle in \( mo-graph \). We defer the proof to paragraph B.2.

Now we will show that the \( mo-graph \) of \( \sigma_{i+1} \) can be extended to the modification orders in \( E_{i+1} \). The \( mo-graph \) is updated when the operational model processes an atomic store, load, or RMW. So we will assume that \( e_{i+1} \) corresponds to an atomic store, load, or RMW. Otherwise, the modification orders in \( E_{i+1} \) are the same as those in \( E_i \), and the \( mo \) edges in \( mo-graph \) are not updated, and hence \( mo-graph \) of \( \sigma_{i+1} \) can be extended to the modification orders in \( E_{i+1} \) by inductive hypothesis.

B.1. Suppose \( e_{i+1} \) is an atomic store that modifies atomic location \( M \). We consider two cases: \( e_{i+1} \) is the last element in \( S^M_{i+1} \) or \( e_{i+1} \) is not the last element \( S^M_{i+1} \). In the first case, \( e_{i+1} \) is the last element in the modification order \( S^M_{i+1} \) of \( E_{i+1} \). Let \( e_j \) be the second last element in \( S^M_{i+1} \). Since \( e_j \) precedes \( e_{i+1} \) in modification order, the modification ordering is either forced by coherence rules, consistent with \( sc \) relations, consistent with Section 29.3 statement 7 of C++11 standard, or not forced by any relations in \( E_{i+1} \). If the modification ordering between \( e_j \) and \( e_{i+1} \) is forced by coherence rules under \( hb \) and \( rf \) relations in \( E_{i+1} \), then the coherence rules being inferred can only be Coherence of Write-Write (CoWW) or Coherence of Read-Write (CoRW), because the other two coherence rules require \( rf \) relations that are not present in \( E_{i+1} \). For CoWW, line 12 in the \( WRITE\_PRIOR\_SET \) procedure considers the atomic store \( X \) corresponding to \( e_j \) and adds it to \( priorset \). We claim that the store \( X \) will not be filtered out by the last function call in line 13, because otherwise there would be an atomic store event \( e_k \) (different from \( e_{i+1} \)) sequenced after \( e_j \), contradicting the assumption that \( e_j \) is the second last element in \( S^M_{i+1} \). The case for CoRW is similar. If the modification ordering between \( e_j \) and \( e_{i+1} \) is consistent with \( sc \) relations, then \( e_j \) and \( e_{i+1} \) are both \( seq_{-}cst \) atomic stores and line 4 in the \( WRITE\_PRIOR\_SET \) procedure considers such case. If the modification ordering between \( e_j \) and \( e_{i+1} \) is consistent with Section 29.3 statement 7 of C++11 standard, then this case is dealt with at line 11 in the \( WRITE\_PRIOR\_SET \) procedure. If the modification ordering between \( e_j \) and \( e_{i+1} \) is not forced by any relations in \( E_{i+1} \), then the \( mo-graph \) does not contain a \( mo \) edge from \( e_j \) to \( e_{i+1} \), and we are free to extend \( mo-graph \) to include \( S^M_{i+1} \). In fact, the algorithm \( WRITE\_PRIOR\_SET \) may add \( mo \) edges from events modification ordered before \( e_j \) to \( e_{i+1} \). This is not a problem, because adding such edges does not introduce modification ordering not present in the modification order of \( E_{i+1} \).

In the second case, let \( e_j \) be the event immediately preceding \( e_{i+1} \) in \( S^M_{i+1} \) and \( e_k \) be the event immediately succeeding \( e_{i+1} \) in \( S^M_{i+1} \). Without loss of generality, assume \( e_k \) is the last event in \( S^M_{i+1} \). The modification ordering between \( e_j \) and \( e_{i+1} \) could be any one of the cases discussed in the first case in this paragraph. So we do not repeat the same argument. However, since all other events in \( E_{i+1} \) including \( e_j \) come before \( e_{i+1} \) in the topological order, there is no chain of \( rf \), \( sb \), \( asw \), and \( sc \) edges that come from \( e_k \) to any other event in \( E_{i+1} \). Thus, the only possibility is that no relations in \( E_{i+1} \) force the existence of the modification ordering between \( e_{i+1} \) and \( e_k \). Hence, the \( mo \) edge between \( e_{i+1} \) and \( e_k \) does not exist in the \( mo-graph \) of \( \sigma_{i+1} \), and we are free to extend \( mo-graph \) to include \( S^M_{i+1} \). If \( e_k \) is not the last event in \( S^M_{i+1} \), then the same argument applies to any event modification ordered after \( e_k \) in \( S^M_{i+1} \).

B.2. Suppose that \( e_{i+1} \) is an atomic load that reads from event \( e_j \) at atomic location \( M \). Adding event \( e_{i+1} \) does not change the modification order at \( M \) from \( E_i \) to \( E_{i+1} \), but performing the instruction corresponding to \( e_{i+1} \) may change the \( mo-graph \) from \( \sigma_i \) to \( \sigma_{i+1} \). We will show that the \( mo-graph \) in \( \sigma_{i+1} \) can still be extended to the modification orders in \( E_{i+1} \). Line 6, 7, and 8 in the \( READ\_PRIOR\_SET \) procedure consider statements 5, 4, and 6 in Section 29.3 of the standard. For each thread, if such \( S_1 \), \( S_2 \), and \( S_3 \) exist, the events corresponding to them are all modification ordered before \( e_j \) in \( E_{i+1} \). Therefore, having \( mo \) edges from \( S_1 \), \( S_2 \), and \( S_3 \) to \( e_j \) in \( mo-graph \) does not conflict with the modification orders in \( E_{i+1} \). Line 9 in the \( READ\_PRIOR\_SET \) procedure considers Write-Read Coherence (CoWR) or Read-Read Coherence (CoRR). If an \( mo \) edge from some event \( e_k \) to \( e_j \) in \( mo-graph \) is induced by CoWR and CoRR, then \( e_k \) must be modification ordered before \( e_j \) in \( E_{i+1} \) by the standard. Hence, when extending \( \sigma_i \) to \( \sigma_{i+1} \), the newly created \( mo \) edges in \( mo-graph \) do
not conflict with modification orders in $E_{i+1}$. Because the mo-graph in $\sigma_i$ can be extended to the modification orders of $E_i$ by inductive hypothesis, so can the mo-graph in $\sigma_i$ be extended to the modification orders of $E_{i+1}$.

B.3. Suppose that $\epsilon_{i+1}$ is an atomic RMW that reads from $\epsilon_j$. An RMW modifies the mo-graph in three phases: performing an atomic load, migrating mo edges using the AddRMWEdge procedure, and performing an atomic store. We also consider two cases.

In the first case, $\epsilon_{i+1}$ is the last element in $S_{i+1}$, of $E_{i+1}$. The first and third phases have been discussed in paragraphs B.1 and B.2. In the second phase, since $\epsilon_{i+1}$ is the last element in $S_{i+1}$, no edges in $\text{mo-graph}$ are migrated. Then an mo edge is added from $\epsilon_j$ to $\epsilon_{i+1}$ in $\text{mo-graph}$, which does not conflict with the modification order $S_{i+1}$, because an atomic RMW is immediately modification ordered after the modification it reads from.

In the second case, $\epsilon_{i+1}$ is not the last element in $S_{i+1}$. Since $\epsilon_{i+1}$ reads from $\epsilon_j$, $\epsilon_j$ must immediately precede $\epsilon_{i+1}$ in $S_{i+1}$. The first phase of the RMW is equivalent to an atomic load. In the second phase, any outgoing mo edges from $\epsilon_j$ will be migrated to outgoing mo edges from $\epsilon_{i+1}$, and an mo edge is added from $\epsilon_j$ to $\epsilon_{i+1}$ in $\text{mo-graph}$. The third phase is the same as the second case of an atomic store, except that for an event $\epsilon_k$ modification ordered after $\epsilon_{i+1}$ in $S_{i+1}$, there may exist mo edges from $\epsilon_{i+1}$ to $\epsilon_k$ in $\text{mo-graph}$ due to edge migrations in the second phase.

In both cases, the mo-graph of $\sigma_{i+1}$ does not contain mo edges that conflict with modification orders in $E_{i+1}$. Hence, the mo-graph of $\sigma_{i+1}$ can be extended to include modification orders in $E_{i+1}$.

Considering all above cases in paragraphs B.1, B.2, and B.3, the mo-graph of $\sigma_{i+1}$ can be extended to the modification orders in $E_{i+1}$, and the proof completes.

**Lemma 6.** Let $P$ be an arbitrary program, and $\sigma \in \text{traces}(P)$ be a trace. Then for all $E \in \text{lift}(\sigma)$, we have $E \in \text{rConsistent}(P)$.

**Proof.** In the backward direction, we want to show that given a program $P$ and a trace $\sigma$ produced by the operational model, then any execution $E \in \text{lift}(\sigma)$ obtained by lifting the trace $\sigma$ is an element of $\text{rConsistent}(P)$. We will prove by induction on the construction of the partial trace $\sigma_i$. Specially, if we have $E_k \in \text{lift}(\sigma_i)$, where $E_k$ is a partial execution graph of $E$ based on a topological sort of $\overline{\mathcal{E}}$, then when $\sigma_i$ is extended to $\sigma_{i+1}$, we have $E_k \in \text{lift}(\sigma_{i+1})$ or $E_{k+1} \in \text{lift}(\sigma_{i+1})$, where $E_{k+1}$ is also a partial execution graph of $E$ subject to the same topological sort.

Let $P$ be an arbitrary program and $\sigma \in \text{traces}(P)$ be a trace produced by our operational model. Let $E \in \text{lift}(\sigma)$ be an execution obtained by lifting the trace $\sigma$. In Section A.2, we have argued that when only considering $sb$, asw, $rf$ and sc edges, any execution obtained by lifting a trace is acyclic. In the process of lifting traces, some transitions create events while some do not. Label the events in $E$ in the order that they are created by transitions in $\sigma$ is a natural topological sort of $\overline{\mathcal{E}}$. All the partial execution graphs described below are based on this natural topological sort. We also have the mo-graph of all partial traces $\sigma_i$ be extended in a way that is consistent with $E$ in the lifting process. We will use $\text{lift}(\sigma_i)$ to refer to the specific execution in $\text{lift}(\sigma_i)$ whose mo-graph extension is consistent with that of $E$.

Base case: when $i = 0$, $\sigma_0$ is the empty trace, which is the initial state of the operation model for $P$. Thus, $\text{lift}(\sigma_0)$ is the empty execution graph $E_0$, which is a valid partial execution graph of $E$.

Inductive step: suppose we have constructed a partial trace $\sigma_i$ of $\sigma$ and that $\text{lift}(\sigma_i) = E_k$, where $E_k$ is a partial execution graph of $E$. We will show that when $\sigma_i$ is extended to $\sigma_{i+1}$ by executing the next transition $t_{i+1}$, we have either $\text{lift}(\sigma_{i+1}) = E_k$ or $\text{lift}(\sigma_{i+1}) = E_{k+1}$, where $E_{k+1}$ is a partial execution graph of $E$.

We will consider different cases for the transition $t_{i+1}$ below.

**Invisible Instruction.** If the transition $t_{i+1}$ is an invisible instruction, such as an $if$ statements, an assignment to non-atomic locations, and the empty statement $\varepsilon$, then it leaves $E_k$ unchanged, and $\text{lift}(\sigma_{i+1}) = E_k$. For $if$ statements, the partial trace at this point determines a branch to take, and proving that taking the branch will produce a valid partial execution graph when lifted comes down to proving the rest of cases analyzed.

**Visible Instruction.** If the next transition $t_{i+1}$ creates a new event $e_{k+1}$ in the lifting process, we have $\text{lift}(\sigma_{i+1}) = E_{k+1}$. Moreover, new $sb$, asw, sc edges, and updates in the modification orders may also be added to $E_k$ to form $E_{k+1}$. We already show that $E_{k+1}$ has acyclic $sb \cup asw \cup rf \cup sc$ edges. So we only need to show that $E_{k+1}$ is a partial execution graph of $E$.

We will discuss the newly added $sb$ and asw edges first. Suppose the transition $t_{i+1}$ is a general visible instruction that corresponds to the event $e_{k+1}$. We will consider new $sb$ and asw edges that may be added to $E_k$ when lifting $\sigma_{i+1}$. Suppose that $t_{i+1}$ is performed by thread $t$ and is not the first visible instruction in thread $t$. Then an $sb$ edge will be drawn from the last visible event performed by $t$ to $e_{k+1}$. Suppose that $t_{i+1}$ is the first visible instruction performed by thread $t$. If $t$ is the main thread, then no new edges to $e_{k+1}$ will be added during lifting. If $t$ is not the main thread, then the parent thread $t^*$ that created $t$ must have performed a Fork instruction, and an asw edge from the last visible instruction sequenced before the Fork instruction to $e_{k+1}$ will be added to $E_k$, to obtain $E_{k+1}$. It is clear that the way we construct $sb$ and asw edges in lifting traces is consistent with the C++ axiomatic model.

**Visible Instruction (Atomic Store).** If the next transition $t_{i+1}$ is an atomic store statement at location $M$, it will create an $\text{StoreElem}$ that corresponds to the event $e_{k+1}$. We will focus on the changes to modification orders and sc relations. If $e_{k+1}$ is a seq_cst store, lifting $\sigma_{i+1}$ will cause an sc edge to be added from the last seq_cst event to $e_{k+1}$. Line 4 in the $\text{WritePriorSet}$ procedure ensures that mo edges in the mo-graph conform with sc relations by adding an mo edge from the last seq_cst store at $M$ to $e_{k+1}$ in the mo-graph if the last seq_cst store at $M$ exists. So modification orders confirm with sc relations in $E_{k+1}$. Since the operation model may only add incoming mo edges to $e_{k+1}$ in the mo-graph when processing $t_{i+1}$, then modification orders in $E_{k+1}$ do not have cycles. By Lemma 4, $sb \cup asw \cup rf \cup sc$ in $E$, it follows that sc relations conform with $hb$ relations in $E_{k+1} = \text{lift}(\sigma_{i+1})$. Line 11 and line 12 in the $\text{WritePriorSet}$ procedure ensure that mo edges induced by seq_cst fences, CoRW and CoWW are added to the mo-graph. Therefore, the new modification orders in $E_{k+1}$ induced by the changes in the mo-graph in the lifting process conform with CoRW, CoWW, Section 29.3
statement 7 of the C++11 standard, and sc relations. Conformity with CoRW and CoWW implies that mo conforms with hb.

Now, if \( e_{k+1} \) is the last element in \( S_{k+1} \), then the above discussion shows that \( E_{k+1} \) is a valid partial execution graph of \( E \) based on the natural topological sort. It is also possible that \( e_{k+1} \) is not the last element in \( S_{k+1} \). Let \( e_j \) be any event modification ordered after \( e_{k+1} \) in \( S_{k+1} \). Note that \( e_j \) is topologically ordered before \( e_{k+1} \). Since the WRITE_PRIORITY procedure only adds incoming mo edges to \( e_{k+1} \), no mo or chain of mo edges from \( e_{k+1} \) to \( e_j \) exists in mo-graph.

Since the operational model forbids cycles in mo-graph, the final mo-graph of \( \sigma \) is free of cycles, and the modification orders in the final mo-graph at each location are extended to a total order in \( E \) during lifting. Because we assume that mo-graph of all partial traces \( \sigma_i \) is extended in a way that is consistent with \( E \) in the lifting process, we can conclude that the modification ordering between \( e_{k+1} \) and \( e_j \) do not cause any cycles in modification orders of \( E_{k+1} \), and is only added to make \( S_{k+1} \) a total order. Therefore, \( E_{k+1} \) is a valid partial execution graph of \( E \).

Visible Instruction (Atomic Load). If the transition \( t_{i+1} \) is an atomic load at location \( M \), it creates an LoadElem that corresponds to the event \( e_{k+1} \). To obtain \( E_{k+1} \), \( \text{lift}(\sigma_{i+1}) \), a new rf edge is added to \( E_k \), and the modification orders at \( M \) may be updated. Suppose that \( e_{k+1} \) reads from \( e_j \), where \( e_j \) is topologically ordered before \( e_{k+1} \). We make the following claim:

**Claim 1** Any valid store that the operational model allows the LoadElem corresponding to \( e_{k+1} \) to read from is also valid for \( e_{k+1} \) to read from under our axiomatic model.

We will prove this claim by contradiction or contrapositive using case analysis. We will assume our axiomatic model forbids \( e_{k+1} \) from reading from \( e_j \).

**Case 1:** If having \( e_{k+1} \) read from \( e_j \) violates CoWR, then there exists an event \( e_{k+1} \) in \( E_{k+1} \) such that \( e_j \rightarrow e_{k+1} \) and \( e_j \rightarrow e_{k+1} \) in mo-graph. We will first assume that \( e_j \rightarrow e_{k+1} \) in the mo-graph of \( \sigma_i \). The WRITE_PRIORITY procedure iterates over each thread, and when considering the thread that performs \( e_j \), line 9 finds either the store \( e_j \rightarrow M \) or a store sequenced after \( e_j \), or a load sequenced after \( e_j \). Then the store \( A \) in line 10 of WRITE_PRIORITY is the store \( e_j \rightarrow M \), a store sequenced after \( e_j \), or a store read by a load sequenced after \( e_j \). In any case, based on CoWW, CoWR and the inductive hypothesis that \( E_k \) is a valid partial execution graph of \( E \), we can deduce that the mo edge (or the equivalent chain of mo edges) \( e_j \rightarrow M \) exists in the mo-graph of \( \sigma_i \). Since \( A \) is reachable from \( e_j \), line 16 in the WRITE_PRIORITY procedure forbids the LoadElem corresponding to \( e_{k+1} \) from reading from the store corresponding to \( e_j \) in the operational model. Then we prove the Claim 1 by contrapositive.

However, it is also possible that \( e_j \) and \( e_{k+1} \) are unordered in the mo-graph of \( \sigma_i \). Then the modification ordering between \( e_j \) and \( e_{k+1} \) in \( E_{k+1} \) is due to the extension of the final mo-graph of \( \sigma \) in \( E \), as the mo-graph of all partial traces \( \sigma_i \) are extended in a way that is consistent with \( E \). Having \( e_{k+1} \) read from \( e_j \) will add mo edges so that \( e_j \rightarrow M \) \( e_{k+1} \) exists in the mo-graph of \( \sigma_i \) and the final mo-graph of \( \sigma \). This is a contradiction because the extension of the final mo-graph is only required between two unordered stores in mo-graph. Therefore, we prove the Claim 1 by contradiction.
edges. Because we assume conformity of modification orders with
sc edges in $E_k$, if any cycle exists in union of sc and $S_{k+1}^M$ in $E_{k+1}$, the
cycle must involve one of the newly added modification ordering.
Suppose a cycle $C$ exists and it contains events $X$ and $e_j$ where
$X \rightarrow e_j$ is a newly added modification ordering in $E_{k+1}$. Since all
events in $S_{k+1}^M$ are atomic stores, the two endpoints of any maximal
chain of sc edges (could also be an sc edge) in $C$ must be seq_cst
atomic stores. Let the two endpoints be events $Z_1$ and $Z_2$. Then,
the relation $Z_1 \xrightarrow{mo} Z_2$ or $Z_2 \xrightarrow{mo} Z_1$ whichever conforms with the
sc edges must exist in the mo-graph of $\sigma_i$. No $S_{k+1}^M$ edges in $C$ can be
due to the extensions of unordered stores in mo-graph of $\sigma_{i+1}$,
because we assume that the mo-graph of $\sigma$ is extended without
cycles. Therefore, we must have $e_j \xrightarrow{mo} X$ in the mo-graph of $\sigma_i$.
Then, the relation $X \xrightarrow{mo} e_j$ cannot exist in the mo-graph of $\sigma_{i+1}$, as
it makes the mo-graph of $\sigma_{i+1}$ acyclic. However, $X \xrightarrow{S_{k+1}^M} e_j$ cannot
be due to the extension of unordered stores in the mo-graph of $\sigma_{i+1}$.
Therefore, we have a contradiction, and sc conforms with
modification orders in $E_{k+1}$. Therefore, we have $E_{k+1}$ is a valid
partial execution graph of $E$.

Visible Instruction (Atomic RMW). If the transition $t_{i+1}$ is an
atomic RMW statement at location $M$, it creates an RMWElem that
corresponds to the event $e_{k+1}$. An atomic RMW is both a load and a
store except that the standard requires that RMW operations shall
always read the last value (in the modification order) written before
the write associated with the RMW operation. In the operational
model, the BuildMayReadFrom procedure forbids two RMWs to
read from the same store. Denote the store that $e_{k+1}$ reads from
as $e_j$. Then the AddRMWEdge procedure adds all outgoing mo
dges from $e_j$ to the set of outgoing edges of $e_{k+1}$ and adds an mo
dge from $e_j$ to $e_{k+1}$ to form the mo-graph of $\sigma_{i+1}$. Therefore, when
lifting $\sigma_{i+1}$, $e_j$ is immediately modification ordered before $e_{k+1}$ in
$E_{k+1}$. Hence, $E_{k+1}$ is a valid partial execution graph of $E$.

If the transition $t_{i+1}$ is a fence instruction, it creates a FenceElem
which corresponds to the event $e_{k+1}$. Lifting $\sigma_{i+1}$ adds $e_{k+1}$ to $E_k$
but does not create rf edges or change modification ordering. If $e_{k+1}$
has seq_cst ordering, a sc edge from the last seq_cst event (if exists)
to $e_{k+1}$ will be created, and sc conforms with hb and modification
orders in $E_{k+1}$. Thus, $E_{k+1}$ is a valid partial execution graph of $E$.

We have completed the proof of induction by case analysis. □

B ADDITIONAL DATA
Table 4 reports some detailed statistics about the 25 JavaScript
benchmarks in the JSBench suite.
Table 4: Performance results (in ms) for individual JavaScript benchmarks in the JSBench suite for tsan11, tsan11rec, and C11Tester under two configurations. Smaller times are better. The "Memory Accesses" columns report the number of atomic operations (including synchronization operations such as mutex and condition variable operations) and normal accesses to shared memory locations executed by individual JavaScript benchmarks under C11Tester.

| JavaScript benchmark     | tsan11 | tsan11rec | C11Tester | # non-atomics | # atoms |
|-------------------------|--------|-----------|-----------|---------------|---------|
|                         | 1 core | all cores | 1 core    | all cores     |         |
| amazon/chrome           | 368    | 348       | 1054      | 383           | 767     | 766    | 92M     | 99M     |
| amazon/chrome-win       | 368    | 349       | 1053      | 385           | 767     | 767    | 90M     | 99M     |
| amazon/firefox          | 362    | 341       | 1085      | 368           | 718     | 718    | 71M     | 73M     |
| amazon/firefox-win      | 351    | 331       | 1038      | 357           | 700     | 701    | 67M     | 71M     |
| amazon/safari           | 400    | 372       | 1170      | 414           | 778     | 779    | 77M     | 86M     |
| facebook/chrome         | 2323   | 2017      | 7693      | 2669          | 4421    | 4434   | 471M    | 449M    |
| facebook/chrome-win     | 3715   | 3135      | 11463     | 3772          | 7084    | 7092   | 607M    | 850M    |
| facebook/firefox        | 1458   | 1303      | 5219      | 1835          | 3002    | 3015   | 278M    | 318M    |
| facebook/firefox-win    | 818    | 721       | 2858      | 944           | 1619    | 1628   | 149M    | 189M    |
| facebook/safari         | 3639   | 3034      | 14009     | 4508          | 7360    | 7375   | 577M    | 1,036M  |
| google/chrome           | 1616   | 1488      | 5619      | 1901          | 3358    | 3365   | 333M    | 399M    |
| google/chrome-win       | 1579   | 1447      | 5489      | 1866          | 3260    | 3249   | 400M    | 359M    |
| google/firefox          | 962    | 899       | 2544      | 1069          | 1984    | 1986   | 171M    | 173M    |
| google/firefox-win      | 1123   | 1033      | 3383      | 1265          | 2279    | 2283   | 214M    | 208M    |
| google/safari           | 1445   | 1320      | 4829      | 1670          | 2943    | 2941   | 273M    | 319M    |
| twitter/chrome          | 618    | 588       | 1078      | 645           | 1305    | 1306   | 154M    | 141M    |
| twitter/chrome-win      | 620    | 587       | 1073      | 644           | 1310    | 1310   | 158M    | 151M    |
| twitter/firefox         | 175    | 165       | 275       | 178           | 376     | 376    | 50M     | 47M     |
| twitter/firefox-win     | 174    | 164       | 277       | 177           | 373     | 374    | 50M     | 47M     |
| twitter/safari          | 466    | 443       | 880       | 490           | 956     | 963    | 103M    | 106M    |
| yahoo/chrome            | 1638   | 1358      | 6191      | 2000          | 4294    | 4308   | 339M    | 663M    |
| yahoo/chrome-win        | 1345   | 1115      | 4953      | 1633          | 3172    | 3176   | 234M    | 496M    |
| yahoo/firefox           | 1638   | 1363      | 6250      | 2016          | 4292    | 4283   | 338M    | 663M    |
| yahoo/firefox-win       | 883    | 747       | 3460      | 1100          | 1895    | 1888   | 113M    | 320M    |
| yahoo/safari            | 1635   | 1370      | 6263      | 2019          | 4292    | 4291   | 338M    | 661M    |