

A Practical Approach for Model Checking C/C++11 Code

BRIAN NORRIS and BRIAN DEMSKY, University of California, Irvine

Writing low-level concurrent software has traditionally required intimate knowledge of the entire toolchain and often has involved coding in assembly. New language standards have extended C and C++ with support for low-level atomic operations and a weak memory model, enabling developers to write portable and efficient multithreaded code.

In this article, we present CDS_{CHECKER}, a tool for exhaustively exploring the behaviors of concurrent code under the C/C++ memory model. We have used CDS_{CHECKER} to exhaustively unit test concurrent data structure implementations and have discovered errors in a published implementation of a work-stealing queue and a single producer, single consumer queue.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; **Software testing and debugging**

Additional Key Words and Phrases: Relaxed memory model, model checking

ACM Reference Format:

Brian Norris and Brian Demsky. 2016. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.* 38, 3, Article 10 (May 2016), 51 pages.

DOI: <http://dx.doi.org/10.1145/2806886>

1. INTRODUCTION

With the wide-scale deployment of multicore processors, software developers must write parallel software to realize the benefits of continued improvements in microprocessors. Many developers in industry have adopted a parallel programming model that uses threads to parallelize computation and concurrent data structures to coordinate and share data between threads.

Careful data structure design can improve scalability by supporting multiple simultaneous operations and by reducing the time taken by each individual data structure operation. Researchers and practitioners have developed a wide range of concurrent data structures designed with these goals in mind [Shavit 2011; Moir and Shavit 2004; Michael and Scott 1996]. Such data structures often use fine-grained conflict detection and avoid contention.

Concurrent data structures often use a number of sophisticated techniques, including the careful use of low-level atomic instructions (e.g., compare and swap (CAS), atomic increment, etc.), careful orderings of loads and stores, and fine-grained locking. For example, while the standard Java hash table implementation can limit program scalability to a handful of processor cores, the carefully designed concurrent hash tables can scale to many hundreds of cores [Click 2007]. Traditionally, developers had to

This project was partly supported by a Google Research Award and by the National Science Foundation under Grants No. CCF-0846195, No. CCF-1217854, No. CNS-1228995, and No. CCF-1319786.

Authors' addresses: B. Norris, 1600 Amphitheatre Pkwy., Mountain View, CA 94043; email: briannorris@google.com; B. Demsky, Department of Electrical Engineering and Computer Science, University of California, Irvine 92697; email: bdemsky@uci.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 0164-0925/2016/05-ART10 \$15.00

DOI: <http://dx.doi.org/10.1145/2806886>

target their implementation of such data structures to a specific platform and compiler, using intimate knowledge of the platform details and even coding some data structure components in assembly.

1.1. C/C++ Memory Model

Recently, standardization committees extended the C and C++ language standards with support for low-level atomic operations [ISO/IEC 14882:2011; ISO/IEC 9899:2011; Boehm and Adve 2008] which allow experts to craft efficient concurrent data structures that avoid the overheads of locks. The accompanying memory model provides for memory operations with weaker semantics than sequential consistency; however, using these weak atomic operations is extremely challenging, as developers must carefully reason about often subtle memory model semantics to ensure correctness. Even experts often make subtle errors when reasoning about such memory models.

The potential performance gains of low-level atomics may lure both expert and novice developers to use them. In fact, some common parallel constructs (e.g., sequential locks) require ordinary developers to use atomics in C/C++ [Boehm 2012]. In the absence of proper tool support, developers will likely write concurrent code that they hope is correct and then rely on testing to find bugs. Adequately testing concurrent code that uses C/C++ atomics is nearly impossible. Even just exploring the behaviors of a given binary on a given architecture can be tricky as some bugs require precise timing to trigger. Moreover, neither existing processors nor compilers make full use of the freedom provided by the C/C++ memory model. As future compiler updates implement more aggressive optimizations, compilers will leverage the freedom provided by the memory model and produce binaries that exhibit new (but legal) behaviors that will likely expose existing bugs.

1.2. Tool Support

While it is possible to use a formal specification of the C/C++ memory model [Batty et al. 2011] to prove code correct, experience suggests that most software developers are unlikely to do so (e.g., because they lack expertise or time). There is a pressing need, then, for tools that allow developers to unit test portions of their code to discover what behaviors the memory model allows. Such tools could guarantee soundness of properly abstracted code via exhaustive exploration. Typically, concurrent data structures are amenable to such a scenario; developers reason about (and rigorously test) their implementation in isolation from the details of a larger client program and then provide that abstraction to users, who only must ensure correct use of the abstraction.

We present a new approach for exhaustively exploring the behaviors of code under the C/C++ memory model, based on stateless model-checking [Godefroid 1997]. Stateless model-checkers typically explore a program's possible behaviors—or *state space*—by repeatedly executing the program under different thread interleavings. However, exhaustive search of potential thread interleavings becomes computationally intractable as programs grow to any reasonable length.

Thus, state-of-the-art stateless model-checking rests on a class of optimization techniques known as dynamic partial-order reduction (DPOR) [Flanagan and Godefroid 2005]. The DPOR algorithm can *reduce* the explored state space by exploring only those executions whose visible behavior may differ from the behavior of previously explored executions. During its state-space exploration, DPOR identifies points at which it must explore program operations in more than one interleaving (e.g., two concurrent stores to the same object conflict, whereas two loads do not). Conflict points are recorded in a backtracking set, so the exploration can return (or *backtrack*) to the recorded program point during a future execution and attempt a different thread interleaving.

DPOR targets a sequentially consistent model, preventing its direct application to the C/C++ memory model, as C and C++ provide no guarantee of a total execution order

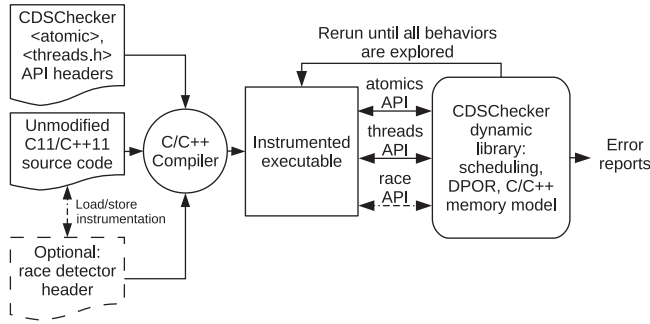


Fig. 1. CDSChecker system overview.

in which loads see the value written by the most recent store. The C/C++ memory model instead defines the relation between loads and the values they see in terms of a *reads-from* relation which is subject to a number of constraints. We present a new approach that exhaustively explores the set of legal reads-from relations, with thread scheduling and support for sequentially consistent atomics influenced by DPOR.

In C/C++, shared variables must be either clearly annotated using the new `<atomic>` library (or higher-level thread support libraries, such as `<mutex>`) or else protected from conflicting concurrent access through use of these atomics or other synchronization primitives; any pair of conflicting accesses to nonatomic variables without proper synchronization constitutes a data race, which yields undefined behavior [ISO/IEC 9899:2011]. Thus, we simply designed CDSChecker as a dynamic library implementation of these threading and atomic libraries, as shown in Figure 1, and generally left other operations uninstrumented. Such a design can readily support a broad range of real-world applications, as users simply compile their code against our library with their compiler of choice. At runtime, CDSChecker schedules program fragments sequentially and determines the values returned by atomic memory operations.

To model all program behaviors, CDSChecker implements a backtracking-based system which performs repeated, controlled program execution until it has explored all necessary program behaviors. CDSChecker reports diagnostic information for all data races, deadlocks, uninitialized atomic loads, and user-provided assertion failures that occur for the provided input. All failure reports include a full trace of all thread and atomic operations performed in the program, a short description of the detected bug(s), and a representation of the reads-from relation for the execution.

Some tools already exist for testing program behavior according to the C/C++ memory model. CPPMEM [Batty et al. 2011] enumerates all potential modification orders and reads-from relations in programs (under a limited subset of C/C++ language constructs) and then eliminates the infeasible ones according to the formal specification of the memory model. CPPMEM lacks support for fences and only supports loops with a priori loop iteration bounds. We contribute an exponentially more efficient approach that makes it possible to check real code. Our approach reduces the search space by avoiding explicitly enumerating orderings that produce equivalent execution behavior. We also contribute support for fences and loops without fixed iteration bounds. Relacy [Vyukov Oct] explores possible behaviors of real C++ programs using library-based instrumentation, but it cannot model all types of relaxed behavior allowed by C/C++. Our approach fully models the relaxed behavior of real C and C++ code.

1.3. Limitations

Generally, CDSChecker will explore every distinct execution behavior allowed by the C/C++ memory model, providing exhaustive test coverage under a particular program

input. However, there are a few considerations in the design and implementation of CDSCHECKER that leave room for incompleteness. We summarize the limitations here and provide more thorough explanation in the body of the article.

- Supporting `memory_order_consume` requires a compiler’s deep knowledge of data dependencies. We opted instead to make CDSCHECKER compiler agnostic.
- Unbounded loops present infinite state spaces, which cannot be completely explored by a stateless model-checker. We explore such loops under the restriction of a fair schedule: either through bounded fairness enforced by our scheduler (bounds adjustable) or through the use of CHES [Musuvathi et al. 2008] yield-based fairness.
- Some programs rely on a live memory system in order to terminate. For such programs, we impose bounded liveness via an adjustable runtime option.
- CDSCHECKER may not explore all behaviors involving satisfaction cycles. Not only are satisfaction cycles difficult to generate in a model-checker, but also they are a thorny, unsolved issue in the current C and C++ specifications, which do not make it clear exactly which behaviors should be allowed and disallowed.
- CDSCHECKER uses a system of promises to allow loads to read from stores that appear later in the execution (Section 6). However, we do not allow these promises to remain forever in an execution which will never satisfy them. Thus, promise expiration theoretically may be a source of incompleteness.

1.4. Contributions

This article makes the following contributions:

- Basic Approach:** It presents new techniques that enable the stateless model-checking of C/C++ code under the C/C++ memory model. Our approach is the first that can model-check unit tests for real-world C/C++ data structure implementations under the C/C++ memory model.
- Constraints-Based Modification Order:** It introduces the first technique for model-checking the C/C++ memory model without explicitly enumerating the modification order of atomic objects, exponentially decreasing the search space.
- Relaxed Memory Model Support:** It develops new techniques to support the full variability of the memory model, including allowing loads to observe the values written by stores that appear later in the execution order while at the same time maintaining compatibility with uninstrumented code in libraries.
- Partial Order Reduction:** It combines our new relaxed model-checking techniques with existing schedule-driven partial order reduction to efficiently support sequentially consistent memory actions.
- Bug Finding:** It shows that our techniques can find bugs in real world code including finding a new bug in a published, peer-reviewed implementation of the Chase-Lev deque.
- Correctness:** It defines a set of constraints on the C/C++ memory model that suffice to prohibit “out-thin-air” behaviors. These constraints are not intended to solve the challenging problem of balancing the concerns of performance and semantics for out-of-thin-air values but rather to provide a memory model that we can show soundness relative to. It proves that the CDSCHECKER algorithm is sound for a version of the C/C++ memory model that satisfies these constraints.
- Evaluation:** It presents an evaluation of the model-checker implementation on several concurrent data structures. With runtimes averaging only a few seconds and no test taking over 11s, empirical results show that our tool is efficient in practice.

The remainder of the article is organized as follows. Section 2 presents an example. Section 3 reviews important aspects the C/C++ memory model. Section 4 gives

```

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

```

Fig. 2. C++11 code example.

an overview of our approach. Section 5 presents our constraint-based approach to modification orders. Section 6 provides more insight on how we support the relaxed memory model. Section 7 discusses release sequence support. Section 10 discusses how we handle fairness and memory liveness. Section 11 presents a soundness proof for the core CDSCHECKER algorithm for a simplified version of the C/C++ memory model. Section 12 evaluates CDSCHECKER. Section 13 presents related work. We conclude in Section 14.

2. 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++11 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 interthread ordering and visibility properties (discussed more in Section 3). 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 $\{r1 = r2 = 0\}$, $\{r1 = 0 \wedge r2 = 1\}$, or $\{r1 = 1 \wedge r2 = 0\}$, but it is somewhat counterintuitive that we may also see $\{r1 = r2 = 1\}$, in which both load statements read from the store statements that appear after the other load. 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, e.g., by compiler reordering).

Now consider a modification of the same example, where the load and store on variable `y` (Line 4 and Line 10) now use `memory_order_acquire` and `memory_order_release`, respectively, so when the load-acquire reads from the store-release, they form a release/acquire synchronization pair. Then in any execution where $r1 = 1$ and thus the load-acquire statement (Line 4) reads from the store-release statement (Line 10), the synchronization between the store-release and the load-acquire forms an ordering between `threadB()` and `threadA()`—particularly, that the actions in `threadA()` after the acquire must observe the effects of the actions in `threadB()` before the release. In the terminology of the C/C++ memory model, we say that all actions in `threadB()` sequenced before the release happen before all actions in `threadA()` sequenced after the acquire.

So when $r1 = 1$, `threadB()` must see $r2 = 0$. In summary, this modified example allows only three of the four previously described behaviors: $\{r1 = r2 = 0\}$, $\{r1 = 0 \wedge r2 = 1\}$, or $\{r1 = 1 \wedge r2 = 0\}$.

3. C/C++ MEMORY MODEL

The C/C++ memory model describes a series of atomic operations and the corresponding allowed behaviors of programs that utilize them. Note that throughout this article, we primarily discuss atomic memory operations that perform either a write (referred to as a *store* or *modification* operation) or a read (referred to as a *load* operation). The discussion generalizes to operations that perform both a read and a write (*read-modify-write*, or *RMW*, operations). Section 8 describes how CDSCHECKER supports fences.

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.

relaxed:: `memory_order_relaxed` – weakest memory ordering
release:: `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst` – a store-release may form release/consume or release/acquire synchronization
consume::¹ `memory_order_consume` – a load-consume may form release/consume synchronization
acquire:: `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst` – a load-acquire may form release/acquire synchronization
seq-cst:: `memory_order_seq_cst` – strongest memory ordering

To ease programming burden, atomic operations default to using `memory_order_seq_cst` when no ordering is specified.

3.1. Relations

The C/C++ memory model expresses program behavior in the form of binary relations or orderings. The following subsections will briefly summarize the relevant relations. Some of this discussion resembles the preferred model from the formalization in Batty et al. [2011], adapted to suit its usage in CDSCHECKER.

Sequenced-Before. The order of program operations within a single thread of execution establishes an intrathread *sequenced-before* (*sb*) relation. Program operations for purpose of the memory model are loads, stores, RMWs, fences, and synchronization operations. Note that while some operations in C/C++ provide no intrathread ordering—the equality operator (`==`), for example—we ignore this detail and assume that *sb* totally orders all operations in a thread. This is necessary because the information of whether two atomic operations from the same thread are ordered by *sb* is not available to CDSCHECKER since it is merely linked against the program.

Reads-From. The *reads-from* (*rf*) relation consists of store-load pairs (X, Y) such that Y reads its value from the effect of X —or $X \xrightarrow{rf} Y$. In the C/C++ memory model, this relation is nontrivial, as a given load operation may read from one of many potential stores in the program execution.

Synchronizes-With. The *synchronizes-with* (*sw*) relation captures synchronization that occurs when certain atomic operations interact across two threads. For instance, release/acquire synchronization occurs between a pair of atomic operations on the same object: a store-release X and a load-acquire Y . If Y reads from X , then X synchronizes with Y —or $X \xrightarrow{sw} Y$. Synchronization also occurs between consecutive unlock and lock operations on the same mutex, between thread creation and the first event in the new thread, and between the last action of a thread and the completion of a thread-join operation targeting that thread.

Note that our discussion of *sw* is incomplete here. We will complete it when we introduce release sequences in Section 7.

¹We don't support consume due to implementation obstacles in detecting data dependencies. See Section 4.5.

Happens-Before. In CDSCHECKER, we avoid consume operations, and so the *happens-before* (*hb*) relation is simply the transitive closure of *sb* and *sw*.

Sequential Consistency. All seq-cst operations in a program execution form a total ordering (*sc*) so, for instance, a seq-cst load may not read from a seq-cst store prior to the most recent store (to the same location) in the *sc* ordering, nor from any store that happens before that store. The *sc* order must be consistent with *hb*.

Modification Order. Each atomic object in a program execution has an associated *modification order* (*mo*)—a total order of all stores to that object—which informally represents a memory-coherent ordering in which those stores may be observed by the rest of the program. Note that, in general, the modification orders for all objects cannot be combined to form a total ordering that is consistent with the reads-from relation. For instance, the surprising behavior in which $r1 = r2 = 1$ in the example in Figure 2 shows an instance where the union of *sb* and *rf* is cyclic, and we can easily extend the example to demonstrate a cyclic union of *sb* and *mo*.

4. CDSCHECKER OVERVIEW

CDSCHECKER’s model-checking algorithm (presented in Section 4.1) builds on partial order reduction concepts from Flanagan and Godefroid [2005]. However, the C/C++ memory model is significantly more complex than DPOR’s sequentially consistent model, and thus simply controlling thread interleavings does not suffice to reproduce the allowed behaviors. Thus it was necessary to develop a new approach to explore the richer set of behaviors allowed by the C/C++ memory model and new partial order reduction techniques to minimize the exploration of redundant executions.

One significant departure from DPOR is that the C/C++ memory model splits memory locations and operations into two categories: (1) normal locations and operations and (2) atomic locations and operations. The memory model forbids data races on normal memory operations (and assigns undefined semantics to programs with such races) but allows arbitrary interleavings of atomic operations. This enables CDSCHECKER to make a significant optimization over existing model-checkers—it *detects* and *reports* data races (a simple feat) on all instrumented normal memory accesses while exhaustively exploring interleavings (an expensive, combinatorial search) *only* for atomic memory operations. If a normal memory access can exhibit more than one behavior under the synchronization pattern established by the atomic operations in a given execution, then it has a data race and is forbidden by the C/C++ specifications.

CDSCHECKER’s design leverages this optimization; it exhaustively enumerates the behaviors of atomic memory accesses and simply checks for data races between normal memory operations, reporting any data races to the user. This cheapens the instrumentation required for normal memory operations and reduces the search space explored for racy (i.e., buggy) programs.

4.1. CDSCHECKER Architecture

We next discuss the high-level architecture of CDSCHECKER, beginning with our algorithm (Figure 3) and its relation to existing literature. In our discussions, we adapt several terms and symbols from Flanagan and Godefroid [2005]. We associate every state transition t taken by processes (i.e., threads) p with the dynamic operation that effected the transition, then define the *execution order*² S of these operations as the total ordering given by the sequence of scheduling choices made in Figure 3, Line 8.

²We use the term execution order instead of transition sequence to make clear the fact that a transition in our model-checker cannot be easily characterized as simply a function of the current state. For example, a load transition can depend on future stores.

```

1: Initially: EXPLORE( $\emptyset$ )

2: function EXPLORE( $S$ )
3:    $s \leftarrow \text{last}(S)$ 
4:   PROCESSACTION( $S$ )
5:   if  $\exists p_0 \in \text{enabled}(s)$  then
6:      $\text{threads}(s) \leftarrow \{p_0\}$ 
7:      $\text{threadsdone} \leftarrow \emptyset$ 
8:     while  $\exists p \in \text{threads}(s) \setminus \text{threadsdone}$  do
9:        $t \leftarrow \text{next}(s, p)$ 
10:       $\text{behaviors}(t) \leftarrow \{\text{Initial behaviors}\}$ 
11:       $\text{behavedone} \leftarrow \emptyset$ 
12:      while  $\exists b \in \text{behaviors}(t) \setminus \text{behavedone}$  do
13:        EXPLORE( $S.(t, b)$ )
14:         $\text{behavedone} \leftarrow \text{behavedone} \cup \{b\}$ 
15:      end while
16:       $\text{threadsdone} \leftarrow \text{threadsdone} \cup \{p\}$ 
17:    end while
18:  end if
19: end function

1: function PROCESSACTION( $S$ )
2:    $(t, b) \leftarrow \text{last}(S)$ 
3:   WAKEUP_SLEEPING_ACTIONS
4:   if  $t$  is a read then
5:     Perform appropriate behavior  $b$  for read (read from past store, read from
     existing promise, read from future value)
6:     Update mo graph with new edges
7:     Backtrack if mo graph is cyclic
8:   end if
9:   if  $t$  is a write then
10:    Resolve promises specified by behavior  $b$ 
11:    Update mo graph with new edges
12:    Backtrack if mo graph is cyclic
13:  end if
14:  if  $t$  is a fence then
15:    Process Fence
16:  end if
17:  if  $t$  is a mutex then
18:    Process Mutex
19:  end if
20:  if  $t$  is a release sequence fixup then
21:    Break or complete pending release sequence at end of execution as specified
    by behavior  $b$ 
22:  end if
23:  repeat
24:    Check for newly formed released sequences
25:    Update mo edges in response to new synchronization
26:    Backtrack if mo graph is cyclic
27:  until No new release sequences are formed
28:  Set backtracking point for  $t$  by updating  $\text{threads}$ .
29: end function

```

Fig. 3. CDSCHECKER algorithm.


```

1  atomic<int> x(0);
2
3  void threadA() {
4      x.store(1, memory_order_relaxed);
5      x.store(2, memory_order_relaxed);
6  }
7  void threadB() {
8      int r1 = x.load(memory_order_relaxed);
9      int r2 = x.load(memory_order_relaxed);
10 }

```

Fig. 4. C++11 code example.

We say that $next(s, p)$ is the next transition in thread p at a given state s ; $last(S)$ is the most recent state visited in S ; $S.t$ denotes extending S with an additional transition t ; and $enabled(s)$ is the set of all threads enabled in state s (threads can be disabled, e.g., when waiting on a held mutex or when completed).

We base the CDSHECKER algorithm on standard backtracking algorithms; we perform a depth-first exploration of the program's state space (recursive calls to EXPLORE, Line 13) by iterating over a set of threads whose next transition must be explored from the given state s (the outer loop, excluding Lines 10 to 15). Most of our algorithmic extensions correspond to the inner loop, which performs a similar (but distinct) backtracking to explore the different possible behaviors of the transition t that was selected in the outer loop. Section 4.3 further describes the purpose of the *behaviors* set.

Note that as presented in Figure 3, the outer loop will only select a single initial execution order (i.e., each state's *threads* set only ever contains the initial thread selection p_0). The PROCESSACTION procedure examines the last transition and may add additional backtracking points for previous states as needed to exhaustively explore the state space. For clarity of presentation, we describe PROCESSACTION's behavior in prose throughout the rest of the article.

4.2. Example

Figure 4 presents a simple example to show the operation of CDSHECKER. Our examples contains one variable, x , and two threads. The variable x is initialized to 0, the first thread stores 1 and then 2 to the variable x , and the second thread reads the variable x twice.

Figure 5 shows how CDSHECKER builds up an execution. It begins with the initialization of the variable x . We assume that threadA runs both of its stores next. The right column shows the various relations in the C/C++ memory model. We show the constraints CDSHECKER constructs on the modification order using green arrows labeled *mo*. The modification order is required by the C/C++ standard to be acyclic—if a cycle is generated in the modification order, the execution is infeasible. After executing the store instructions, CDSHECKER next executes the loads from ThreadB. At each load, it constructs a *may-read-from-set* that contains the set of stores that the happens-before relation allows the load to see. In the example, both of the loads read-from the initial value of x .

After constructing the initial execution, CDSHECKER backtracks to explore other values from the may-read-from-sets. Figure 6 shows the construction of a later execution in the exploration process. In this execution, we assume that the first load in ThreadB reads from the store of 2 to x . CDSHECKER then constructs the may-read-from-set for the second load. This set contains the values $\{0, 1, 2\}$ even though only the value 2 is actually feasible. We assume that CDSHECKER explores the second load reading from the store of 1 to x . This adds the modification edge constraint shown by dashed green line and introduces a cycle into the modification order graph. Thus this execution is not feasible and CDSHECKER backtracks to explore other executions. CDSHECKER

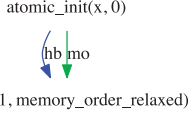
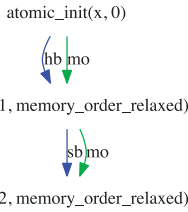
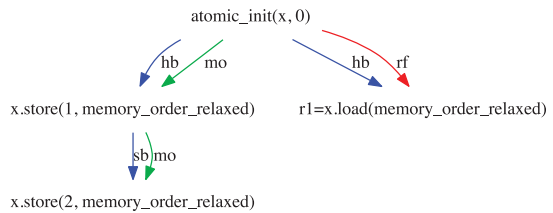
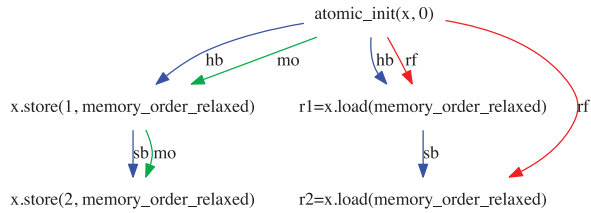
Trace	Graph representation of relations
1. <code>atomic_init(x, 0);</code>	<code>atomic_init(x, 0)</code>
2. <code>atomic_init(x, 0);</code> <code>x.store(1, relaxed);</code>	 <p><code>atomic_init(x, 0)</code> <code>x.store(1, memory_order_relaxed)</code></p>
3. <code>atomic_init(x, 0);</code> <code>x.store(1, relaxed);</code> <code>x.store(2, relaxed);</code>	 <p><code>atomic_init(x, 0)</code> <code>x.store(1, memory_order_relaxed)</code> <code>x.store(2, memory_order_relaxed)</code></p>
4. <code>atomic_init(x, 0);</code> <code>x.store(1, relaxed);</code> <code>x.store(2, relaxed);</code> <code>r1 = x.load(relaxed)=0;</code>	 <p><code>atomic_init(x, 0)</code> <code>x.store(1, memory_order_relaxed)</code> <code>x.store(2, memory_order_relaxed)</code> <code>r1=x.load(memory_order_relaxed)</code></p> <p>may-read-from-set = {0, 1, 2}</p>
5. <code>atomic_init(x, 0);</code> <code>x.store(1, relaxed);</code> <code>x.store(2, relaxed);</code> <code>r1 = x.load(relaxed)=0;</code> <code>r2 = x.load(relaxed)=0;</code>	 <p><code>atomic_init(x, 0)</code> <code>x.store(1, memory_order_relaxed)</code> <code>x.store(2, memory_order_relaxed)</code> <code>r1=x.load(memory_order_relaxed)</code> <code>r2=x.load(memory_order_relaxed)</code></p> <p>may-read-from-set = {0, 1, 2}</p>

Fig. 5. Exploration process.

contains an optimization that allows rolling back just the last load operation to make the exploration of alternative reads-from values efficient.

4.3. Transitions with Multiple Behaviors

We now discuss another major departure from DPOR, which comes from the nature of relaxed memory models. On one hand, DPOR assumes that all memory modifications form a consistent total ordering and that all memory accesses read only the last prior value written to memory. However, it is clear that the relaxed C/C++ memory model does not fit this model. More precisely, while the union of the *sb*, *hb*, and *sc* relations must be acyclic and consistent with some interleaving of threads, the addition of *rf* and *mo* introduces the possibility of cycles. Therefore, in order to explore a program's state space using a linear, totally ordered execution trace, we must account for behaviors which are inconsistent with the execution order.

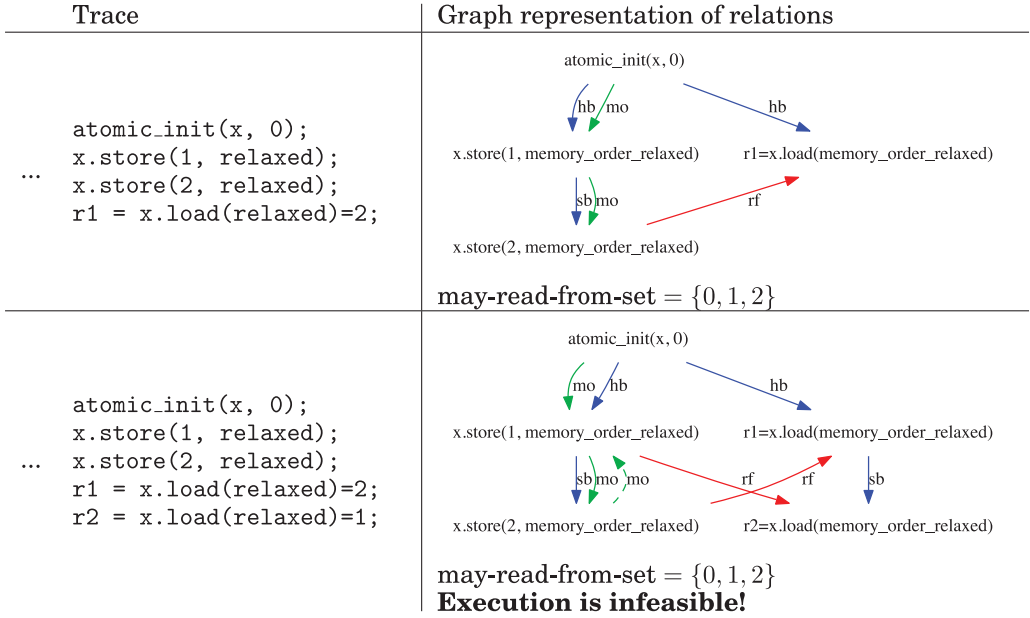


Fig. 6. Exploration process continued.

In order to explore a relaxed model, a backtracking-based search not only must select the next thread to execute but also must decide how that thread's next step should behave. We represent this notion in Figure 3 as a backtracking iteration not only over *threads* but over *behaviors* (the aforementioned inner loop). Together, a thread choice and behavior selection define a unique state transition.

A key source of different transition behaviors arises from the reads-from relation—in C/C++, loads can read from modifications besides simply the “last” store to an object. We introduce the concept of a *may-read-from set* to enumerate the stores that appear earlier in the execution order that a given load may read from.³ When we execute a load Y , we build the may-read-from set as a subset of $stores(Y)$ (the set of all stores to the same object from which Y reads):

$$\begin{aligned}
may-read-from(Y) = \{ & X \in stores(Y) \mid \neg(Y \xrightarrow{hb} X) \wedge \\
& (\nexists Z \in stores(Y). X \xrightarrow{hb} Z \xrightarrow{hb} Y) \}.
\end{aligned}$$

The clause $\neg(Y \xrightarrow{hb} X)$ prevents a load from seeing values from stores that are ordered later by happens-before, and the clause $(\nexists Z \in stores(Y). X \xrightarrow{hb} Z \xrightarrow{hb} Y)$ prevents a load from observing stores that are masked by another store.

Successive executions then iterate over this set, exploring executions in which a load may read from each one of the potential stores. Each execution forms a different *rf* relation (and, by extension, *mo* and *hb* relations). If $\nexists X \in may-read-from(Y)$ such that $X \xrightarrow{hb} Y$, then we report an *uninitialized load*—a bug in the program under test.

The reads-from mechanism allows CDSHECKER to explore most of the behaviors of the C/C++ memory model without rearranging the thread schedule. In fact, in the

³Loads can also read from stores that appear later in the execution order. Section 6 presents our approach for handling this case.

absence of synchronization or sequentially consistent operations, CDSCHECKER does not use the DPOR backtracking mechanism to change the thread schedule at all.

4.4. Handling Sequential Consistency

The memory model guarantees a total ordering sc over all seq-cst operations. CDSCHECKER forces the sc relation to be consistent with the execution order. Thus CDSCHECKER relies on a modified DPOR-like algorithm to rearrange the scheduled execution order to implement sequentially consistent operations—CDSCHECKER identifies conflicting sequentially consistent operations and sets backtracking points as described in the DPOR algorithm. We combine DPOR with sleep sets [Godefroid 1996]. Note that in addition to using DPOR-style backtracking for maintaining a consistent sc ordering, we use it to ensure that hb is consistent with the execution order (see Section 6.3) and to explore the behaviors of higher-level synchronization primitives (mutexes and condition variables).

4.5. Happens-Before and Clock Vectors

In the absence of consume operations, happens-before is simply the transitive closure of synchronizes-with and sequenced-before. Thus, CDSCHECKER represents happens-before succinctly using a Lamport-style clock vector [Lamport 1978]. Events consist of atomic loads and stores, thread creation and join, mutex lock and unlock, and other synchronizing actions. Every event increments its own thread's clock (representing a step in sb), and then CDSCHECKER tags the event with the current thread's clock vector. Synchronization between two threads— T_i and T_j , where $T_i \xrightarrow{sw} T_j$ —should merge T_i 's clock vector with T_j 's clock vector, according to a pairwise maximum over all the thread clocks. We assign the resulting vector to the synchronizing event in T_j .

Some processor architectures (e.g., Power and ARM) respect low-level data dependencies such that while synchronization is generally expensive it can be cheapened for operations that are data dependent on a synchronizing memory access. Thus, C and C++ provide release/consume atomics as a weaker, dependency-based synchronization alternative to release/acquire. However, on stronger architectures (e.g., x86), consume provides no benefit over acquire, so we find it reasonable to omit support of consume in favor of minimizing CDSCHECKER's compiler-specific dependencies.

Still, given compiler and runtime support for computing the intrathread *carries a dependency* relation, we can extend our approach to support release/consume synchronization. One approach is to associate a secondary clock vector with a program event if it is *dependency ordered* (§1.10p9-10 [ISO/IEC 14882:2011]) after a store-release from a different thread—never forwarding the clock vector to subsequent actions ordered only by sequenced-before. When present, the model-checker would use this secondary clock vector for detecting data races and computing may-read-from sets. A store-release that follows operations that are dependency ordered would then merge the clock vectors for all operations sequenced before the store, transferring them to any operation with which it synchronizes.

4.6. Deadlock Detection

CDSCHECKER can easily detect deadlocks during its state space search. Given our knowledge of the next transition $next(s, p)$ for each thread p , it is straightforward to check if a thread's next transition is disabled (i.e., blocking due to a mutex operation or a thread join) and waiting on another thread. Then, CDSCHECKER can simply check for a circular wait by traversing the chain of waiting threads whenever a thread takes a step; if that thread can reach itself, then we report a deadlock to the user.

5. CONSTRAINTS-BASED MODIFICATION ORDER

The modification order relation presents unique challenges and opportunities in model-checking C/C++, as program executions never directly observe it. One approach taken by other tools (e.g., CPPMEM) is to exhaustively enumerate both the *mo* and *rf* relations, discarding any executions that violate memory model constraints. In the following subsections, we present a new approach, in which we record *mo* not as an exhaustively explored ordering but as a constraints graph, in order to reduce (by an exponential factor) the work spent on both infeasible and redundant executions.

5.1. Motivation

We could constructively maintain the modification order using an approach similar to CPPMEM—as soon as CDSCHECKER executes a store, we could assign it an absolute ordering within *mo*. However, at the time of a store's execution, a program has not formed many constraints for its modification order, so we would have to choose its ordering arbitrarily and then explore an additional exponential space of reorderings to enumerate all possible choices. This would often incur a very large overhead, as constraints observed later in the execution often invalidate many orderings and many different modification orderings produce no visible difference in program behavior.

Therefore, rather than constructively (and expensively) maintaining *mo* as a total ordering, we chose a lazy approach to the modification order. CDSCHECKER represents *mo* as a set of constraints, built as a constraints graph—the *modification order graph*, or *mo-graph*. A node represents a single store in the execution and an edge directed from a node *A* to a node *B* represents the constraint $A \xrightarrow{mo} B$. CDSCHECKER dynamically adds edges to the *mo-graph* as *hb* and *rf* relations are formed, as described in Section 5.5. Then CDSCHECKER must only ensure that exploration of a particular execution yields a satisfiable set of *mo* constraints—or equivalently, an acyclic *mo-graph*. A cyclic *mo-graph* implies an ill-formed execution, and so CDSCHECKER can discard the current execution and move on to explore the next execution.

5.2. Representing the Memory Model as Constraints

The memory model specifies several properties governing the interplay of *rf*, *hb*, *sc*, and *mo*. We contribute the insight that these properties can be formulated as constraints on the modification order. Thus, we present them as implications, shown in the left-to-right progressions in Figure 7. For example, consider READ-READ COHERENCE (§1.10p16 [ISO/IEC 14882:2011]); we can say that any pair of loads *A* and *B* with a corresponding pair of stores *X* and *Y* (all operating on the same object *v*), where

$$X \xrightarrow{rf} A, Y \xrightarrow{rf} B, \text{ and } A \xrightarrow{hb} B$$

imply a particular modification ordering for *X* and *Y*—namely that $X \xrightarrow{mo} Y$. In other words, such a constraint prevents other loads from observing *X* and *Y* in the reverse order. The reader can examine the similar WRITE-READ, READ-WRITE, or WRITE-WRITE coherence requirements.

In addition to COHERENCE, we summarize the following memory model requirements:

- SEQ-CST/MO CONSISTENCY: A pair of seq-cst stores must form *mo* consistently with *sc* (§29.3p3 [ISO/IEC 14882:2011])
- SEQ-CST WRITE-READ COHERENCE: A seq-cst load must read from a store no earlier (in *mo*) than the most recent (in *sc*) seq-cst store (§29.3p3 [ISO/IEC 14882:2011])
- RMW/MO CONSISTENCY: A read-modify-write must be ordered after the store from which it reads (§29.3p12 [ISO/IEC 14882:2011])

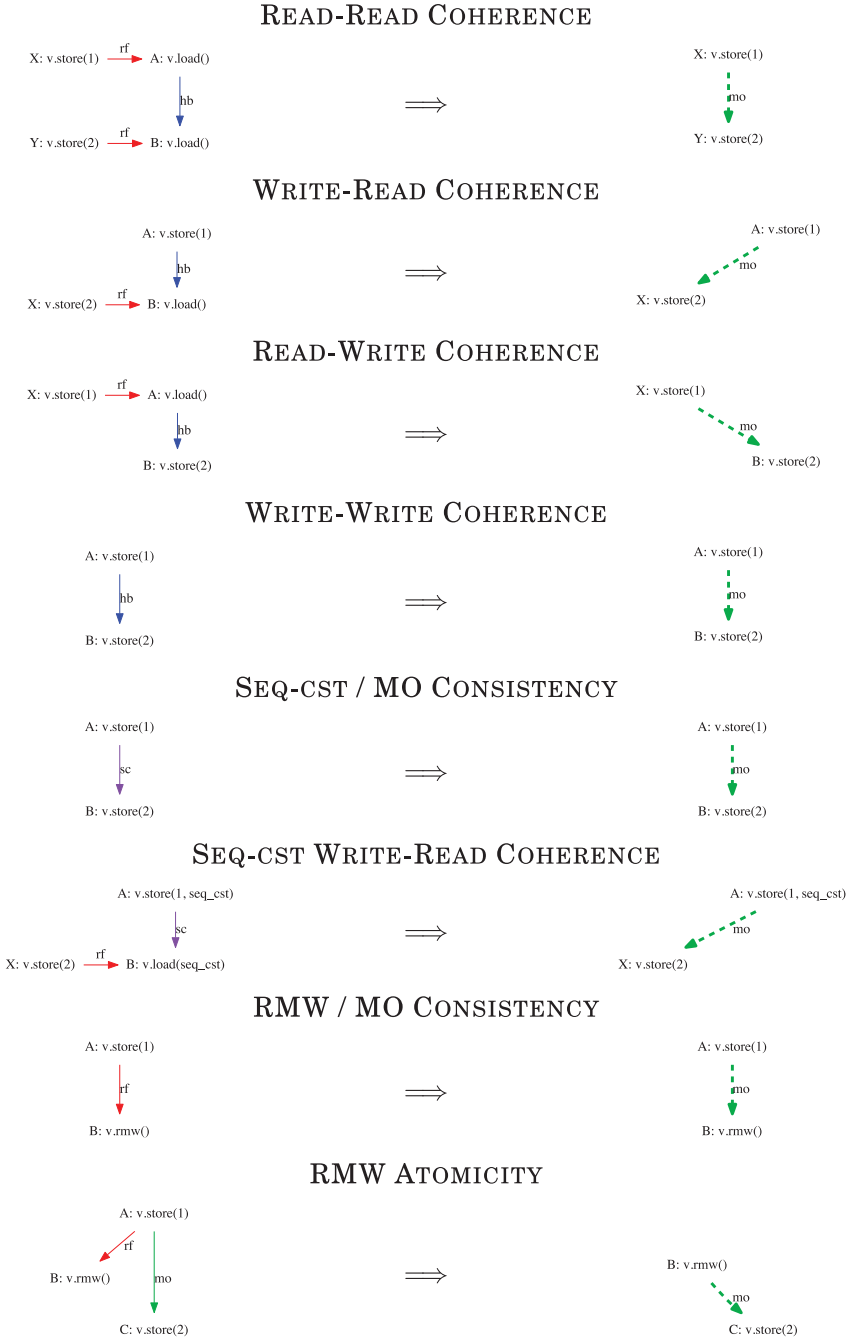


Fig. 7. Modification order implications. On the left side of each implication, A , B , C , X , and Y must be distinct.

—RMW ATOMICITY: A read-modify-write must be ordered immediately after the store from which it reads (§29.3p12 [ISO/IEC 14882:2011])

5.3. Example

We examine the application of these constraints in the *mo*-graph using the following example:

```

1  atomic<int> x(0);
2
3  void threadA() {
4      x.store(1, memory_order_relaxed); // A
5      x.store(2, memory_order_relaxed); // B
6  }
7  void threadB() {
8      int r1 = x.load(memory_order_relaxed); // C
9      int r2 = x.load(memory_order_relaxed); // D
10 }
```

As CDSCHECKER executes the stores in `threadA()`, the WRITE-WRITE COHERENCE constraint implies a *mo* edge from store A to store B. Consider an example execution where load C has read from store B. Now consider the possibility of load D reading from store A. In such a case, READ-READ COHERENCE would require a *mo*-graph edge from B to A—forming a *mo*-graph cycle between A and B and prohibiting such an execution.

5.4. Search Space Reduction

We will demonstrate in a short example how our approach to constraints-based modification order reduces the inefficient exploration of redundant and infeasible execution behaviors in comparison to simpler approaches, such as CPPMEM's. Consider the following program, written in the syntax style of CPPMEM, where `{{{ statement1; ||| statement2; }}}}` means that *statement1* and *statement2* execute in parallel.

```

1  atomic<int> x = 0;
2
3  {{{ x.store(1, relaxed);
4      ||| x.store(2, relaxed);
5      ||| x.store(3, relaxed); }}}
6
7  r1 = x.load(relaxed);
```

CPPMEM's search algorithm considers that a load may read from any store in the program, and that those stores may have any arbitrary (total) modification ordering; it performs no analysis of the interplay of reads-from, modification order, and happens-before when enumerating candidate executions. Thus in this program, it enumerates 24 potential modification orderings for the three stores and one initialization (the permutations of a four-element sequence) and considers four potential stores to read from at line 7, yielding 96 combinations. However, one can easily see that there are actually only three valid behaviors for this program: those represented by the results `r1 = 1`, `r1 = 2`, or `r1 = 3`. In fact, many of the modification orderings are impossible; none of the stores can be ordered before the initialization, due to WRITE-WRITE COHERENCE. Additionally, many of the remaining modification orderings are irrelevant; this program only cares which of the stores appears *last* in the order, as this is the store from which the load must read.

CDSCHECKER's constraint construction captures exactly the observations of the previous paragraph because it only establishes modification orders as they are observed. So, for example, when line 7 reads a value of 3, CDSCHECKER rules that line 5 must be ordered after all of the other stores (due to WRITE-READ COHERENCE), but it does not bother enumerating the modification ordering of the remaining stores, since no operations observe their ordering. Additionally, CDSCHECKER can avoid exploring executions

where line 7 reads a value of 0, since such a *rf* relation would immediately generate a *mo*-cycle. In fact, CDSCHECKER explores exactly the three consistent behaviors without enumerating the other 93 redundant or inconsistent orderings.

5.5. Optimized Constraint Construction

CDSCHECKER derives its *mo*-graph using the implications presented in Figure 7. However, these requirements are nontrivial to implement, as a naïve approach involves a search of the entire execution history every time we update *hb*, *sc*, *rf*, or *mo*—that is, at least once for every program operation. But with a few observations and optimizations, we can efficiently build this graph.

COHERENCE: Because the antecedents of the four coherence implications involve only the *rf* and *hb* relations on a single object, we must compute additional *mo* edges only on exploration of new loads and stores or when *rf* or *hb* are updated. Now, consider an implementation of READ-READ COHERENCE. Rather than searching for all pairs of loads ordered by happens-before, we conclude that when exploring a new load *B*, we only need to consider the most recent load *A_i*, from each thread *i*, which happens before *B* and reads from the same object. For any other load *Z* (reading the same object) that happens before *B*, either $Z = A_i$ for some *i*, or else $Z \xrightarrow{sb} A_j$ for some *j*. By induction, then, CDSCHECKER must already have considered any prior loads.

The other three coherence conditions have similar inductive behavior, and so we can limit the number of computations necessary: two rules correspond to a new load (READ-READ and WRITE-READ), and two rules correspond to a new store (READ-WRITE and WRITE-WRITE); all four apply to a read-modify-write. Furthermore, by a similar inductive argument, we can combine the coherence rules such that it is only necessary to search for the most recent load or store (and not both). Finally, note that lazy updates of *hb* (see Section 7) must trigger similar constraint updates.

SEQ-CST/MO CONSISTENCY: Upon exploration of a new seq-cst store, CDSCHECKER must add an edge from the most recent seq-cst store to the same object in the execution order (and hence, in *sc*) to the current store. By a simple induction, this computation will soundly cover all seq-cst stores, if applied at exploration of each new seq-cst store.

SEQ-CST WRITE-READ COHERENCE: In similar fashion to SEQ-CST/MO CONSISTENCY, CDSCHECKER must search for the most recent seq-cst store upon exploration of a seq-cst load.

RMW/MO CONSISTENCY: Consistency is trivial; CDSCHECKER simply adds a *mo*-graph edge whenever a read-modify-write executes.

RMW ATOMICITY: Not only must CDSCHECKER be sure to order a RMW *B* after the store *A* from which it reads (i.e., RMW/MO CONSISTENCY), it must also ensure that any store *C* ordered after *A* is also ordered after *B*. Thus, CDSCHECKER records metadata in each graph node *A* to show which RMW (if any) reads from *A*; a new edge from *A* to *C* then creates an additional edge from *B* to *C*. Note that RMW CONSISTENCY and ATOMICITY combine to ensure that two RMW's cannot read from the same store. If two RMW's, *B* and *C*, each read from *A*, then the *mo*-graph forms a cycle between *B* and *C*, invalidating the current execution.

5.6. Modification Order Rollback

A naïve implementation of our *mo*-graph approach would have to rollback the entire execution whenever it assigns a load to read from a store that results in immediate violations of *mo*-graph constraints. To optimize for this common case, our *mo*-graph supports rolling back the most recent updates. Then in Section 5.3's example, for instance, CDSCHECKER can check whether it is feasible for load *D* to read from store *A* before committing *D* to read from *A*. This reduces the number of infeasible executions that must be backtracked.

6. RELAXING READS-FROM

The framework as described thus far can only simulate loads that read from stores that appear earlier in the execution order. However, the C/C++ memory model allows executions in which the union of the *rf* and *sb* relations is cyclic, implying that regardless of the execution order, this strategy will not suffice to model all legal executions. The counterintuitive result (i.e., $\{r1 = r2 = 1\}$) from our example in Figure 2 is one such execution.

To fully model all behaviors allowed by the memory model, CDSCHECKER must also model executions in which values flow backwards in the execution order, allowing loads to read from stores which have not yet occurred at the time of the load—we say that such loads are observing *future values*.

The key idea for modeling future values is to leverage backtracking of transition behaviors to allow loads to read from stores that appear later in the execution order. As an illustrative example, consider—without loss of generality—an execution order of the example from Figure 2 in which all statements in threadA appear before all statements in threadB. In such an execution, it is relatively easy to see how to simulate $r2 = 1$ from the counterintuitive result. However, simulating $r1 = 1$ requires that the load in Line 4 of threadA read from the store in Line 10 of threadB. The challenge here is that this load appears before CDSCHECKER has even seen the store.

To address this challenge, we introduce an extension to our may-read-from set: the *futurevalues* set which associates pairs (v, t) with loads X , where v is a predicted future value (written by thread t) that X may read. Suppose an execution encounters a store Y and a number of loads X_1, X_2, \dots, X_n from earlier in the execution order. As long as X_i does not happen before Y (i.e., $\neg X_i \xrightarrow{hb} Y$), it may read from Y , and so CDSCHECKER will add the pair $(value(Y), thread(Y))$ to the set *futurevalues* (X_i) for each $i = 1, \dots, n$ (if Y 's thread did not yet exist at the time of X_i , it will use an appropriate ancestor thread). On subsequent executions, CDSCHECKER will diverge from previous behavior and explore executions in which load X_i chooses a pair (v, t) from *futurevalues* (X_i) and reads the value v . In our example, this allows CDSCHECKER to simulate the load reading the value 1 that is written by the later store. We next need to verify that a later store (from thread t or one of its descendants) will still write the value 1 and that the memory model constraints allow the load to read from the store.

6.1. Promising a Future Value

When CDSCHECKER backtracks in order to evaluate a load using a future value—a *speculative* load behavior—we cannot precisely associate the future value with a particular store that will generate it; any dependencies on the value observed might cause divergent program behavior, so in the new execution, several later stores may generate the observed value (validating the observation) or else such a store may no longer occur (making the observation infeasible).

For every speculative load (v, t) made in an execution, CDSCHECKER establishes a *promised future value* (or *promise*)—an assertion that, at some later point in the execution, thread t (or one of its descendants) will perform a store that can legally pass value v back to the speculative load. In our example, CDSCHECKER would generate a promise when it simulates the load in Line 4 reading a future value of 1 from the pair $(1, threadB)$. This promise asserts that a later store from threadB will write the value 1 and that the load can read from that store.

Once CDSCHECKER detects such a store, we consider the promise to be *satisfied*, and we can remove the promise from future consideration. In our example, the promise would be satisfied when the store in Line 10 writes the value 1.

We must allow a speculative load to read not only from the first satisfactory store to follow it in the execution order but also from subsequent stores. Thus, we model two

execution behaviors for each store: one in which the store chooses to satisfy a matching promise and one in which it chooses not to do so, instead allowing a later store to satisfy it.

Sending back a future value may cause an execution to diverge such that its promise is never satisfied nor can the model-checker ever rule out the possibility that it will eventually be satisfied. To address this, promises expire if they are not resolved by the expected number of program operations plus a tunable threshold.

6.2. Treating Promises as Stores

After a load observes a promised future value, we assume that some store will eventually satisfy it, and so we must allow subsequent loads to observe the same store. Rather than generating independent promises for each instance of the observed value, we track these speculative *rf* relations by treating the promise as a placeholder for a future store; we can then add this placeholder to the appropriate loads' may-read-from set. In practice, then, the may-read-from set for a given load is composed of three separate components: stores from earlier in the execution order, its *futurevalues* set, and the set of yet-unsatisfied promises for the same memory location.

Over the lifetime of a promised future value (that is, the period between its generation and satisfaction/invalidation), we can build a form of modification order constraints for it in much the same way as with nonspeculative stores. For example, whenever a promise can be satisfied only by a single thread, we can order it after all operations in that thread (in the *sb* relation and therefore in *hb* and *mo*); and we know which loads read from the promised value, so we can apply the COHERENCE implications.

These *mo* constraints are useful for reasoning about the feasibility of a promise. For instance, if an execution observes promised values in an inconsistent order, we can detect a graph cycle and terminate the execution. Additionally, the modification order can tell us when a thread can no longer satisfy a promise, aiding us in eliminating unsatisfiable promises. For example, COHERENCE implies that a load *A* cannot read from a store *C* whenever there exists a store *B* such that $A \xrightarrow{hb} B \xrightarrow{mo} C$. Thus, when such a *B* exists, we eliminate *C*'s thread from satisfying a promise to load *A*. If instead we encounter a store that satisfies a promise, then we can merge the promise and store nodes in the constraints graph, retaining the constraint information that we have gathered so far.

6.3. Synchronization Ordering

Allowing loads to see values written by stores that appear later in the execution order may yield a synchronization relation directed backward in the execution order. Such a synchronization would break any code (e.g., libraries or operating system calls) that used uninstrumented memory accesses to implement normal loads and stores. Moreover, it would require complicated mechanisms to ensure that normal shared memory accesses observe the correct values.

We observe that since the specification guarantees that happens-before is acyclic (§1.10p12 [ISO/IEC 14882:2011]), we can address this problem by ensuring that we always direct *hb* forward in the execution trace (note that *hb* must be acyclic). If *hb* is always consistent with the execution order of program fragments, then normal loads and stores (including those in libraries and in many operating system invocations) will behave as expected; reading the last-written value from memory will always be consistent with the happens-before behavior intended in the original program. This also explains another design decision made in CDSCHECKER: rather than instrumenting all shared memory loads to read from the correct stores, CDSCHECKER generally leaves

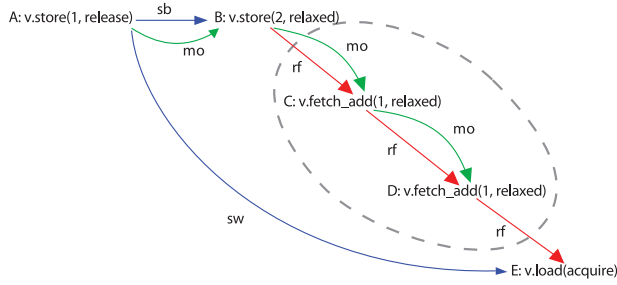


Fig. 8. An example release sequence. Program execution is generally ordered from left to right.

nonatomic memory accesses uninstrumented (with the exception of a happens-before race detector).

We now present a few observations we use in guaranteeing that *hb* remains consistent with the execution order. Because *sequenced-before* is trivially consistent, our guarantee reduces (in the absence of *memory_order_consume*) simply to the guarantee that *synchronizes-with* forms consistently with the execution order. We achieve this guarantee in two parts. First, whenever we detect a synchronization relation that is inconsistent with the execution order (i.e., $X \xrightarrow{sw} Y$ but Y appears earlier in the execution order than X), we terminate the execution. Second, we must ensure that whenever such termination occurs, we will also explore an equivalent execution with well-formed synchronization; thus, we backtrack whenever an execution trace encounters a load-acquire ordered before a store-release to the same location. Finally, note that if we extend our approach to include consume operations as described in Section 4.5, this discussion of *synchronizes-with*, load-acquire, and store-release can be trivially extended to *dependency-ordered-before*, load-consume, and store-release.

7. RELEASE SEQUENCES

Thus far, our discussion has assumed that release/acquire synchronization only occurs when a load-acquire reads from a store-release. Unfortunately, such a simplistic synchronization criteria would force implementations of common synchronization constructs to declare more atomic operations with release or acquire memory orders instead of relaxed and thus generate suboptimal compiler output (e.g., with extraneous fence instructions). To address this problem, the C/C++ memory model defines a *release sequence* (§1.10p7 [ISO/IEC 14882:2011]), which both extends the conditions under which a load-acquire and store-release synchronize and correspondingly increases the complexity of determining synchronization relationships as they form.

We summarize the definition (note that all operations in consideration must act on the same atomic object): A release sequence consists of a release operation A —the *release head*—followed by a contiguous subsequence of the modification order consisting only of (1) stores in the same thread as A or (2) read-modify-write operations; a non-RMW store from another thread breaks the sequence. Figure 8 shows a release sequence headed by A and followed by modifications B , C , and D ; note how a chain of RMW's (encircled with a dotted boundary) may extend the release sequence beyond the thread which contained the release head A .

Then we redefine release/acquire synchronization⁴: A store-release A synchronizes with a load-acquire B if B reads from a modification M in the release sequence headed

⁴This definition subsumes the previous definition; a store-release is in the release sequence headed by itself.

by A [ISO/IEC 14882:2011]. In Figure 8, the load-acquire E reads from D , which is part of the release sequence headed by A —so $A \xrightarrow{sw} E$.

This definition of release/acquire synchronization poses several challenges as we attempt to eagerly form the sw relation, since CDSCHECKER does not establish the modification order eagerly. For one, future values allow the possibility of lazily satisfied reads-from relationships, so we may not establish the modification order of a read-modify-write operation until its read portion is satisfied. More generally, recall that CDSCHECKER uses a constraints-based approach to establishing modification order, so at a given point in a program execution, two modifications may be unordered with respect to each other, leaving us uncertain as to whether a given sequence of modifications is contiguous (e.g., in Figure 8 we must guarantee that no non-RMW store M exists in another thread such that $A \xrightarrow{mo} M \xrightarrow{mo} B$). Either of these two factors may prevent CDSCHECKER from eagerly deciding synchronization when exploring load-acquire operations, so we resort to lazy evaluation.

Lazy evaluation of release sequences means that for any release/acquire pair whose corresponding release sequence we cannot establish or invalidate with certainty at first discovery, CDSCHECKER leaves the pair unsynchronized⁵ and places it into a set of pending release sequences, along with any unconstrained (or *loose*) stores which might break up the release sequence. By the end of the execution, a program will usually build up enough constraints to mo such that CDSCHECKER can resolve these pending release sequences deterministically and drop the release/acquire pair from the pending set. However, if at the end of a program execution the constraints are still undecided for one or more pending release sequences, then CDSCHECKER must search for a particular constraints solution by selecting one of two possibilities for each pending sequence: either that one of the loose stores breaks the sequence or that the sequence is contiguous, causing the release/acquire pair to synchronize. Selections may not be independent (one contiguous release sequence may imply another, for instance) and therefore many solutions are infeasible.

Now, sound model-checking does not require exploration of all possible solutions, as some solutions only allow a subset of behaviors exhibited by an equivalent, less-constrained execution. Particularly, in a constraints problem where one solution might result in no additional synchronization and a second solution results in one or more additional release/acquire synchronizations, the first solution must exhibit a superset of the erroneous behaviors (e.g., data races) exhibited by the second one. Thus, an optimized search would prioritize constraint solutions where all pending release sequences are broken (i.e., noncontiguous); such a minimally synchronizing solution precludes the need to explore other release sequence combinations in which the release sequences resolve to some nonempty set of synchronizations.

The discussion so far has failed to account for the effect of delayed synchronization on the rest of the model-checking process, where we previously assumed that CDSCHECKER establishes synchronization eagerly. When delayed resolution of a release sequence causes synchronization, CDSCHECKER must perform a number of updates for all clock vectors and mo -graph edges that are dependent on this update. A newly established relation $X \xrightarrow{sw} Y$, where Y is in the interior of the execution trace, must generate a cascading update in the clock vectors for all operations which have previously synchronized with Y (previously unordered operations are unaffected). Additionally, each updated clock vector may yield new information about mo constraints. Thus, after such a lazy synchronization $X \xrightarrow{sw} Y$, CDSCHECKER performs an iterative pass over all

⁵Lazy synchronization is acceptable because an execution in which synchronization does not occur can only exhibit a superset of behaviors seen in the equivalent synchronizing execution.

operations ordered after Y in the execution order, recalculating the happens-before clock vectors and mo constraints.

Lazy synchronization presents a few other problems for CDSCHECKER. For one, it may reveal that rf is inconsistent with hb long after the relevant load, causing unnecessary state-space exploration. Furthermore, because lazy synchronization may protect the memory accesses which previously constituted a data race, our happens-before race detector must delay recognizing data races until there are no pending synchronizations.

Despite this discussion of the complexity involved in release sequences, we suspect that most valid programs will never incur significant overhead when resolving release sequences. In our tests of real data structures, all release sequences have been trivially resolvable: Either a load-acquire reads directly from a store-release or it reads from a chain of one or more RMW's.⁶ With the former, synchronization is immediately evident, and with the latter, the chain of RMW's guarantees a contiguous subsequence of the modification order (see Figure 7, RMW ATOMICITY). Such programs will never incur the costs of the more complicated constraints checks for determining a contiguous subsequence of mo .

8. FENCES

In addition to the atomic loads, stores, and read-modify-writes discussed previously, C and C++ support atomic fence operations. C/C++ fences loosely imitate the low-level fence instructions used in multiprocessors for ordering memory accesses and are included to allow developers to more efficiently represent their algorithms. Fences may use the release, acquire, `rel_acq`, or `seq_cst` memory orders (`relaxed` is a no-op and `consume` is an alias for `acquire`, §29.8p5 [ISO/IEC 14882:2011]) and have additional modification order constraints and synchronization properties, whose support we will discuss in this section.

8.1. Fence Modification Order Constraints

C and C++ introduce several rules governing rf and mo when dealing with sequentially consistent fences. As in Section 5.2, we transform these rules directly into modification order implications for use by CDSCHECKER. Figure 9 presents the set of modification order implication rules for fences.

- SC FENCES RESTRICT RF: Seq-cst fences impose restrictions on the “oldest” store from which a load may read (§29.3p4-6 [ISO/IEC 14882:2011]).
- SC FENCES IMPOSE MO: A pair of stores separated by seq-cst fences must form mo consistently with sc (§29.3p7 [ISO/IEC 14882:2011]). Notably, the C++ specification leaves out the COLLAPSED constraints that are presented here, but they are included in the formal model developed for Batty et al. [2011]. The report for C++ Library Issue 2130 indicates that the specification committee plans to include these rules in future revisions.

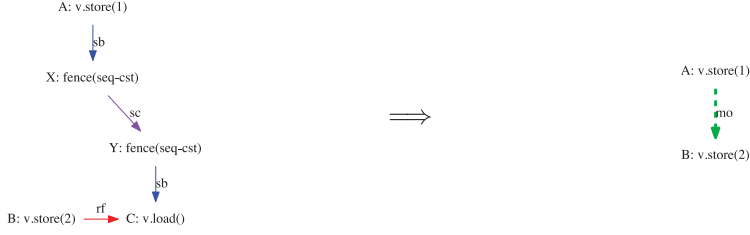
These implications can be applied using similar induction arguments to those developed in Section 5.5; because sc is a total order, we can always find the last store A in each thread that satisfies the left-hand side (if any exists). Any prior store must already be ordered before A in mo , and so we must look no further than A when building constraints for a newly explored program operation.

8.2. Fence Synchronization

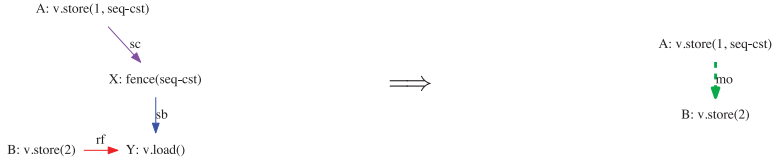
Besides the modification order constraints imposed by sequentially consistent fences, fences can induce synchronization (sw) via an extension to release sequences. The

⁶We also observed simple release sequences in the presence of fence operations (see Section 8).

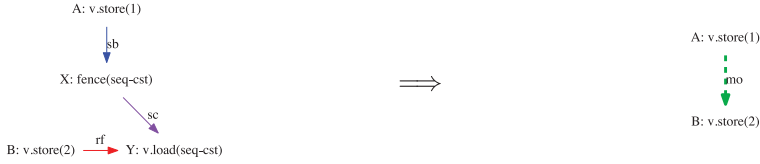
SC FENCES RESTRICT RF



SC FENCES RESTRICT RF (COLLAPSED STORE)



SC FENCES RESTRICT RF (COLLAPSED LOAD)



SC FENCES IMPOSE MO

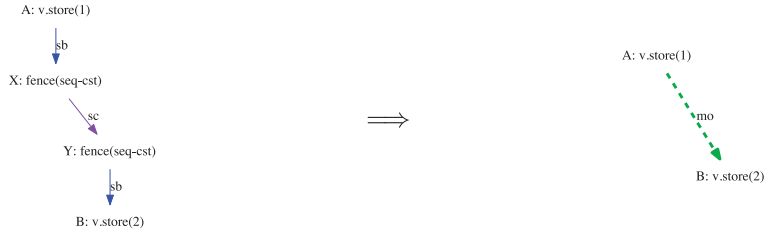
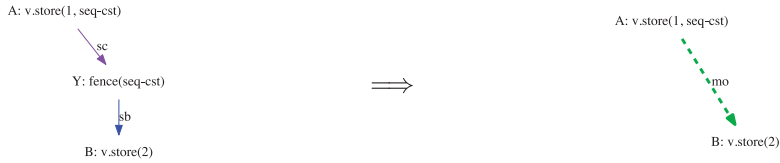
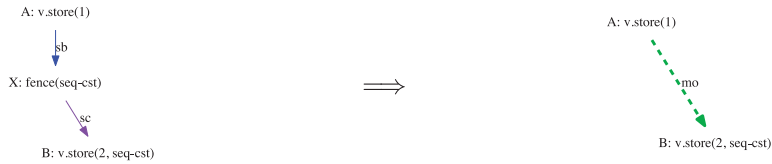
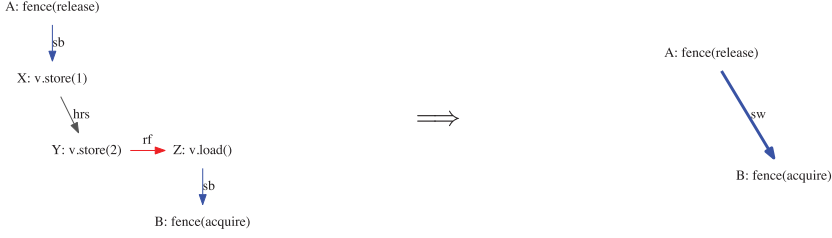
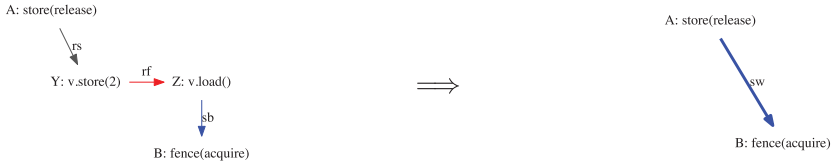
SC FENCES IMPOSE MO (COLLAPSED 1ST STORE)SC FENCES IMPOSE MO (COLLAPSED 2ND STORE)

Fig. 9. Fence modification order implications. On the left side of each implication, A, B, C, X, and Y must be distinct.

FENCE SYNCHRONIZATION



FENCE SYNCHRONIZATION (COLLAPSED STORE)



FENCE SYNCHRONIZATION (COLLAPSED LOAD)

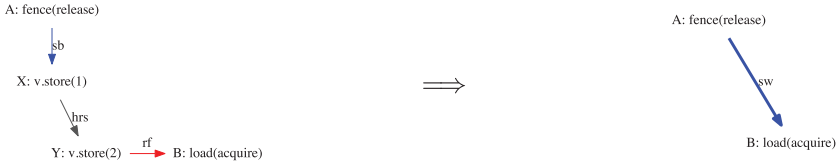


Fig. 10. Fence synchronization implications.

specification defines a *hypothetical release sequence* headed by a store X as the release sequence that would exist if X were a release operation. We will say that if store Y is part of the hypothetical release sequence headed by X , then $X \xrightarrow{hrs} Y$ (or, similarly, $X \xrightarrow{rs} Y$ for true release sequences).

We take the synchronization implications in Figure 10 directly from §29.8p2-4. Informally, these rules cause a load-relaxed followed by a fence-acquire to act like a load-acquire and cause a store-relaxed preceded by a fence-release to act like a store-release.

These synchronization implications can be easily computed with simple extensions to the methods described in Section 7. In fact, a hypothetical release sequence makes synchronization detection even simpler than with traditional release sequences because the “loose” store problem is no longer an issue; as soon as we find *any* store X such that $X \xrightarrow{hrs} Y$, there is no longer a need to establish a contiguous modification order: we only need to search for the last fence-release A that is sequenced before X in the same thread. In other words, hypothetical release sequence computations only require knowledge of *rf* (to follow the RMW chain, if any) and the intrathread ordering *sb* (to find prior fence-releases) but do not require querying the partially ordered *mo*-graph.

8.3. Fence Backtracking

Because fences can synchronize with other loads, stores, or fences, we must order them properly in the execution order such that their synchronization is consistent (recall Section 6.3). We extend our previous backtracking approach to accommodate any potential synchronization involving the fence rules in Section 8.2. So whenever

CDS_{CHECKER} observes an acquire B ordered earlier in the execution than a release A , and we determine that A may synchronize with B ($A \xrightarrow{sw} B$), we must backtrack to allow the thread which performed A to execute before B . Note that identifying such A and B may also involve identifying, for instance, an appropriate load/store pair X and Y (when applying FENCE SYNCHRONIZATION); similar observations can be made for the COLLAPSED synchronization rules.

As described in Section 4.4, we force sc to be consistent with the execution order and use DPOR backtracking to explore the necessary interleavings of conflicting seq-cst operations. To extend this to seq-cst fences, we simply say that a seq-cst fence conflicts with any other seq-cst operation.

9. PRUNING FUTURE VALUES

To reduce the search space generated by the exploration of future values, we developed a few optimizations. With these optimizations, we attempt to avoid introducing future values when their introduction is guaranteed to generate infeasible (or otherwise unnecessary) executions. Reductions in infeasible future values provide a compounding reduction in overhead, since such ill-advised values may generate a significant amount of unproductive exploration space in between the speculative load and its promise resolution—at which point we finally realize an execution-ending *mo*-graph cycle. Thus, we present a few derived constraints for pruning those future values which, when observed, would guarantee a cyclic *mo*-graph. Additionally, we introduce a few other optimizations for reducing redundant or otherwise unnecessary exploration.

For any load A and store X , we can show that $X \not\xrightarrow{rf} A$ whenever there exists a store B such that

$$A \xrightarrow{hb} B \wedge B \xrightarrow{mo} X.$$

Allowing $X \xrightarrow{rf} A$ would yield a *mo* cycle in B and X , due to READ-WRITE COHERENCE. Therefore, X should never send a future value to A . Without this constraint, CDS_{CHECKER} would let X send its future value to A , not recognizing the cycle until it established the *rf* edge concretely. Similarly, we do not send a future value from store B to load A if $A \xrightarrow{hb} B$.

Knowledge of promise behavior presents further opportunity for optimization of future values. If a store Y is scheduled to satisfy an outstanding promise P , then we limit the cases in which Y sends its future value to prior loads X_i —we avoid sending Y 's future value to any load X_i whose may-read-from set contains P (as a placeholder for Y). Specifically, Y does not send its future value to the load X which first generated promise P , nor to any load which follows X in the execution since such loads may also read from P (Y can, however, send its future value to loads prior to X).

A speculative load X can cause a later store Y to send a new future value back to X , even when Y actually depends on X . Such a cyclic dependence can potentially cause CDS_{CHECKER} to explore an infinite space of infeasible executions. We eliminate these cycles by an additional constraint when sending a future value from such a store Y to a load X ; we check whether there exists a yet-unresolved promise created by a speculative load Z , where Z is between X and Y in the execution order. If not, then Y can send its future value safely (subject to previously-discussed constraints). If such a Z does exist, however, we delay sending the future value until Z 's promise is resolved—breaking the cycle while still allowing noncyclic dependencies to be resolved.

The correctness of this optimization follows from the following argument. If the satisfying store S for Z does not depend on X observing Y 's future value, then Z 's promise will eventually be resolved and the future value will be sent. If the satisfying store S for

Z does depend on X observing Y 's future value, then either (1) X occurs after Z in the execution order and hence does not trigger the delay condition or (2) when Z eventually reads from a different store, Y can then add its future value to *futurevalues*(X) (Y can only depend on Z in the presence of a satisfaction cycle); the backtracking algorithm will later revisit the current situation without the need to send the future value as the value already exists in *futurevalues*(X).

10. LIVENESS AND FAIRNESS

Some programs present infinite spaces of execution when allowed to continually read a particular value from an atomic object, even after new values have been written; C and C++ require that these new values “become visible to all other threads in a finite period of time” (§1.10p25 [ISO/IEC 14882:2011]), posing a practicality problem for our exhaustive search. We conclude that, for some programs which rely on memory system liveness, we must trade state-space coverage for liveness. CDSCHECKER provides users with a runtime-configurable bound on the number of times a thread can read from the same store while the modification order contains another later store to the same location.

A related issue arises for sequentially consistent atomics; thread starvation can prevent some algorithms from terminating. CDSCHECKER supports the CHES [Musuvathi et al. 2008] fairness algorithm through the use of thread-yields placed in the program under test. Or, if a user cannot insert appropriate yields, we also support a tunable fairness parameter such that threads which are enabled sufficiently many times within an execution window without taking a step should receive priority for execution, allowing users to automatically balance fairness and completeness.

11. CORRECTNESS

Showing the soundness of any practical model checking for C/C++11 is a complex problem as it inherently involves some sort of precise prohibition against “out-of-thin-air” results and other types of satisfaction cycles [Batty et al. 2013; Vafeiadis et al. 2015]. Prohibiting “out-of-thin-air” results remains an open problem [Boehm and Demsky 2014]. While the C/C++11 standard included an attempt at prohibiting such results in §29.3p9, this attempt both failed to prohibit many troublesome satisfaction cycles and at the same time unintentionally disallowed reasonable implementations of relaxed atomics on architectures like ARM or POWER. As a result, this part of the standard has been modified in C++14 by N3786 [Boehm 2013] to simply forbid “out-of-thin-air” values without defining what they are.

11.1. Memory Model

The correctness of the CDSCHECKER algorithm depends first on framing the discussion about what it means to be correct. This is not straightforward, because there are known issues in the C/C++11 memory model such that it formally allows undesirable behaviors (“out-of-thin-air” value or “satisfaction cycles”) and it disallows harmless behaviors which are allowed on certain modern architectures and then only informally discourages various behaviors. While N3786 [Boehm 2013] removes the prohibition on the harmless behaviors for C/C++14, it does not define what an “out-of-thin-air” value is. Instead it simply forbids them without definition.

Thus, before we can show correctness, we must smooth some of the rough edges of the current memory model. One approach to prohibiting satisfaction cycles [Vafeiadis and Narayan 2013] which is suggested in N3710 [Boehm et al. 2013] is to require that $hb \cup rf$ be acyclic. This approach eliminates some of the interesting behaviors allowed by real implementations, and thus we adopt a more complicated and weaker set of

constraints. Our constraints are not intended to be a final solution to the “out-of-thin-air” problem.

To focus the proof on the core challenges, we next define a core subset of the C/C++ features.

11.1.1. Simplified C/C++11 Model. For the purposes of simplicity and focusing on the core technical issues, we will consider the following simplified version of the C/C++ memory model:

- Atomics only perform a load or a store. We will not consider fence or read-modify-write operations.
- We omit `memory_order_consume` and `memory_order_seq_cst`. We support only `memory_order_release`, `memory_order_acquire`, and `memory_order_relaxed`. One implication is that *hb* is the transitive closure of *sb* and *sw*.
- We also remove release sequences as a consequence of removing read-modify-write operations.⁷ Synchronization occurs only as follows:

For store-release *x* and load-acquire *y*, if $x \xrightarrow{rf} y$, then $x \xrightarrow{sw} y$.

- We drop the now-removed requirements of C++11 §29.3p9 and replace them with our own *satisfaction order* that we define in Section 11.1.2.
- We assume that programs exist as a static set of *n* threads t_1, t_2, \dots, t_n , all of which happen after some set of atomic initialization operations.
- We omit mutexes and other high-level synchronization primitives.
- There is a modification order (*mo*) that totally orders memory operations to a given memory location. This order is consistent with happens before.

Although read-modify-write, fence, mutex, and `memory_order_seq_cst` operations along with release sequences are all supported by CDSCHECKER, we elide the details in our proof as they either complicate the proof with little benefit (e.g., fences) or are handled orthogonally to the core of this proof (e.g., mutexes—via standard dynamic partial-order reduction techniques).

11.1.2. Satisfaction Order. C++11 previously defined an *evaluation sequence* (§29.3p9) which generates the value written by a store, to try to eliminate “out-of-thin-air” values. However, the authors of this article developed a variation of IRIW that showed existing architectures violate §29.3p9; as a result, N3786 removes the evaluation sequence from the standard and replaces it with an informal statement.

Therefore, we next make precise a set of constraints that implement C/C++14’s informal prohibition against “out-of-thin-air” values. Appendix A provides the set of satisfaction cycle examples that motivated our definition of dependence order.

Definition 11.1. The *dependence order* (*dep*) is the minimal relation that includes all pairs of evaluations (*a*, *b*) where any one of the following is true:

- (1) *b* is data dependent on *a* (excluding reads-from (*rf*))
- (2) There exists some potentially nonterminating loop *l* such that *l* is conditionally dependent on *a* and $l \xrightarrow{sb} b$
- (3) *a* is a load-acquire, *b* is a load or store, and $a \xrightarrow{sb} b$ ⁸

⁷CDSCHECKER handles the other complications of release sequences (that the head store could potentially be relaxed) by simply assuming such release sequences do not synchronize and then checking whether they must have synchronized after the entire execution is generated.

⁸This is required because acquire operations can “leak” information about synchronization (or lack thereof) to subsequent operations.

- (4) a is a load, b is a store, and there exists a condition statement (e.g., if statement) a_0 such that a_0 depends on a and $a_0 \xrightarrow{sb} b$
- (5) a is a load, b is a store, and there exists a store a_0 such that the address of a_0 depends on a and $a_0 \xrightarrow{sb} b$

Definition 11.2. Let sat be a relation that is the transitive closure of dep extended with the edges from the reads-from (rf) relation. Thus, sat is the transitive closure of $dep \cup rf$.

To forbid satisfaction cycles we add the axiom that sat must be antisymmetric:

$$a \xrightarrow{sat} b \implies \neg b \xrightarrow{sat} a$$

Transitivity and antisymmetry together imply acyclicity of the satisfaction order, its central property.

To illustrate the satisfaction order, consider the following example, in which C++11 standard discourages allowing the result $r1 = r2 = 42$. Our requirements (rules 1 and 4, plus acyclicity) disallow such a satisfaction cycle.

```
// Thread 1                                // Thread 2
r1 = x.load(memory_order_relaxed);          r2 = y.load(memory_order_relaxed);
if (r1 == 42)                                if (r2 == 42)
    y.store(r1, memory_order_relaxed);      x.store(42, memory_order_relaxed);
```

At the same time that the acyclicity of our satisfaction order outlaws problematic satisfaction cycles, it removes the onerous restrictions that the former §29.3p9 placed on evaluations, where an evaluation must observe values written by the “last” prior assignment in the evaluation order.

11.1.3. Load-Load Dependencies. Because it is acyclic, the satisfaction order can be topologically sorted such that any evaluation depends only on those evaluations preceding it in the sequence, and the root evaluation depends only on constants or program inputs. However, the satisfaction order as defined does not yet account for all types of dependence. For instance, consider the following program fragment:

```
a: r1 = x.load(memory_order_relaxed);
b: if (r1 == 1)
c:   r2 = y.load(memory_order_relaxed);
```

and consider a target execution in which $r1 = r2 = 1$. In such an execution, we want to capture the information flow from statement a to statement c via the conditional branch at b because we cannot guarantee that CDSCHECKER will explore statement c until it satisfies all dependencies for a (that is, until we see $r1 = 1$). But the definition of sat does not specify that $a \xrightarrow{sat} c$ —it only requires this if c is a store.

We observe, however, a secondary property of the satisfaction order, which we present in the following lemma.

LEMMA 11.3 (CONDITIONAL LOAD ORDERING). *Let U be an execution of a program. For all loads $a, c \in U$ and condition statement b such that b depends on a or a store b whose address depends on a and $b \xrightarrow{sb} c$, then*

$$\neg c \xrightarrow{sat} a.$$

PROOF. Assume that $c \xrightarrow{sat} a$. Then there must be some store d such that $c \xrightarrow{sb} d$ and $c \xrightarrow{sat} d \xrightarrow{sat} a$. But because $a \xrightarrow{sb} b \xrightarrow{sb} d$, then $a \xrightarrow{sat} d$ —a cycle. Therefore, $\neg c \xrightarrow{sat} a$. \square

As a corollary, whenever such a pair of loads a and c exist that satisfy the hypothesis of this lemma, we can safely extend the satisfaction order to include $a \xrightarrow{sat} c$ (and its transitive closure). From now on, consider that the satisfaction order includes these conditional load-load dependencies.

11.2. CDSCHECKER Algorithm

We next review key points of the CDSCHECKER algorithm with respect to simplified C/C++ memory model used for this proof.

Execution Order. We define the *execution order* according to the sequence of scheduling choices made in Figure 3, Line 8. If an operation x is executed before an operation y , then we say $x \xrightarrow{exec} y$.

Coherence. When we refer to COHERENCE, we refer to the following four rules, for loads x, y and stores a, b , all to the same location:

$$\begin{aligned}
 a \xrightarrow{hb} b &\implies a \xrightarrow{mo} b && \text{(WRITE-WRITE COHERENCE)} \\
 a \xrightarrow{hb} x \wedge b \xrightarrow{rf} x &\implies a \xrightarrow{mo} b && \text{(WRITE-READ COHERENCE)} \\
 x \xrightarrow{hb} b \wedge a \xrightarrow{rf} x &\implies a \xrightarrow{mo} b && \text{(READ-WRITE COHERENCE)} \\
 x \xrightarrow{hb} y \wedge a \xrightarrow{rf} x \wedge b \xrightarrow{rf} y &\implies a \xrightarrow{mo} b && \text{(READ-READ COHERENCE)}
 \end{aligned}$$

Early Termination. CDSCHECKER ensures that at all points in an execution, mo is acyclic and sw is consistent with $exec$. If at any point either of these conditions is violated, CDSCHECKER discards the execution immediately as invalid. Note that the acyclicity of hb allows the latter optimization.

Reading from the Past. For a load x , we define $stores(x)$ to be the set of all stores in the program execution which store to the same location as x .

The set $may-read-from(x)$ is built up as a subset of stores $u \in stores(x)$, such that $u \xrightarrow{rf} x$ would not create a mo -cycle. When x is first explored, we consider prior stores in the execution order. When new stores occur after x , we consider passing their value as *future values*.

Note that the mo -cycle requirement eliminates the following set, for instance,

$$\{u \in stores(x) \mid \exists a \in stores(x). u \xrightarrow{hb} a \xrightarrow{hb} x\}$$

because

$$\begin{aligned}
 u \xrightarrow{hb} a &\implies u \xrightarrow{mo} a && \text{(WRITE-WRITE COHERENCE)} \\
 a \xrightarrow{hb} x \wedge u \xrightarrow{rf} x &\implies a \xrightarrow{mo} u. && \text{(WRITE-READ COHERENCE)}
 \end{aligned}$$

Intuitively, x should not read from any store in this set because each such u is masked by the later store a .

Reading from Future Stores. In general, the C/C++ memory model allows loads to read from stores that appear later in the trace. CDSCHECKER uses a future value mechanism to simulate this behavior. The idea is to learn the value that a future store writes in one execution and then use backtracking plus speculation to allow a load to read from a future store. When a load reads from a future store, CDSCHECKER generates a promise or assertion that some later store will write that value. There is at most one

promise for a given store, and once a promise is instantiated other loads can read from that same promise.

The `PROCESSACTION` method in Figure 3 encapsulates the process of sending future values from stores to prior loads in the execution order. For any load x and store y to the same location, where $x \xrightarrow{\text{exec}} y$ and y is the last operation in S , `PROCESSACTION(S)` will add $(\text{value}(y), \text{thread}(y))$ to $\text{futurevalues}(x)$ unless one of the following are true:

- (1) $x \xrightarrow{hb} y$
- (2) $\exists b. x \xrightarrow{hb} b \xrightarrow{mo} y$
- (3) $\exists a, b. x \xrightarrow{hb} b \wedge a \xrightarrow{rf} b \wedge a \xrightarrow{mo} y$
- (4) There exists a promise p such that $x \xrightarrow{\text{exec}} p \xrightarrow{\text{exec}} y$ (a.k.a. “promises may allow”).

The first three rules are derived from `COHERENCE`, and the fourth is an optimization which reduces the exploration of cyclic behaviors. Because it is not central to `CDSCHECKER`, we first prove correctness without this rule and then argue correctness in the presence of this optimization. See Corollary 11.17.

Behaviors. The *behaviors* set is different for loads and stores. For a load, it represents the *may-read-from* set, which consists of

- prior stores (in the execution order) and
- future stores—because these have not yet been executed in the current execution, they remain “speculative” and must be treated differently until we produce a store that satisfies the promised value.

Stores do not select a *rf* behavior like loads do, so they have a single operational behavior. They are, however, given the choice of whether to satisfy a given promise—recall that for any store which may be read by a prior load, the store must explore both the scenario in which it satisfies the prior load and the scenario in which it does not satisfy the load, potentially allowing a subsequent store to satisfy it. Thus, a store’s *behaviors* consist of only a promise satisfaction behavior, with one behavior for each potential promise to satisfy and one behavior for the scenario in which it satisfies no promises.

11.3. Proof Overview

Before continuing, we next discuss our overall proof strategy. The first issue is that the C/C++ memory model does not contain a notion of a trace order (or execution order) and a given execution can be generated by a number of different traces of the `CDSCHECKER` algorithm.

Thus we begin in Section 11.4 by building up a notion of equivalence between subsets of actions of different traces. Given this definition of equivalence, in Section 11.5 we show a number of properties that hold for equivalent traces prefixes.

In Section 11.6, we then build the framework for mapping stores in one trace prefix to stores in an equivalent trace prefix. We then formally define a bijection between subsequences of stores from equivalent subtraces and show that this bijection preserves a number of properties including happens before.

In Section 11.7, we then define the notion of a consistent subtrace. Finally, in Section 11.8 we use an induction on consistent subtraces to show the correctness of the `CDSCHECKER` algorithm.

11.4. Trace Equivalence

We want to show that for a target execution trace V that is consistent with the memory model, CDSHECKER will explore some execution that is equivalent. In the following definitions, we build on our definition of satisfaction order to develop the formal infrastructure for representing trace equivalence.

Now, we observe that the *sat* and *dep* relations partially order the operations in a program execution. Particularly, they capture an acyclic flow of information in the program, such that two operations, where one “causes” another, are related by *dep*.

Next, we will begin to compare two execution traces U and V , constructively building up a set of operations from U and V with matching thread sequencing and *dep* structure. Given an initial “matching” set X , we can pair up a sequencing of matching operations which depend only on operations in X —only their *rf* structure is missing.

Definition 11.4. Let U and V be execution traces of the same program and input, with threads t_1, t_2, \dots, t_n , and let $X \in \mathcal{P}(U \times V)$. We can index the evaluations in each thread according to sequenced-before⁹ where $d_{U,i,j}$ is the j th evaluation in thread t_i of trace U . We define $e_{U,i,j}$ to be the subsequence of $d_{U,i,j}$ which is restricted to the subset

$$\{d_{U,i,j} \mid \forall u \xrightarrow{dep} d_{U,i,j}. \exists (u, v) \in X\}.$$

We define $e_{V,i,j}$ similarly.

Having defined these sequences, we next define equivalence.

Definition 11.5. Given two traces U and V of a program with n threads t_1, t_2, \dots, t_n and a set of initial matches $M \in U \times V$ from future values in U to stores in V such that a future value u is only mapped to a store v if in a previous execution the store u' that sent the future value u was equivalent to v . $g_{equiv} : \mathcal{P}(U \times V) \rightarrow \mathcal{P}(U \times V)$ is defined as follows:

- (1) Let $X \in \mathcal{P}(U \times V)$
- (2) Define $e_{U,i,j}$ and $e_{V,i,j}$ as in Definition 11.4.
- (3) Let

$$\begin{aligned} N_i = \{ & (x, y) \mid \exists j \in \mathbb{N}. x = e_{U,i,j} \wedge y = e_{V,i,j} \wedge \\ & (\forall a \in U. a \xrightarrow{dep} x \implies \exists b \in V. (a, b) \in X \wedge b \xrightarrow{dep} y) \wedge \\ & (\forall b \in V. b \xrightarrow{dep} y \implies \exists a \in U. (a, b) \in X \wedge a \xrightarrow{dep} x) \wedge \\ & (\forall a \in U. a \xrightarrow{rf} x \implies \exists b \in V. (a, b) \in X \wedge b \xrightarrow{rf} y) \wedge \\ & (\forall b \in V. b \xrightarrow{rf} y \implies \exists a \in U. (a, b) \in X \wedge a \xrightarrow{rf} x) \} \end{aligned}$$

- (4) Let

$$g_{equiv}(X) = X \cup \bigcup_{i=1}^n N_i.$$

Then g_{equiv} is a fixed-point generating function, which can be applied iteratively, starting with the set of initial matches $X = M$. Note that M does not in general monotonically increase as we progress towards the target execution. We make use of the set M in the proof of Theorem 11.14.

⁹We assume *sb* to be a total order on events from the same thread. This assumption arises because CDSHECKER is architected as a library and thus does not have sufficient information available to determine when operations from one thread are not ordered by *sb*.

The fixed point X_0 of g_{equiv} (that is, where $g_{equiv}(X_0) = X_0$) is a set of evaluation pairs $(x, y) \in U \times V$, where we say x is *equivalent* to y if $(x, y) \in X_0$. We denote this $x \cong y$.

To make these definitions clearer, consider the following example code:

```

1  atomic<int> x(0), y(0), z(0);
2
3  void threadA() {
4      int r1 = y.load(memory_order_relaxed);
5      int r2 = z.load(memory_order_relaxed);
6      if (r2 != 0)
7          r2 = z.load(memory_order_relaxed);
8      int r3 = 0;
9      if (r1 != 0)
10         r3 = x.load(memory_order_relaxed); /* \label{line:loadx} @*/
11 }
12 void threadB() {
13     x.store(1, memory_order_relaxed);
14     y.store(2, memory_order_relaxed);
15     z.store(3, memory_order_relaxed);
16 }
```

Let's consider the following two executions:

$$\begin{aligned}
 U = & y.\text{load}(\text{memory_order_relaxed})=1^a, z.\text{load}(\text{memory_order_relaxed})=0^b, \\
 & x.\text{load}(\text{memory_order_relaxed})=0^c, x.\text{store}(1, \text{memory_order_relaxed})^d, \\
 & y.\text{store}(2, \text{memory_order_relaxed})^e, z.\text{store}(3, \text{memory_order_relaxed})^f
 \end{aligned}$$

where $e \xrightarrow{rf} a$, initial value $\xrightarrow{rf} b$, and initial value $\xrightarrow{rf} c$.

$$\begin{aligned}
 V = & y.\text{load}(\text{memory_order_relaxed})=1^A, z.\text{load}(\text{memory_order_relaxed})=3^B, \\
 & x.\text{store}(1, \text{memory_order_relaxed})^C, y.\text{store}(2, \text{memory_order_relaxed})^D, \\
 & z.\text{load}(\text{memory_order_relaxed})=3^E, x.\text{load}(\text{memory_order_relaxed})=2^F, \\
 & z.\text{store}(3, \text{memory_order_relaxed})^G
 \end{aligned}$$

where $D \xrightarrow{rf} A$, $G \xrightarrow{rf} B$, $G \xrightarrow{rf} E$, and $C \xrightarrow{rf} F$.

For these executions,

$$\begin{aligned}
 d_{U,1} = & y.\text{load}(\text{memory_order_relaxed})=1^a, z.\text{load}(\text{memory_order_relaxed})=0^b, \\
 & x.\text{load}(\text{memory_order_relaxed})=0^c \\
 d_{U,2} = & x.\text{store}(1, \text{memory_order_relaxed})^d, y.\text{store}(2, \text{memory_order_relaxed})^e, \\
 & z.\text{store}(3, \text{memory_order_relaxed})^f \\
 d_{V,1} = & y.\text{load}(\text{memory_order_relaxed})=1^A, z.\text{load}(\text{memory_order_relaxed})=3^B, \\
 & z.\text{load}(\text{memory_order_relaxed})=3^E, x.\text{load}(\text{memory_order_relaxed})=2^F, \\
 d_{V,2} = & x.\text{store}(1, \text{memory_order_relaxed})^C, y.\text{store}(2, \text{memory_order_relaxed})^D, \\
 & z.\text{store}(3, \text{memory_order_relaxed})^G
 \end{aligned}$$

Starting with the matching set $M = \{\}$ as U has no unresolved future values, the fixed-point of the generating function for this example will yield $X_0 = \{\langle a, A \rangle, \langle d, C \rangle, \langle e, D \rangle, \langle f, G \rangle\}$. Alternatively, $a \cong A$, $d \cong C$, $e \cong D$, $f \cong G$.

Using this X_0 , we obtain:

$$\begin{aligned}
 e_{U,1} &= y.\text{load}(\text{memory_order_relaxed})=1^a, z.\text{load}(\text{memory_order_relaxed})=0^b, \\
 &\quad x.\text{load}(\text{memory_order_relaxed})=0^c \\
 e_{U,2} &= x.\text{store}(1, \text{memory_order_relaxed})^d, y.\text{store}(2, \text{memory_order_relaxed})^e, \\
 &\quad z.\text{store}(3, \text{memory_order_relaxed})^f \\
 e_{V,1} &= y.\text{load}(\text{memory_order_relaxed})=1^A, z.\text{load}(\text{memory_order_relaxed})=3^B, \\
 &\quad x.\text{load}(\text{memory_order_relaxed})=2^F, \\
 e_{V,2} &= x.\text{store}(1, \text{memory_order_relaxed})^C, y.\text{store}(2, \text{memory_order_relaxed})^D, \\
 &\quad z.\text{store}(3, \text{memory_order_relaxed})^G
 \end{aligned}$$

To make notation easier throughout this proof, we will use the following additional notations for referring to the matching prefix of traces U and V .

Definition 11.6. Given execution traces U and V , let

$$\begin{aligned}
 PRE(U, V) &= \{(x, y) \in U \times V \mid x \cong y\} \\
 P_U &= PRE_1(U, V) = \{x \mid \exists (x, y) \in PRE(U, V)\} \\
 P_V &= PRE_2(U, V) = \{y \mid \exists (x, y) \in PRE(U, V)\}
 \end{aligned}$$

We call $PRE(U, V)$ the *prefix* of U and V (PRE_1 and PRE_2 are the left- and right-hand projections). We make use of notation P_U and P_V throughout the rest of the article for brevity. Intuitively, the prefix of U and V is a matching portion at the “beginning” of U and V such that no evaluations in the prefix may depend on any evaluation outside of the prefix.

We say that traces U and V are equivalent (denoted $U \cong V$) if

$$PRE_1(U, V) = U \wedge PRE_2(U, V) = V.$$

For ease of use, we define a bijection $equiv : P_U \rightarrow P_V$ where $equiv(u) = v \iff u \cong v$.

In addition to equivalence (\cong), it is useful to recognize those operations which are equivalent in everything except for their *rf* behavior. We define a similarity property for this purpose.

Definition 11.7. Given traces U and V , let us also define a similarity operator (\sim) using a mapping $SIM : U \times V \rightarrow U \times V$

- (1) Let $X = PRE(U, V)$, and define $e_{U,i,j}$ and $e_{V,i,j}$ as in Definition 11.4.
- (2) Let

$$\begin{aligned}
 O_i &= \{(x, y) \mid \exists j \in \mathbb{N}. x = e_{U,i,j} \wedge y = e_{V,i,j} \\
 &\quad (\forall a \in U. a \xrightarrow{dep} x \implies \exists b \in V. (a, b) \in X \wedge b \xrightarrow{dep} y) \wedge \\
 &\quad (\forall b \in V. b \xrightarrow{dep} y \implies \exists a \in U. (a, b) \in X \wedge a \xrightarrow{dep} x)\}.
 \end{aligned}$$

- (3) Let

$$SIM(U, V) = X \cup \bigcup_{i=1}^n O_i.$$

Then we say that $x \in U$ is *similar* to $y \in V$ if $(x, y) \in SIM(U, V)$. We denote similarity as $x \sim y$. We define the projections $Q_U = SIM_1$ and $Q_V = SIM_2$ as before.

For our example, $a \sim A, d \sim C, e \sim D, f \sim G, c \sim F$. Note the addition of $c \sim F$ relative to the equivalence relation. The load c and load F both have all of their dependencies in PRE , the just happen to read from different stores.

Intuitively, similarity includes evaluations that, aside from their selection of rf , are equivalent.

For ease of use, we define a bijection $sim : \mathcal{Q}_U \rightarrow \mathcal{Q}_V$ where $sim(u) = v \iff u \sim v$.

11.5. Trace Properties

We next build on the notion of equivalence to prove properties that hold for equivalent prefixes. First, our definition of equivalence yields the following properties, where the third statement is of particular noteworthiness.

LEMMA 11.8 (PREFIX EQUIVALENCE). *Let U and V be execution traces, and let $u_1, u_2, u_3 \in U, v_1, v_2, v_3 \in V$ where $u_1 \sim v_1, u_2 \cong v_2$, and $u_3 \sim v_3$. Then*

- (1) $u_1 \xrightarrow{rf} u_2 \iff v_1 \xrightarrow{rf} v_2$
- (2) $u_1 \xrightarrow{sw} u_2 \iff v_1 \xrightarrow{sw} v_2$
- (3) $u_1 \xrightarrow{hb} u_2 \implies v_1 \xrightarrow{hb} v_2$
- (4) $\exists u' \in U. u_1 \xrightarrow{hb} u' \xrightarrow{sb} u_3 \implies v_1 \xrightarrow{hb} v_3$

PROOF. *Statement 1:* Follows from the definition of equivalence.

Statement 2: Follows from Statement 1 and the definition of sw , since $u_1 \xrightarrow{sw} u_2 \implies u_1 \xrightarrow{rf} u_2$.

Statement 3: Assume $u_1 \xrightarrow{hb} u_2$.

It follows trivially that $u_1 \xrightarrow{sb} u_2 \iff v_1 \xrightarrow{sb} v_2$ (and sb implies hb). Additionally, $u_1 \xrightarrow{sw} u_2$ is already covered by Statement 2.

Assume instead that u_1 and u_2 are in different threads and that $\neg u_1 \xrightarrow{sw} u_2$. Then there exists one or more chains of sb and sw which transitively construct this hb such that none of the stores are future values. That is, for some $n \in \mathbb{N}$, $\exists a_i, b_i. i = 1, 2, \dots, n$ such that

$$u_1 \xrightarrow{sb} a_1 \xrightarrow{sw} b_1 \xrightarrow{sb} a_2 \xrightarrow{sw} b_2 \xrightarrow{sb} \dots \xrightarrow{sb} a_n \xrightarrow{sw} b_n \xrightarrow{sb} u_2.$$

Now b_i must be an acquire operation (with $b_i \xrightarrow{sb} a_{i+1}$ for $i = 1, 2, \dots, (n-1)$ and $b_n \xrightarrow{sb} u_2$), and $a_i \xrightarrow{rf} b_i$; hence, we can inductively show that $a_i \xrightarrow{sat} b_i \xrightarrow{sat} u_2$ for all $i = 1, 2, \dots, n$. Because $u_2 \in PRE_1(U, V)$, then $a_i, b_i \in PRE_1(U, V)$. Thus, for any such chain in U , there exists an equivalent chain in V such that $v_1 \xrightarrow{hb} v_2$.

Statement 4: Follows from same argument used in statement 3 but extended with the observation that since last load acquire is dep ordered before u_3 and $u_3 \in \mathcal{Q}_U$ that the last load acquire must be in P_U . \square

Our proof will focus on the behavior of loads; as long as all the dependencies for a store are in the equivalence prefix, the store itself must be in the prefix.

11.6. Bijection between Stores in Trace Prefixes

The primary challenge at this point is that loads cannot read from arbitrary stores in C/C++11 — the C/C++11 memory model restricts the values loads can observe with a number of constraints involving both the modification order and the happens-before relations. These constraints could potentially prevent the model checker from making progress towards the target execution. Our approach towards this problem is to show

that we can always generate an execution prefix for which both the happens-before and modification order constraints are weaker than those in target execution.

The first step towards showing this is to create a bijection between the stores in the equivalent prefixes of two traces and show that this bijection preserves a number of properties.

Before we can do this, we have to first address the problem that the stores in our definition of Q_U or Q_V may not be contiguous and may contain future values. We thus define the store subprefixes S_U and S_V to contain all of the stores in P_U and P_V , respectively, plus all stores that are sequenced before any load in SIM or a store in PRE :

Definition 11.9 (Store Subprefix). Let U, V be execution traces with prefixes Q_U and Q_V .

We define

$$S_U = \{u \in U \mid is_store(u) \wedge \exists x \in Q_U. \neg is_fut_val(x) \wedge u \xrightarrow{sb} x\} \cup P_U$$

and

$$S_V = \{v \in V \mid is_store(v) \wedge \exists x \in Q_U. \neg is_fut_val(x) \wedge v \xrightarrow{sb} equiv(x)\} \cup P_V.$$

Additionally, for any thread t_i in trace U we define $s_{U,i}$ as the subsequence of stores in S_U performed by thread t_i , ordered according to sequenced-before with future values ordered according to the sequenced-before relation of their equivalents in V .¹⁰ The notation $s_{U,i,j}$ refers to the j^{th} store in the sequence $s_{U,i}$. For any thread t_i in trace V , define $s_{V,i}$ as the subsequence of stores in S_V performed by thread t_i .

Having defined store subprefixes, we next establish a bijection between them.

LEMMA 11.10 (STORE BIJECTION). *For each thread t_i , let $s_{U,i,j}$ and $s_{V,i,j}$ be from Definition 11.9.*

Then the function $f(s_{U,i,j}) = s_{V,i,j}$ that maps a store $s_{U,i,j}$ from S_U to the equivalently numbered store $s_{V,i,j}$ from S_V is a bijection.

PROOF. To show that f is a bijection, we need to show that it is (1) a function, (2) onto, and (3) one-to-one.

We begin by showing that f is a function. If a store $s_{U,i,j}$ is a member of S_U , then we know that it is sequenced before some action x that is in Q_U . By Lemma 11.3 we know that any load that determines whether $s_{U,i,j}$ is executed is satisfaction ordered before x . Thus, if $s_{U,i,j}$ is in U , there must be an analogous store in $S_{V,i}$ and, furthermore, by Lemma 11.3 that store must write to the same address.

By the same argument, applied to stores in S_V , we can show that f is onto.

All that remains is to show that f is one to one, and is easy to see that it is by construction. \square

This lemma proves, then, that we can bijectively map stores from S_U to stores in S_V —we no longer have to reason about stores which may appear in U but not in V based on unrelated execution choices, for instance.

The next lemma shows that the bijection f preserves equivalence in thread, location, sequenced-before, and happens-before.

LEMMA 11.11. *Let U, V be execution traces. Let S_U, S_V , and f be from Definitions 11.9 and Lemma 11.10.*

¹⁰Future values in U are not inherently ordered as we have lost the information about the stores that produced them. We can instead use the ordering of the corresponding stores in V to ensure that they match up properly in the bijection that we build later.

Then for all stores $a, b \in S_U$

- (1) a and $f(a)$ are in the same thread
- (2) a and $f(a)$ store to the same location
- (3) $a \xrightarrow{sb} b \iff f(a) \xrightarrow{sb} f(b)$
- (4) $a \xrightarrow{hb} b \implies f(a) \xrightarrow{hb} f(b)$.

Additionally, $\forall u \in P_U. \text{equiv}(u) = f(u)$.

PROOF. Statements 1 and 3 follow directly from the definition of f .

Statement 2: For each thread t_i , let $s_{U,i,j}$ and $s_{V,i,j}$ be from Definition 11.9.

Suppose $s_{U,i,j}$ and $s_{V,i,j}$ do not store to the same location. This means there exists a load $a \in U$ and $b \in V$ on which the address of $s_{U,i,j}$ and $s_{V,i,j}$ depends, respectively.

By the definition of S_U , there exists an operation x such that $x \in Q_U$ and $s_{U,i,j} \xrightarrow{sb} x$.

By Lemma 11.3 we also have $a \xrightarrow{sat} x$. Since $x \in Q_U$ all of its dependencies in the satisfaction order other than reads-from must be resolved and hence a must be in P_U . But once $a \in P_U$, the analogous b must be in P_V and read the same value—a contradiction.

Thus, u and v store to the same location.

The converse arguments apply similarly.

Statement 4: Assume that $a \xrightarrow{hb} b$. Since $b \in S_U$, there must either exist $u \in Q_U. b \xrightarrow{sb} u$ (by definition of S_U) or $b \in P_U$ and let $u = b$.

Consider $a \xrightarrow{hb} b$ in cases. Because they are both stores, we can split cases as follows:

- (1) $a \xrightarrow{sb} b$
- (2) $\exists a_i, b_i. a = a_0 \wedge a_0 \xrightarrow{sw} b_0 \xrightarrow{sb} a_1 \xrightarrow{sw} \dots \xrightarrow{sw} b_n \xrightarrow{sb} b$
- (3) $\exists a_i, b_i. a \xrightarrow{sb} a_0 \xrightarrow{sw} b_1 \xrightarrow{sb} a_1 \xrightarrow{sw} \dots \xrightarrow{sw} b_n \xrightarrow{sb} b$

Case 1: If $a \xrightarrow{sb} b$, then by Statement 3, $f(a) \xrightarrow{sb} f(b)$.

Case 2: For all i , a_i is a store, b_i is a load-acquire, and $a_i \xrightarrow{rf} b_i$, so $a_i \xrightarrow{sat} b_i \xrightarrow{sat} a_{i+1}$. Additionally, $b_n \xrightarrow{sat} u \in Q_U$, so all $a_i, b_i \in P_U$ by the definition of the satisfaction order. Thus, $a_0 \cong f(a_0)$, $b_n \cong f(b_n)$, and by Lemma 11.8, $f(a_0) \xrightarrow{hb} f(b_n) \xrightarrow{sb} f(b)$.

Case 3: In the same way as Case 2, we can show that $f(a_0) \xrightarrow{hb} f(b)$. We then apply Statement 3 to show that $f(a) \xrightarrow{sb} f(a_0) \xrightarrow{hb} f(b)$. \square

11.7. Consistent Subtraces

We are now ready to reason about the exploration of a series of traces in order to form executions with increasingly larger matching prefixes. But first, we define a subtrace, so we can present some properties about subportions of a trace. There are two concerns that motivate our definition of a consistent subtrace: (1) each subtrace has to satisfy our modification order constraints such that the trace is not terminated early by CDSHECKER, (2) each subtrace should contain minimal constraints on the reads-from relation such that CDSHECKER can take the next step towards the target trace, and (3) the definition should contain enough freedom to allow loads to read from future values as necessary for backtracking to build back up a trace.

Definition 11.12. Let U be an execution trace and let $x \in U$.

A *subtrace* U_x of U is a subset of operations in U such that

$$\forall u \in U. u \xrightarrow{exec} x \vee x = u \implies u \in U_x$$

and

$$\forall u \in U, v \in U_x. u \xrightarrow{rf} v \implies u \in U_x.$$

A subtrace intuitively includes (1) the subsequence of the execution order up to and including x and (2) any promised future stores from which this subsequence reads.

When constructing execution traces U' , we want to constrain all loads in an execution trace to reading only from stores which we can show are consistent with the target trace V . That is, we want to ensure that we don't generate *mo*-cycles and that the modification order does not unnecessarily constrain the stores from which later reads may read. The bijection f provides much of the information required for selecting stores from which a given load should read.

We now define the consistency property that we want to maintain as we build up increasingly large subtraces.

Definition 11.13 (Consistent subtrace). Given traces U and V and a subtrace U_x of U , we define

$$\begin{aligned} & \text{consistent_subtrace}(U, V, x) \\ &= (\forall a, b \in U_x. (a \xrightarrow{rf} b \wedge b \notin P_U) \implies (\exists c. a \xrightarrow{hb} c \xrightarrow{sb} b \vee a \xrightarrow{rf} c \xrightarrow{hb} b)) \\ &\wedge (\forall a, b \in U_x \cap S_U. a \xrightarrow{mo} b \implies f(a) \xrightarrow{mo} f(b)) \\ &\wedge (\forall a \in U_x \setminus S_U, b \in U_x \cap S_U. \neg a \xrightarrow{mo} b) \end{aligned}$$

The *consistent_subtrace* predicate asserts that a subtrace of U has a modification order which is consistent with V and provides as few additional constraints on the modification order as possible. The predicate consists of the logical conjunction (\wedge) of three separate clauses. The first clause provides restrictions for loads that are not in the prefix, and the latter two clauses restrict the modification order of stores that are and are not (respectively) in the bijective mapping f .

In the first clause, we prevent nonprefix loads from establishing additional modification order constraints by reading from a previously unconstrained store.

The second clause requires that stores in the bijective mapping f have a modification order that is compatible with their match in V .

The third clause forbids stores that are not in the bijective mapping f from being ordered before any store in the bijective mapping. Stores that are not in the bijective mapping can obstruct our intended *rf* relations in arbitrary ways unless we can constrain them in this way.

Given this consistency predicate, we can construct an execution U' using progressively larger subtraces U_x that satisfy *consistent_subtrace*(U', V, x) until we have $U' = U_x$. We prove this in the following theorem.

THEOREM 11.14. *Let U and V be traces with a subtrace U_x of U .*

If $\text{consistent_subtrace}(U, V, x)$ is true, then either $U = U_x$ or CDSCHECKER can generate an execution U' with subtrace U'_y such that

$$\begin{aligned} & \text{consistent_subtrace}(U', V, y) \wedge U_x \subset U'_y \wedge \\ & (\text{PRE}_1(U, V) \cap U_x) \subseteq (\text{PRE}_1(U', V) \cap U'_y) \end{aligned}$$

PROOF. If $U = U_x$, then we are done.

Assume instead that $U \neq U_x$, and so there exists a load $y \in U \setminus U_x$. If we select the earliest y (in the execution order), then we can construct U' as follows.

If $y \in P_U$, then we note that by definition of P_U that y reads from u such that $f(u) \xrightarrow{rf} f(y)$. Thus the execution already trivially satisfies *consistent_subtrace* and retains the prefix such that $(PRE_1(U, V) \cap U_x) \subseteq (PRE_1(U', V) \cap U'_y)$.

Now, assume $y \notin P_U$.

Let $stores_hb(y) = \{u \in U_x \mid is_store(u) \wedge same_location(u, y) \wedge (\exists c. u \xrightarrow{hb} c \xrightarrow{sb} y \vee u \xrightarrow{rf} c \xrightarrow{hb} y)\}$. We will select $u \in stores_hb(y)$ with which to construct a new subtrace U'_y such that $u \xrightarrow{rf} y$, then we will show that *consistent_subtrace*(U', V, y). Note that U_x and U'_y differ only in load y , diminishing our proof obligations. In cases, we have:

- (1) $stores_hb(y) \subseteq S_U$
- (2) $stores_hb(y) \not\subseteq S_U$

Case 1: Assume $stores_hb(y) \subseteq S_U$. We simply select $u \in stores_hb(y)$ such that $\forall a \in stores_hb(y). u = a \vee f(a) \xrightarrow{mo} f(u)$. We then let y read from u in our new trace U' , with subtrace U'_y .

The first clause of *consistent_subtrace* is trivially satisfied because $u \in stores_hb(y)$.

For the second clause, suppose that in U_x , there exists some store a such that y introduces a new constraint $a \xrightarrow{mo} u$. The only two rules that apply for such a load are Write-Read and Read-Read coherence. By either rule, we also have $a \in stores_hb(y) \subseteq S_U$ and, therefore, $f(a) \xrightarrow{mo} f(u)$. Thus, y cannot create a cycle with a , and we retain the consistency of the second clause.

The third clause can be easily verified because for any store a where $u \xrightarrow{rf} y \implies a \xrightarrow{mo} u$, there must be a happens-before relation between a and y and thus $a \in S_U$.

Because we do not change any loads in the subtrace U_x and the future value mapping M remains the same for those loads, we trivially retain the property $(PRE_2(U, V) \cap U_x) \subseteq (PRE_2(U', V) \cap U_y)$.

Case 2: In this case, $\exists u. u \in (stores_hb(y) \setminus S_U)$. Then we select some “late” u , such that

$$\forall v. v \in stores_hb(y) \setminus S_U \implies \neg u \xrightarrow{mo} v.$$

We can show that $u \xrightarrow{rf} y$ in U' will retain *consistent_subtrace*(U', V, y).

Again, the first clause is satisfied because $u \in stores_hb(y)$.

The second and third clauses can be argued together. No store $a \in S_U$ can happen after either u or y , so y can only provide at most the constraint that $a \xrightarrow{mo} y$. This has no bearing on the second clause, and it is consistent with the third clause.

Because we do not change any loads in the subtrace U_x and the future value mapping M remains the same for those loads, we trivially retain the property $(PRE_2(U, V) \cap U_x) \subseteq (PRE_2(U', V) \cap U_y)$. \square

11.8. Induction on Subtraces

We next state the two key theorems: Theorem 11.15 shows that CDSCHECKER will always either generate a trace that is closer to the target trace or find a bug. Theorem 11.16 is an induction on Theorems 11.14 and 11.15 to show that CDSCHECKER will either produce the target execution, find a bug, or fail to terminate.

THEOREM 11.15. *Given two execution traces U and V of the same program and input, such that*

$$U \neq V \wedge \text{PRE}(U, V) \neq \emptyset,$$

then if CDSHECKER explores U , either

- (1) *it also explores some trace U' where it reports an execution bug or*
- (2) *for any load x such that $x \in Q_U$, $\neg x \in P_U$, $\exists u \in P_U$, $\text{equiv}(u) \xrightarrow{rf} \text{sim}(x)$, $\text{consistent_subtrace}(U, V, x)$ and $\text{consistent_subtrace}(U, V, u)$ it explores some trace U' that contains x' such that $x' \cong \text{sim}(x)$, $(\text{PRE}_1(U, V) \cap U_x) \cup \{x'\} \subseteq \text{PRE}_1(U', V)$, and $\text{consistent_subtrace}(U', V, x')$.*

PROOF. Let $P_U = \text{PRE}_1(U, V)$ and $P_V = \text{PRE}_2(U, V)$.

Given that $x \in Q_U$ and $x \notin P_U$, we have $y = \text{sim}(x) \notin P_V$.

We will show in cases how we can transform U into U' , where $P_U \cup \{x'\} \subseteq \text{PRE}_1(U', V)$ where $x' \cong y$.

Now let $v = \text{equiv}(u) \xrightarrow{rf} y$ ($v \in P_V$).

First, we must show that CDSHECKER will explore U' where x may read the value from u . Consider cases for u :

- (1) $u \xrightarrow{\text{exec}} x$
- (2) $x \xrightarrow{\text{exec}} u$

Case 1: $u \xrightarrow{\text{exec}} x$

In the first case, we must show that $u \in \text{may-read-from}(x)$ when we explore x .

We will show that $u \in \text{may-read-from}(x)$ by contradiction. Assume $u \notin \text{may-read-from}(x)$. This means that the addition of $u \xrightarrow{rf} x$ must create a *mo*-cycle due to one of the read COHERENCE constraints that creates an *mo* edge from some other store c to u . This means that either $c \xrightarrow{hb} x$ (Write-Read Coherence) or there is a load l that reads from c and happens before x (Read-Read Coherence).

In the case $c \xrightarrow{hb} x$, we note that $c \in S_U$ and thus the same modification order edges must exist in V and thus we have a contradiction.

In the case $l \xrightarrow{hb} x$, we consider two cases:

- (1) $l \in P_U$: This implies that $c \in P_U$ and thus the same modification order edges must exist in V and we have a contradiction.
- (2) $l \notin P_U$: In this case, by the first clause of *consistent_subtrace* we have $\exists e. c \xrightarrow{hb} e \xrightarrow{sb} l \vee c \xrightarrow{rf} e \xrightarrow{hb} l$. In either case, we either have some action e earlier in the execution order for which either this recursively applies or the store c must happens before the load. Eventually this induction must end in the store case and hence $c \xrightarrow{hb} x$ and thus c must be in S_U , the same modification order edges must exist in V , and we have a contradiction.

Once we reach a partial execution U' in which $u \xrightarrow{rf} x$, it is simple to complete the execution trace by having loads read from stores that simply remain consistent with the *mo* of V .

As we previously had *consistent_subtrace* up to the action x , all we need show is that we satisfy *consistent_subtrace* for the new action x' . As $x' \in P_U$, the first clause is trivially satisfied. Consider a store a that could violate the second clause. To violate the clause, we must have the load x produce the edge $a \xrightarrow{mo} u$ while $f(u) \xrightarrow{mo} f(a)$

(assuming without loss of generality that \xrightarrow{mo} is a total order for a given location in V). Thus by cases we either have $a \xrightarrow{hb} x$ (write-read coherence) or $l \xrightarrow{hb} x$ where $a \xrightarrow{rf} l$. By application of the *consistent_subtrace* using the same reasoning as above we have in either case that $a \xrightarrow{hb} x$. By application of Lemma 11.8 we get $f(a) \xrightarrow{mo} f(u)$ which is a contradiction and thus the store a cannot exist. Hence the second clause is satisfied.

The last clause is trivial to show as we do not have an operation after x' that could violate the condition.

We have the property $(PRE_1(U, V) \cap U_x) \cup \{x'\} \subseteq PRE_1(U', V)$ by construction.

Case 2: $x \xrightarrow{exec} u$

Because $u \in P_U$, CDSHECKER reaches u in trace U . We must show that $(value(u), thread(u)) \in futurevalues(x)$. In other words, we must demonstrate some trace (U or a similar execution) where the future value pruning rules 1 to 3 (Section 11.2) do not prevent adding u to $futurevalues(x)$.

If u cannot send a future value to x , then one of the future value pruning rules must be blocking it. We can show that rules 1 and 2 cannot block $u \xrightarrow{rf} x$.

With rule 1, we must have $x \xrightarrow{hb} u$. Then by Lemma 11.8, $y \xrightarrow{hb} v$ which contradicts $v \xrightarrow{rf} y$.

For rule 2, we must have $\exists b. x \xrightarrow{hb} b \xrightarrow{mo} u$. Consider in cases:

- (1) $b \notin S_U$.
- (2) $b \in S_U$

The first case is impossible, due to the third clause of *consistent_subtrace*.

In the second case, the second clause of *consistent_subtrace* implies $f(b) \xrightarrow{mo} f(u)$, and Lemma 11.8 implies $sim(x) \xrightarrow{hb} f(b)$. But $f(u) \xrightarrow{rf} sim(x)$, and so $f(u) \xrightarrow{mo} f(b)$ (READ-WRITE COHERENCE) yielding a *mo*-cycle in V —a contradiction.

Now rule 3 may still prevent sending a future value, so we will show how to generate an equivalent execution U'' in which x may read from u .

Consider the set of potentially blocking loads:

$$\begin{aligned} blocking(u, x) = \{ & b \in U. is_load(b) \wedge same_location(x, b) \\ & \wedge x \xrightarrow{hb} b \xrightarrow{exec} u \wedge (\exists a. a \xrightarrow{rf} b \wedge a \xrightarrow{mo} u) \}. \end{aligned}$$

Note that none of the blocking loads can be in the prefix P_U as that would imply the same modification ordering constraint in V .

We will inductively show how to transform U into a similar execution U'' where $blocking(u, x)$ is empty (if it is empty, then rule 3 no longer blocks $u \xrightarrow{rf} x$) via a finite sequence of intermediate executions $U_i, i = 0, 1, \dots, n$, where $U_0 = U$ and $U_n = U''$.

Let $last_blocking(u, x)$ be the latest (in the execution order) load in $blocking(u, x)$ (if it is nonempty) for execution U_i .

Let $b = last_blocking(u, x)$ in U_i . Now suppose $\exists c \in blocking(u, b)$. Because $x \xrightarrow{hb} b \xrightarrow{hb} c$, we also have $c \in blocking(u, x)$. But $b \xrightarrow{exec} c$, which contradicts the “last” selection of b . Therefore, $blocking(u, b)$ is empty, and b can read a future value from u . The model checker will then find a new execution with $u \xrightarrow{rf} b$ as $u \xrightarrow{sat} b$ and having the blocking loads read from u cannot generated any *mo* edges that are not already present in V .

Thus, for each U_i where $blocking(u, x)$ is nonempty, we generate the successive execution U_{i+1} in which $u \xrightarrow{rf} b$. Note that after this change to b in execution U_{i+1} , there

may be a new $b' = \text{last_blocking}(u, x)$ such that $b \xrightarrow{\text{exec}} b'$. But we will not see infinitely many such executions because we can bound the number of blocking loads b' that can appear.¹¹ Thus, in a finite number of steps we reach an execution $U_n = U'$ in which $\text{blocking}(u, x)$ is empty, at which point x may read from u .

We can complete this partial execution U' in the same way as in Case 1. As we previously had *consistent_subtrace* up to the action x , all we need show is that we satisfy *consistent_subtrace* for the new action x' . The same argument from Case 1 applies here.

We have the property $(\text{PRE}_1(U, V) \cap U_x) \cup \{x'\} \subseteq \text{PRE}_1(U', V)$ by construction. \square

THEOREM 11.16. *Given a program's execution trace V , then for any initial execution U of the same program and input, either:*

- (1) *there exists a (potentially empty) series of backtracking transformations from U by which CDSCHECKER will produce an execution trace U' such that $U' \cong V$,*
- (2) *CDSCHECKER will discover a bug in the program under test,*
- (3) *CDSCHECKER will fail to terminate.*

PROOF. Proof by induction on consistent subtraces. Theorem 11.14 states that we can always make the consistent portion of a subtrace longer while still maintaining all of the actions in the consistent part of the trace in P_U until the entire trace is consistent (or CDSCHECKER fails to terminate by generating increasingly long traces). Once the entire trace U_c is consistent, it follows that because the satisfaction order is acyclic that if $\neg U_c \cong V$, then at least one load l in U_c has all of its dependencies other than reads-from resolved.

Theorem 11.15 then shows that we can generate a new consistent subtrace in which l is in the prefix with V while still retaining all actions that are execution ordered before l in the prefix. Thus by induction, if CDSCHECKER terminates, then it must eventually obtain a trace U' in which all loads are in the prefix and thus we have $U' \cong V$. \square

COROLLARY 11.17. *The “promises may allow” future value optimization (Rule 4) can be applied to CDSCHECKER and still retain the correctness of Theorem 11.15.*

PROOF. Consider $a \xrightarrow{\text{exec}} b \xrightarrow{\text{exec}} c$, where a may read from c and b reads from the future. We do not allow c to send its value to a until b 's promise is resolved. To show that this is sound, we reason about a few cases. Consider some hypothetical d that might satisfy b . We have the following cases:

- (1) $d \xrightarrow{\text{exec}} c$
- (2) $c = d$
- (3) $c \xrightarrow{\text{exec}} d$.

In the first case, d already executed, so it no longer can satisfy outstanding promises.

In the second case ($c = d$), d will first resolve b , eliminating the outstanding promise and allowing itself to send a future value to a .

This leaves only the third case, where $c \xrightarrow{\text{exec}} d$. If d does not depend on a , then a does not affect the existence of d ; CDSCHECKER will explore d , d will satisfy b , and c can now send its future value to a . Now, instead, assume that d depends on a . We can show c does not depend on b (if it did, then we should not send c 's value to a , as this would be

¹¹Any potentially nonterminating loops in threads that affect the generation of u must be in the prefix (they are *sat*-ordered before u) and therefore consist of a bounded number of blocking loads. And potentially nonterminating loops in unrelated threads only provide a bounded number of blocking loads, since a fair schedule guarantees that there are a bounded number of steps between x and u .

a satisfaction cycle: $a \xrightarrow{\text{sat}} d \xrightarrow{\text{rf}} b \xrightarrow{\text{sat}} c \xrightarrow{\text{rf}} a$). Thus, for some execution in which b does not read d , the execution will still include c , which can then pass its future value to a . \square

11.9. Discussion

We note that the definition of the satisfaction order in many cases specifies a stronger notion of dependence than compilers necessarily preserve. For example, a compiler may break the dependence from x to y in the following code: $y=x*0;$.

In practice, CDSCHECKER is sound with respect to a weaker definition of dependence as it only observes the effects of dependencies dynamically. If the effect of a dependence between a load and a store is not observed for any of the subtraces that comprise the induction sequence, then CDSCHECKER must produce the target execution. Thus, the types of dependencies that compiler optimizations are likely to eliminate are in general not visible to CDSCHECKER, and therefore CDSCHECKER will generate any executions that breaking these dependencies may enable.

12. EVALUATION

Because C++11 is so new, there are few tools that test programs under its memory model and few benchmarks against which to run. For those tools that do exist, there are limitations either on scalability (they can only test very small programs) or on soundness (they miss a significant number of potential program behaviors). We evaluated CDSCHECKER against these tools where possible, while separately measuring CDSCHECKER's performance on real data structures. We ran our evaluations on an Ubuntu Linux 12.04 machine with an Intel Core i7 3770 CPU. We have made both our model-checker and benchmarks publicly available at <http://demskey.eecs.uci.edu/c11modelchecker.html>.

We compiled and ran our evaluations with compiler optimizations enabled (GCC's -O3 flag). However, because we implement instrumented versions of atomic operations within CDSCHECKER's (opaque) shared library, the compiler has limited ability to reorder the atomic operations in the unit tests, and so compiler optimizations performed on the program under test do not affect the correctness of model-checking. To verify this, we studied the implementation of atomic operations in GCC and clang/LLVM. Both compilers utilize library headers which we can easily substitute with CDSCHECKER's header; thus, we transform atomic operations into function calls which cannot be reordered. Additionally, a simple experiment showed no behavioral differences in our benchmarks results when using GCC to compile them with and without optimization.

12.1. Data Structure Benchmarks

For testing CDSCHECKER on real code, we have gathered five data structure implementations—a synchronization barrier, a mutual exclusion algorithm, a contention-free lock, and two different types of concurrent queues—downloaded from various publicly accessible Internet websites and a work-stealing deque taken from Lê et al. [2013]. Additionally, we ported our own implementations of the Linux kernel's reader-writer spinlock from its architecture-specific assembly implementations and the Michael and Scott queue from its original C and MIPS source code [Michael and Scott 1996].

Most benchmarks were originally written simply as data structure implementations, so we wrote test drivers for many of them in order to run them under CDSCHECKER. We briefly describe each data structure, our test methodology, and our performance results and analysis. For our performance results (Figure 11), we record the total number of times CDSCHECKER executed the test program (# Executions) and the number of

Benchmark	# Executions	# Feasible	Total Time (s)
Chase-Lev deque	748	81	0.14
SPSC queue	18	11	0.01
SPSC queue (bug free)	19	16	0.02
Barrier	10	10	0.01
Dekker critical section	19,319	2,313	3.22
MCS lock	18,035	14,017	3.61
MPMC queue	40,148	13,028	7.66
M&S queue	272	114	0.07
Linux RW lock	54,761	1,366	10.56

Fig. 11. Benchmark results.

executions whose behavior was consistent with the memory model (# Feasible). The ratio of the feasible executions to the total number of executions provides a measure of the overhead of exploring infeasible executions.

Many benchmarks have an infinite space of executions under memory systems that do not guarantee liveness, so for all our tests, we ran CDSCHECKER with a memory liveness parameter of 2 (see Section 10). For all benchmarks with nonatomic shared memory, we manually instrumented the normal memory accesses to check for data races.

Chase-Lev Deque: We took this implementation from a peer-reviewed, published C11 adaptation of the Chase-Lev deque [Lê et al. 2013]. It predominantly utilizes relaxed operations (for efficiency) while utilizing fences and release/acquire synchronization to establish ordering. While the article proves an ARM implementation correct, it does not contain a correctness proof for its C11 implementation. Our test driver for this benchmark utilizes two threads in which the thread that owns the deque pushes three work items and takes two work items while the other thread steals a work item.

Our model-checker discovered a bug in the published implementation. The bug occurs when both a steal and push operation occur concurrently and the push operation resizes the deque. The bug reveals itself as a load from a potentially uninitialized memory location. We contacted the article’s authors and they confirmed the bug in the C11 implementation.

SPSC queue: This single-producer, single-consumer queue allows concurrent access by one reader and one writer [sps]. We utilize the test driver provided along with the queue, which uses two threads—one to enqueue a single value and the other to dequeue it and verify the value.

This queue utilizes seq-cst atomics, a C++ mutex/condition variable and only a few non-seq-cst atomics, allowing CDSCHECKER to easily reduce the search space. It contained a known bug—a deadlock—which CDSCHECKER detected on its first execution, pruning the search space early and resulting in fewer executions for the buggy benchmark than for our modified bug-free version.

Barrier: This implements a synchronizing barrier [bar], where a given set of threads may wait on the barrier, only continuing when all threads have reached the barrier. The barrier should synchronize such that no memory operation occurring after the barrier may race with a memory operation placed before the barrier. The implementation is simple and contentious, as the first $n - 1$ threads will spin on a global flag, waiting for the n th thread to reach the barrier.

Our test driver utilizes two threads with a nonatomic shared memory operation executed on either side of the barrier, one in each thread.

Because the barrier is implemented with seq-cst atomic operations, it exhibits relatively few behaviors—those determined by simple thread interleavings. Under a fair schedule, this test required only seven executions.

Dekker critical section: This implements a simple critical section using Dekker’s algorithm [dek], where a pair of nonatomic data accesses are protected from concurrent data access. This benchmark successfully utilizes sequentially consistent, release, and acquire fences to establish ordering and synchronization.

Contention-free lock: This contention-free lock implements the algorithm proposed by Mellor-Crummey and Scott (known as an *MCS lock*) [Mellor-Crummey and Scott 1991; mcs]. The lock acts like a concurrent queue, where waiting threads are queued—first-in, first-out. Our test driver uses two threads, each of which alternates between reading and writing the same shared variable, releasing the lock in between operations.

As with several other benchmarks, heavy usage of non-seq-cst operations in multiple threads required exploration of a larger state space; weak loads and stores provide many more potential combinations of store/load pairs in the *rf* relation.

MPMC queue: This multiple-producer, multiple-consumer queue allows concurrent access by multiple readers and writers [mpml]. Our test driver runs two identical threads. Each thread first enqueues an item and then dequeues an item.

M&S queue: This benchmark is an adaptation of the Michael and Scott lock free queue [Michael and Scott 1996] to the C/C++ memory model. Our adaptation uses relaxed atomics when possible. Our test driver runs two identical threads. Each thread first enqueues an item and then dequeues an item.

Linux reader-writer lock: A reader-writer lock allows either multiple readers or a single writer to hold the lock at any one time—but no reader can share the lock with a writer. We ported this benchmark from a Linux kernel implementation, likely making this the most deployed example out of all our benchmarks.

To test the Linux reader-writer lock, our test driver runs two identical threads, with a single `rwlock_t` protecting a shared variable. Each thread reads the variable under a reader lock, then writes to the variable under the protection of a writer lock.

This benchmark utilizes a large number of relaxed memory operations, thoroughly testing the efficiency of our relaxed model optimizations. In fact, our naïve early implementations of future values typically took 30 or more minutes to complete, whereas the current results show an exploration time of under 11s.

Discussion: While many bugs are straightforward to find with simple test cases, others can be more difficult to find. The bug we discovered in the Chase-Lev Deque is an example of a bug that requires a test case that exercises a corner case—exposing this bug requires the test case to trigger a resize while another thread performs a concurrent steal operation.

In our experience, effective testing sometimes requires the developer to think about the corner cases in the code and build test cases that exercise these in the presence of potentially conflicting, concurrent accesses.

12.2. Litmus Tests

To help verify that CDSHECKER performs sound exploration of the memory model, we tested it against a set of litmus tests, including the tests described in Nitpicking [Blanchette et al. 2011] as well as a few of our own custom tests. With the Nitpicking litmus tests, we wrote assertion-based tests when possible and manually checked other properties (e.g., when testing for the existence, rather than avoidance, of a particular behavior). We ran all the listed relaxed, release/acquire and seq-cst tests, all of which exhibited the expected behaviors.

Whereas the Nitpicking litmus tests only tested the memory ordering behaviors of loads and stores, we performed additional tests to verify the treatment of, for example, read-modify-writes in CDSHECKER. In one such test we ran two threads, with each thread performing n identical `fetch_add(1)` operations on a single variable. We verified

that we see the correct number of distinct execution behaviors (enumerating rf) and that each execution yields a sum of $2n$. We performed other similar tests and checked the combinatorial behavior.

12.3. Comparison to CPPMEM and Nitpick

Researchers have developed two tools—CPPMEM [Batty et al. 2011] and Nitpick [Blanchette et al. 2011]—for exploring the behaviors of short code fragments under the C/C++ memory model. Both of these tools are targeted toward understanding the memory model and not toward testing real code. Additionally, Nitpick is not publicly available, and due to various constraints of CPPMEM, it is impossible to port our benchmarks to CPPMEM. Hence, we cannot directly compare these tools to CDSCHECKER using our benchmarks.

Instead, to roughly compare CDSCHECKER to Nitpick, we reconstructed the largest relaxed WRC example for which they published results. Their example contained additional constraints to limit the exploration to a subset of the legal executions by constraining loads to specific values, while CDSCHECKER is intended to explore all legal executions of the program and hence CDSCHECKER must explore a much larger space of executions. CDSCHECKER took 0.03s to explore all possible executions for this example, while the published results show that Nitpick took 982s to explore a subset of the results. We then ran our unrestricted version of the benchmark on CPPMEM, and it took 472.87s to complete. CDSCHECKER is significantly faster than both CPPMEM and Nitpick as both of those tools make modification orders explicit. CDSCHECKER avoids enumerating modification orders, thereby exponentially decreasing its search space. The other two tools also use generic search or SAT solving frameworks, whereas CDSCHECKER has been designed specifically for the C/C++ memory model and can leverage memory model constraints to prune its search.

13. RELATED WORK

Researchers have created tools to find bugs in concurrent data structures. State-based model-checkers such as SPIN [Holzmann 2003] can be used to debug designs for concurrent data structures. The CHES [Musuvathi et al. 2008] tool is designed to find and reproduce concurrency bugs in C, C++, and C#. It systematically explores thread interleavings. Line-Up [Burckhardt et al. 2010] extends CHES to check for linearization. The Inspect tool combines stateless and stateful model-checking to model-check C and C++ code [Yang et al. 2009; Wang et al. 2008; Yang et al. 2008]. These tools are designed to check code using the sequential consistency model rather than the more relaxed memory model of the C/C++ standards and therefore are not suitable for catching concurrency bugs arising from reordered memory operations.

Adversarial memory increases the likelihood of observing relaxed memory system behavior during testing [Flanagan and Freund 2010]. While it helps to uncover rare erroneous behaviors, it makes no guarantee of exhaustive testing. Moreover, adversarial memory is unable to simulate executions in which a load observes the value of a store that has not yet happened and therefore cannot catch bugs that are exposed by such behavior. CDSCHECKER can exhaustively explore a data structure's behavior for a given input and simulates loads that observe values of stores that appear later in the execution order.

State-based model-checkers have been developed for C# [Huynh and Roychoudhury 2006] and Java [De et al. 2008] that use reordering tables. As the C/C++11 memory model is not based on reordering tables, these approaches are not applicable to C/C++.

Other tools have been developed that systematically explore interleavings and memory operation reorderings. The Relacy race detector [Vyukov Oct] systematically explores thread interleavings and memory operation reorderings for C++11 code. The

Relacy race detector has a number of limitations that cause it to miss executions allowed by the C/C++ memory model. Like CDSCHECKER, Relacy imposes an execution order on the program under test. However, Relacy cannot produce executions (allowed by the memory model) in which loads read from stores that appear later in the execution order. Moreover, Relacy derives the modification order from the execution order; it cannot simulate (legal) executions in which the modification order is inconsistent with the execution order. Relacy also does not support partial order reduction.

Researchers have formalized the C++ memory model [Batty et al. 2011]. The CPPMEM tool is built directly from the formalized specification with a primary goal of allowing researchers to explore implications of the memory model. It explores all legal modification orders and reads-from relations—a source of redundancy—and therefore must search a significantly larger search space than CDSCHECKER, whose search algorithm limits redundancy by only exploring the space of legal reads-from relations. Furthermore, at this point CPPMEM lacks support for much of the C/C++ language. Nitpick translates the memory model constraints into SAT problems and then uses a SAT solver to find legal executions [Blanchette et al. 2011]. Simple experiments reveal that CDSCHECKER is significantly faster than either of these tools.

Several tools have been designed to detect data races in code that uses standard lock-based concurrency control [Elmas et al. 2007; Flanagan and Freund 2009; Lucia et al. 2010; Engler and Ashcraft 2003; Savage et al. 1997]. These tools typically verify that all accesses to shared data are protected by a locking discipline. They are not designed to check concurrent code that makes use of low-level atomic operations.

In the context of relaxed hardware memory models, researchers have developed tools for inferring the necessary fences [Kuperstein et al. 2011] and stateful model-checkers [Kuperstein et al. 2010; Jonsson 2009; Park and Dill 1999].

Researchers have also argued that reasoning about relaxed memory models is challenging and have made a case that compilers should preserve sequential consistency [Marino et al. 2011]. Whether such approaches can replace the need for a relaxed memory model depends to some degree on the memory models of future processors. We agree with the authors regarding the difficulty of reasoning about relaxed memory models, and we believe that tool support is necessary.

Our previous work presented the CDSCHECKER tool [Norris and Demsky 2013] and showed that it can effectively unit test data structures and discovered bugs in peer-reviewed concurrent data structures. This article extends the conference publication by showing that the core model checking algorithm used by CDSCHECKER is correct.

14. CONCLUSION

The C/C++ memory model promises to make it possible to write efficient, portable low-level concurrent data structures. The weak memory model that C/C++ provides for these low-level operations can result in unexpected program behaviors and can make writing correct code challenging. CDSCHECKER is the first tool that can both test real concurrent data structures while still simulating all of the weak memory model behaviors that C/C++ implementations are likely to produce. Our results indicate that CDSCHECKER can successfully test real low-level concurrent code.

A. EXAMPLES THAT MOTIVATED THE DEFINITION OF SATISFACTION ORDER

In this section, we list several examples of satisfaction cycles that motivated our definition of satisfaction order.

A.1. Information Flows through Conditional Branches

Figure 12 shows an implicit information flow from the load in Line 6 to the store in Line 10 via control flow. Figure 13 shows the execution that exhibits the implicit flow satisfaction cycle.


```

1  atomic_int x,y;
2  /* Initially x=y=0 */
3
4  void T1() {
5      int t=0;
6      int r1=x.load(relaxed);
7      if (r1==0)
8          t=1;
9      if (t==0)
10         y.store(1,relaxed);
11 }
12
13 void T2() {
14     int r2=y.load(relaxed);
15     x.store(r2,relaxed);
16 }
17
18 /* Can r1=1, r2=1? */

```

Fig. 12. Implicit flow satisfaction cycle.

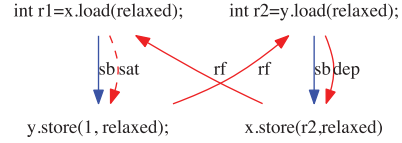


Fig. 13. Implicit flow satisfaction cycle.

```

1  atomic_int x,y,f,s;
2  /* Initially x=y=f=s=0 */
3
4  void T1() {
5      int r1=x.load(relaxed)
6      if (r1==0) {
7          f.store(1,relaxed);
8      }
9      s.store(1, release);
10 }
11
12 void T2() {
13     int r2=s.load(acquire);
14     int r3=f.load(relaxed);
15     if (r2==1 && r3==0) {
16         y.store(1,relaxed);
17     }
18 }
19
20 void T3() {
21     int r4=y.load(relaxed);
22     x.store(r4, relaxed);
23 }
24
25 /* Can r1=1, r2=1, r3=0, r4=1? */

```

Fig. 14. Implicit flow #2 satisfaction cycle.

One might believe that the implicit information flow from Figure 12 could be addressed by a thread local analysis that detects the flow from the load to the store through the conditional code.

Figure 14 shows that prohibiting such flows is not so simple. Figure 15 shows the execution that exhibits the second implicit flow satisfaction cycle. The absence of the store to *f* in *T1* implicitly leaks information across the synchronization from *s.store* to *s.load*.

This suggests the following more conservative thread local constraint:

$a \xrightarrow{sat} b$ if: *a* is a load, *b* is a store, and there exists a condition statement (e.g., if statement) a_0 such that a_0 depends on *a* and $a_0 \xrightarrow{sb} b$.

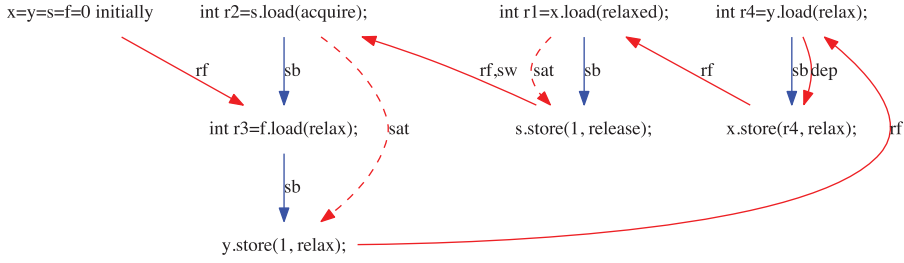


Fig. 15. Implicit flow #2 satisfaction cycle.

```

1  atomic_int x[2], idx, y;
2  /* Initially x[0]=1, idx=0, y=0 */
3
4  int r1, r2, r3; /* "local" variables */
5
6  void T1() {
7      r1=idx.load(relaxed);
8      x[r1].store(0, relaxed);
9
10     /* Key point: can we guarantee that &x[0] == &x[r1]? */
11     r2=x[0].load(relaxed);
12     y.store(r2);
13 }
14
15 void T2() {
16     r3=y.load(relaxed);
17     idx.store(r3, relaxed);
18 }
19
20 /* Can r1=1, r2=1, r3=1? */

```

Fig. 16. Address satisfaction cycle.

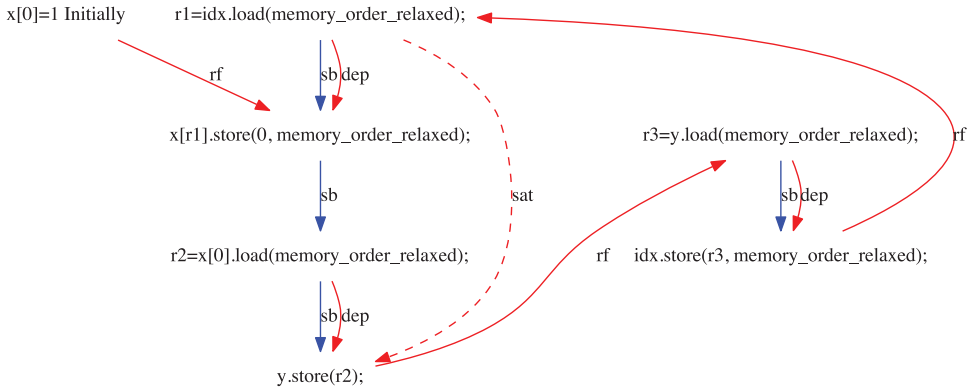


Fig. 17. Address satisfaction cycle.

A.2. Implicit Flows through Store Addresses

In Figure 16, we have an implicit flow of information from the store in Line 8 to the load in Line 11. Figure 17 shows the execution that exhibits the address satisfaction cycle. If $r1$ is 0, then the store in Line 8 will prevent the load in Line 11 from seeing the initial value of 1 in $x[0]$.

It does not appear that processors in general will guarantee a dependence between the loads. But we can get by with the following alternative constraint:

```

1  atomic_int x, y, z;
2  /* Initially x=y=z=0 */
3
4  int r1, r2, r3; /* "local" variables */
5
6  void T1() {
7      x.store(relaxed, 1);
8      y.store(release, 1);
9  }
10
11 void T2() {
12     r1 = y.load(acquire);
13     r2 = x.load(relaxed);
14     if (r1==1 && r2==0)
15         z.store(1, relaxed)
16 }
17
18 void T3() {
19     r3=z.load(relaxed);
20     if (r3==1)
21         y.store(relaxed, 1);
22 }
23
24 /* Can r1=1, r2=0, r3=1? */

```

Fig. 18. Failed synchronization satisfaction cycle-load dep.

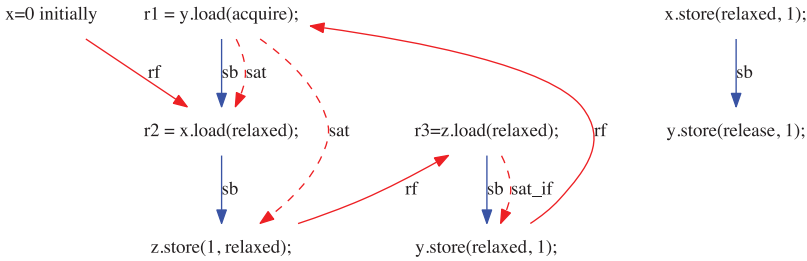


Fig. 19. Failed synchronization satisfaction cycle.

$a \xrightarrow{sat} b$ if: a is a load, b is a store, and there exists a store a_0 such that the address of a_0 depends on a and $a_0 \xrightarrow{sb} b$ ¹²

A.3. Synchronization Satisfaction Cycles

In this section, we explore two examples of satisfaction cycles in which the failure to synchronize produces the store that enables the failure.

Figure 18 shows an execution in which the failure to synchronize allows a later load to see an “old value” that then produces the store that allows the synchronization to fail. Figure 19 shows the execution that exhibits the failed synchronization satisfaction cycle.

Figure 20 shows an example where the failure to synchronize creates the store that allows the synchronization to fail. Figure 21 shows the execution that exhibits the failed synchronization satisfaction cycle with a store dependence. In this case, the failed synchronization is detected by another thread by the ordering of the stores.

We add the following constraint to eliminate satisfaction cycles involving the failure to synchronize. This constraint is required because acquire operations that synchronize

¹²Note that store a_0 need not be an atomic store. This constraint can be ignored if a_0 is a dead store and no load can ever read from it.

```

1  atomic_int x, y;
2  /* Initially x=y=0; */
3
4  int r0, r1, r2, r3; /* "local" variables */
5
6  void T1() {
7      y.store(10, relaxed);
8      x.store(1, release);
9  }
10
11 void T2() {
12     r0 = x.load(relaxed);
13     r1 = x.load(acquire);
14     y.store(11, relaxed);
15 }
16
17 void T3() {
18     r2 = y.load(relaxed);
19     r3 = y.load(relaxed);
20     if (r2==11 && r3==10)
21         x.store(0, relaxed);
22 }
23
24 /* Can r0 = 1, r1 = 0, r2 = 11, r3 = 10?

```

Fig. 20. Failed synchronization satisfaction cycle-store dep.

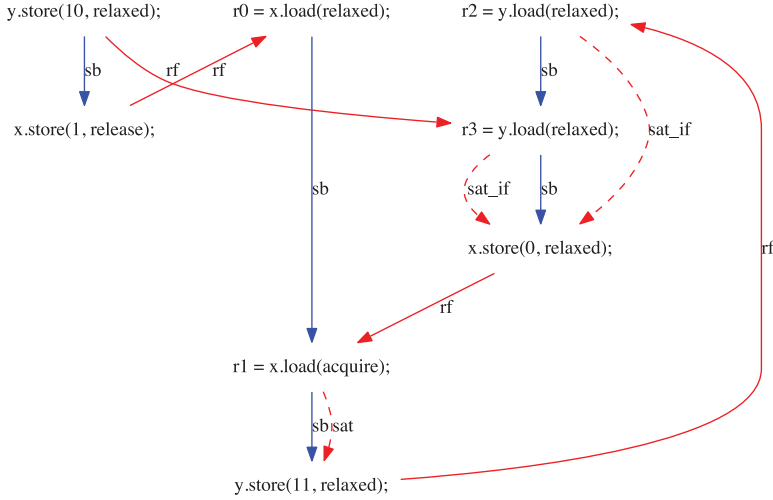


Fig. 21. Failed synchronization satisfaction cycle.

can “leak” information in a way that must be accounted for in our proof. We expect that all implementations will satisfy this constraint as load acquires must have a fence after the load (or provide a similarly strong guarantee for the load itself) to guarantee correct behavior in the case that the load synchronizes.

$a \xrightarrow{sat} b$ if: a is a load-acquire, b is a load or store, and $a \xrightarrow{sb} b$

REFERENCES

- Synchronization Algorithm Verificator for C++0x. <https://groups.google.com/forum/#!msg/comp.programming.threads/nSSFT9vKEe0/7eD3ioDg6nEJ>. Oct. 2012.
- Writing a (spinning) thread barrier using c++11 atomics. <http://stackoverflow.com/questions/8115267/writing-a-spinning-thread-barrier-using-c11-atomics>. Oct. 2012.
- Multithreading and Concurrency. <http://www.justsoftwaresolutions.co.uk/threading/>. Dec. 2012.

- MCS list-based lock. http://cbloomrants.blogspot.com/2011/07/07-18-11-mcs-list-based-lock_18.html. Oct. 2012.
- A look at some bounded queues - part 2. <http://cbloomrants.blogspot.com/2011/07/07-30-11-look-at-some-bounded-queues.html>. Oct. 2012.
- Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*.
- Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*.
- Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. 2011. Nitpicking C++ concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*.
- Hans Boehm. 2012. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.
- Hans J. Boehm. 2013. N3786: Prohibiting “out of thin air” results in C++14. Retrieved from <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm>.
- Hans J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- Hans-J. Boehm, Mark Batty, Brian Demsky, Olivier Giroux, Paul McKenney, Peter Sewell, and Francesco Zappa Nardelli. 2013. N3710: Specifying the absence of “out of thin air” results. Retrieved from <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html>.
- Hans-J. Boehm and Brian Demsky. 2014. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.
- Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: A complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- Cliff Click. 2007. A Lock-Free Hash Table. Retrieved from http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf.
- Arnab De, Abhik Roychoudhury, and Deepak D’Souza. 2008. Java memory model aware software validation. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*.
- Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*.
- Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- Cormac Flanagan and Stephen N. Freund. 2010. Adversarial memory for detecting destructive races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the Symposium on Principles of Programming Languages*.
- Patrice Godefroid. 1996. Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem. *Lecture Notes in Computer Science* 1032.
- Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the Symposium on Principles of Programming Languages*.
- Gerard J. Holzmann. 2003. *The SPIN Model Checker: Primer and Reference Manual* (1st ed.). Addison-Wesley Professional, New York, NY.
- Thuan Quang Huynh and Abhik Roychoudhury. 2006. A memory model sensitive checker for C#. In *Proceedings of the 14th International Conference on Formal Methods*.
- ISO/IEC 14882:2011. ISO/IEC 14882:2011, Information Technology – Programming Languages – C++.
- ISO/IEC 9899:2011. ISO/IEC 9899:2011, Information Technology – Programming Languages – C.
- Bengt Jonsson. 2009. State-space exploration for concurrent algorithms under weak memory orderings. *SIGARCH Computer Architecture News* 36, 5 (June 2009), 65–71.
- Michael Kuperstein, Martin Vechev, and Eran Yahav. 2010. Automatic inference of memory fences. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*.

- Michael Kuperstein, Martin Vechev, and Eran Yahav. 2011. Partial-coherence abstractions for relaxed memory models. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565.
- Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY.
- Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. 2010. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*.
- Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A case for an sc-preserving compiler. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- John M. Mellor-Crummey and Michael L. Scott. 1991. Synchronization without contention. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. 269–278.
- Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, NY.
- Mark Moir and Nir Shavit. 2004. *Handbook of Data Structures and Applications*. Chapman and Hall/CRC Press, Boca Raton, FL.
- Madanlal Musuvathi, Shaz Qadeer, Piramanayagam Arumuga Nainar, Thomas Ball, Gerard Basler, and Iulian Neamtiu. 2008. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*.
- Brian Norris and Brian Densky. 2013. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.
- Seungjoon Park and David L. Dill. 1999. An executable specification and verifier for relaxed memory order. *IEEE Trans. Comput.* 48 (1999), 227–235.
- Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computing Systems* 15, 4 (Nov. 1997), 391–411.
- Nir Shavit. 2011. Data structures in the multicore age. *Commun. ACM* 54, 3 (March 2011).
- Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
- Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.
- Dmitriy Vyukov. Oct. Relacy Race Detector. Retrieved from <http://relacy.sourceforge.net/>.
- Chao Wang, Yu Yang, Aarti Gupta, and Ganesh Gopalakrishnan. 2008. Dynamic model checking with property driven pruning to detect race conditions. *ATVA LNCS* 126–140 (2008).
- Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software*.
- Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Chao Wang. 2009. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. 279–295.

Received May 2014; revised March 2015; accepted July 2015