# CDSCHECKER: Checking Concurrent Data Structures Written with C/C++ Atomics

Brian Norris and Brian Demsky
University of California, Irvine
{banorris,bdemsky}@uci.edu


## Abstract

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.

Developing correct low-level concurrent code is well-known to be especially difficult under a weak memory model, where code behavior can be surprising. Building reliable concurrent software using C/C++ low-level atomic operations will likely require tools that help developers discover unexpected program behaviors.

In this paper we present CDSCHECKER, a tool for exhaustively exploring the behaviors of concurrent code under the C/C++ memory model. We develop several novel techniques for modeling the relaxed behaviors allowed by the memory model and for minimizing the number of execution behaviors that CDSCHECKER must explore. We have used CDSCHECKER to exhaustively unit test several concurrent data structure implementations on specific inputs and have discovered errors in both a recently published C11 implementation of a work-stealing queue and a single producer, single consumer queue implementation.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

***Keywords*** relaxed memory model; model checking


*OOPSLA '13*, October 29–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509514

## 1. Introduction

With the wide-scale deployment of multi-core 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 [14, 33]. 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, carefully designed concurrent hash tables can scale to many hundreds of cores [14]. Traditionally, developers had to 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 [2, 3, 12] 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++ [11]. 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 freedoms 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 [8] 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, 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 [22]. 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 model-checking rests on a class of optimization techniques known as dynamic partial-order reduction (DPOR) [20]. 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 that 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 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 some optimizations influenced by DPOR.

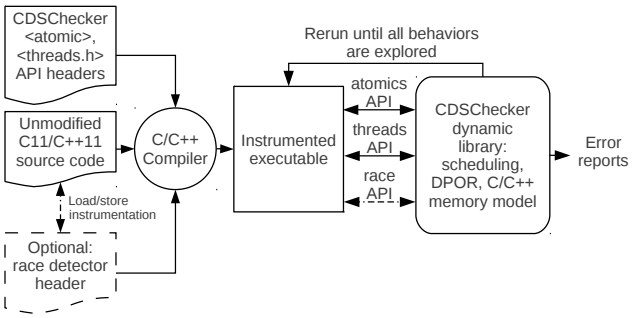


**Figure 1.** CDSCHECKER system overview

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 non-atomic variables without proper synchronization constitutes a data race, which yields undefined behavior [2]. 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 [8] enumerates all potential modification orders and reads-from relations in programs (under a limited subset of C/C++ language constructs), 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 [37] 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 paper.

- Supporting `memory_order_consume` requires a compiler's deep knowledge of data dependences. 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 CHESS [34] 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 run-time option.
- CDSCHECKER may not explore all behaviors involving satisfaction cycles. Not only are satisfaction cycles difficult to generate in a model-checker, but 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. See Appendix B for further discussion.
- 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.
- In rare circumstances, CDSCHECKER can generate false positives (behaviors not allowed by the memory model) due to an absence of dependence information. Given additional information from the compiler, however, it would be straightforward to check each generated trace for these false positives.

### 1.4 Contributions

This paper 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.
- **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 11 seconds, empirical results show that our tool is efficient in practice.

The remainder of the paper is organized as follows. Section 2 presents an example. Section 3 reviews important aspects the C/C++ memory model. Section 4 gives 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 8 discusses how we handle fairness and memory liveness. Section 9 evaluates CDSCHECKER. Section 10 presents related work. We conclude in Section 11.

## 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 inter-thread ordering and visibility properties (discussed more in Section 3).

In this example, a few simple interleavings of `threadA()` and `threadB()` show that we may see executions in

```
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 }
```

**Figure 2.** C++11 Code Example

which $\{$`r1 = r2 = 0`$\}$, $\{$`r1 = 0` $\wedge$ `r2 = 1`$\}$, or $\{$`r1 = 1` $\wedge$ `r2 = 0`$\}$, but it is somewhat counter-intuitive 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 that 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 paper, 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). Appendix A 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 [8], adapted to suit its usage in CDSCHECKER.

***Sequenced-Before*** The order of program operations within a single thread of execution establishes an intra-thread *sequenced-before* (*sb*) relation. Note that while some operations in C/C++ provide no intra-thread ordering—the equality operator (==), for example—we ignore this detail and assume that *sb* totally orders all operations in a thread.

***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 non-trivial, 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.

[1] We don't support consume due to implementation obstacles in detecting data dependencies. See Section 4.4.

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

***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 that, 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 consistent total ordering. For instance, the surprising behavior in Section 2's example 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 [20]. 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.

```
1:  Initially: EXPLORE(∅)

2:  function EXPLORE(S)
3:      s ← last(S)
4:      PROCESSACTION(S)
5:      if ∃p_0 ∈ enabled(s) then
6:          threads(s) ← {p_0}
7:          threadsdone ← ∅
8:          while ∃p ∈ threads(s) \ threadsdone do
9:              t ← next(s, p)
10:             behaviors(t) ← {Initial behaviors}
11:             behavedone ← ∅
12:             while ∃b ∈ behaviors(t) \ behavedone do
13:                 EXPLORE(S.(t, b))
14:                 behavedone ← behavedone ∪ {b}
15:             end while
16:             threadsdone ← threadsdone ∪ {p}
17:         end while
18:     end if
19: end function
```

**Figure 3.** CDSCHECKER algorithm

### 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 [20]. 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. 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 CDSCHECKER 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

[2] We 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.

behaviors of the transition *t* that was selected in the outer loop. Section 4.2 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 paper.

### 4.2 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 pecisely, 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.

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):

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

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 $(\nexists Z \in \textit{stores}(Y).\ X \xrightarrow{hb} Z \xrightarrow{hb} Y)$ 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 $\nexists X \in \textit{may-read-from}(Y)$ such that $X \xrightarrow{hb} Y$, then we report an *uninitialized load*—a bug in the program under test.

The reads-from mechanism allows CDSCHECKER to explore most of the behaviors of the C/C++ memory model without rearranging the thread schedule. In fact, in the absence of synchronization or sequentially consistent operations, CDSCHECKER does not use the DPOR backtracking mechanism to change the thread schedule at all.

### 4.3 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 [21]. 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 (e.g., mutexes).

### 4.4 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 [28]. 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 intra-thread *carries a dependency to* relation, we can extend our approach to support release/consume synchronization. One approach is to associate a secondary clock vector

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

with a program event if it is *dependency ordered* (§1.10p9-10 [3]) 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.

### 4.5 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, 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 4. For example, consider READ-READ COHERENCE (§1.10p16 [3]); 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 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 [3])
- 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 [3])
- RMW / MO CONSISTENCY: A read-modify-write must be ordered after the store from which it reads (§29.3p12 [3])
- RMW ATOMICITY: A read-modify-write must be ordered immediately after the store from which it reads (§29.3p12 [3])

### 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

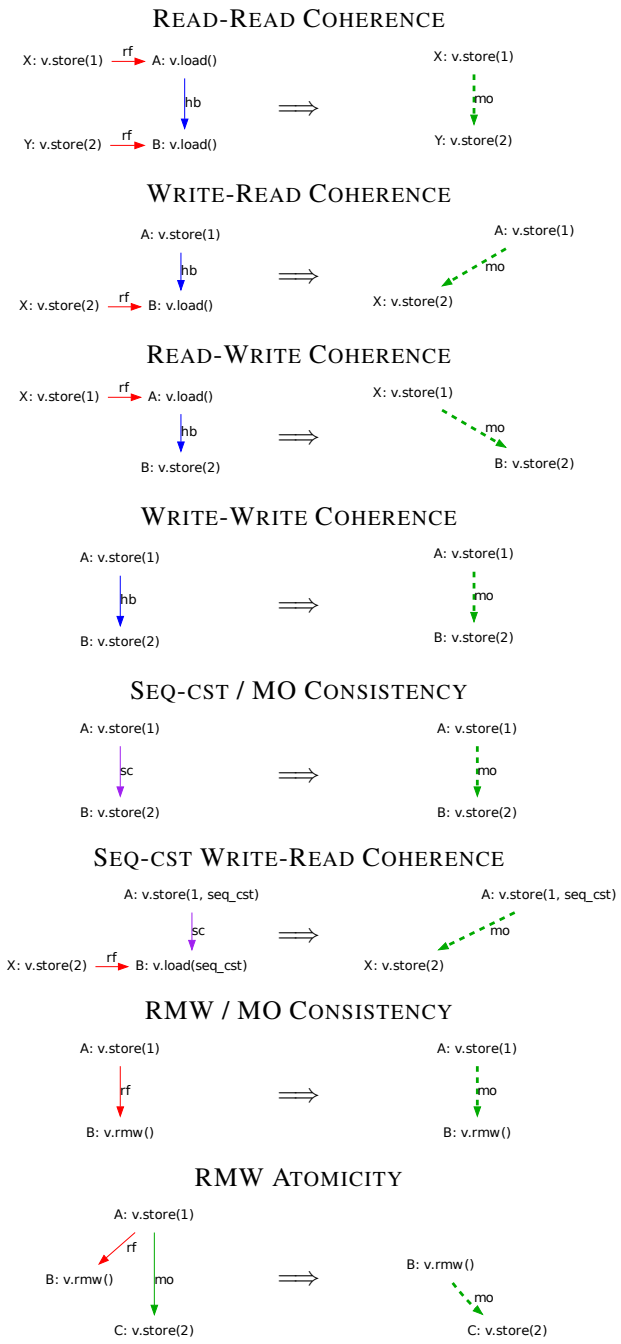


**Figure 4.** Modification order implications. On the left side of each implication, $A$, $B$, $C$, $X$, and $Y$ must be distinct.

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 3 stores and 1 initialization (the permutations of a 4-element sequence) and considers 4 potential stores to read from at line 7, yielding 96 combinations. However, one can easily see that there are actually only 3 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 doesn't 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 3 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 4. 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 $Y$, we only need to consider the most recent load $X_i$, from each thread $i$, which happens before $Y$ and reads from the same object. For any other load $Z$ (reading the same object) that happens before $Y$, either $Z = X_i$ for some $i$, or else $Z \xrightarrow{sb} X_j$ 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 $X_1, X_2, \ldots, X_n$ from earlier in the execution order. As long as $X_i$ does not happen before $Y$, it may read from $Y$, and so CDSCHECKER will add the pair $(\mathit{value}(Y), \mathit{thread}(Y))$ to the set $\mathit{futurevalues}(X_i)$ for each $i = 1, \ldots, n$ (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 $\mathit{futurevalues}(X_i)$ 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 that 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 $(1, \texttt{threadB})$. 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. If, however, CDSCHECKER finds that such a store is no longer possible, then the promise *fails* and we must discard the execution as invalid.

Because programs create threads dynamically, loads may read future values from stores whose threads do not yet exist at the time of the load. So when establishing a promise from an observed future value pair $(v, t)$, we assert that thread $t$ or *one of its descendants* will write the value $v$. Thus, a promise must track the set of threads which are available to satisfy the promise. This set grows when one of its member threads forks another thread, and it shrinks when one of its member threads can no longer satisfy the promise (e.g., the thread has completed). If at any point this set is empty, then the promise fails.

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 non-speculative 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, we can merge the promise and store nodes in the constraints graph, retaining the constraint information that we already gathered.

### 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 [3]), 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, 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 non-atomic 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 an inconsistent synchronization relation, we terminate the execution as infeasible. 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 4.4, 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 [3]), 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 5 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 synchronization[4]: a store-release $A$ synchronizes with a load-acquire $B$ if $B$ reads from a modification $M$ in the release sequence headed by $A$ [3]. In Figure 5, 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 or not a given sequence of modifications is contiguous (e.g., in Figure 5 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 unsynchronized[5] 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 so 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 se-

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

[5] Lazy 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.

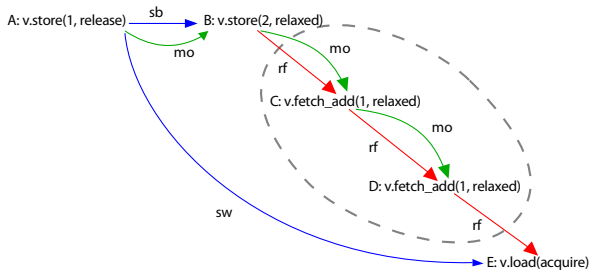


**Figure 5.** An example release sequence. Program execution is generally ordered from left to right.

quences are broken (i.e., non-contiguous); such a minimally-synchronizing solution precludes the need to explore other release sequence combinations in which the release sequences resolve to some non-empty 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 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 realizing 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 4, RMW ATOMICITY). Such programs will never incur the costs of the more complicated constraints checks for determining a contiguous subsequence of *mo*.

[6] We also observed simple release sequences in the presence of fence operations (see Appendix A).

## 8. 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 [3]), posing a practicality problem for our exhaustive search. We conclude that, for some programs which rely on memory system liveness, we must trade off 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 CHESS [34] 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.

## 9. 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://demsky.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 be-

| 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 |

**Figure 6.** Benchmark Results

havioral differences in our benchmarks results when using GCC to compile them with and without optimization.

### 9.1 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 [29]. 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 [33].

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 6), we record the total number of times CDSCHECKER executed the test program (# Executions) and the number of 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 8). For all benchmarks with non-atomic 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 [29]. It predominantly utilizes relaxed operations (for efficiency) while utilizing fences and release/acquire synchronization to establish ordering. While the paper 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 3 work items and takes 2 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 paper'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 [7]. 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 [1], 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 non-atomic 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 7 executions.

**Dekker critical section:** This implements a simple critical section using Dekker's algorithm [4], where a pair of non-atomic 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*) [5, 32]. 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 [6]. 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 [33] 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 thirty or more minutes to complete, whereas the current results show an exploration time of under 11 seconds.

### 9.2 Litmus Tests

To help verify that CDSCHECKER performs sound exploration of the memory model, we tested it against a set of litmus tests, including the tests described in Nitpicking [10] 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, e.g., read-modify-writes in CDSCHECKER. 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.

### 9.3 Comparison to CPPMEM and Nitpick

Researchers have developed two tools—CPPMEM [8] and Nitpick [10]—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 0.03 seconds to explore all possible executions for this example, while the published results show that Nitpick took 982 seconds to explore a subset of the results. We then ran our unrestricted version of the benchmark on CPPMEM, and it took 472.87 seconds 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.

## 10. Related Work

Researchers have created tools to find bugs in concurrent data structures. State-based model-checkers such as SPIN [23] can be used to debug designs for concurrent data structures. The CHESS [34] tool is designed to find and reproduce concurrency bugs in C, C++, and C#. It systematically explores thread interleavings. However, it can miss concurrency bugs as it does not explore all thread interleavings nor does it reorder memory operations. Line-Up [13] extends CHESS to check for linearization. Like CHESS, it can miss bugs that are exposed by reordering of memory operations. The Inspect tool combines stateless and stateful model-checking to model-check C and C++ code [38–40]. The Inspect tool checks code using the sequential consistency model rather than the more relaxed memory model of the C/C++ standards and therefore may miss concurrency bugs arising from reordered memory operations. Adversarial memory increases the likelihood of observing relaxed memory system behavior during testing [19]. 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# [24] and Java [15] 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 [37] 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 [8]. 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 [10]. 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 [16–18, 30, 36]. 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 [27] and stateful model-checkers [25, 26, 35].

Researchers have also argued that reasoning about relaxed memory models is challenging and have made a case that compilers should preserve sequential consistency [31]. 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.

## 11. 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.

## Acknowledgments

This project was partly supported by a Google Research Award and by the National Science Foundation under grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786.

## References

[1] `http://stackoverflow.com/questions/8115267/writing-a-spinning-thread-barrier-using-c11-atomics`. Oct. 2012.

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

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

[4] `http://www.justsoftwaresolutions.co.uk/threading/`. Dec. 2012.

[5] `http://cbloomrants.blogspot.com/2011/07/07-18-11-mcs-list-based-lock_18.html`. Oct. 2012.

[6] `http://cbloomrants.blogspot.com/2011/07/07-30-11-look-at-some-bounded-queues.html`. Oct. 2012.

[7] `https://groups.google.com/forum/#!msg/comp.programming.threads/nSSFT9vKEe0/7eD3ioDg6nEJ`. Oct. 2012.

[8] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*, 2011.

[9] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*, 2013.

[10] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, 2011.

[11] H. Boehm. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2012.

[12] H. J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[13] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[14] C. Click. A lock-free hash table. `http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf`, May 2007.

[15] A. De, A. Roychoudhury, and D. D'Souza. Java memory model aware software validation. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2008.

[16] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[17] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[18] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[19] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[20] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the Symposium on Principles of Programming Languages*, Jan. 2005.

[21] P. Godefroid. Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem. *Lecture Notes in Computer Science*, 1996.

[22] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the Symposium on Principles of Programming Languages*, 1997.

[23] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2003.

[24] T. Q. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *Proceedings of the 14th International Conference on Formal Methods*, 2006.

[25] B. Jonsson. State-space exploration for concurrent algorithms under weak memory orderings. *SIGARCH Computer Architecture News*, 36(5):65–71, June 2009.

[26] M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*, 2010.

[27] M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[29] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[30] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[31] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an sc-preserving compiler. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

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

[33] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 1996.

[34] M. Musuvathi, S. Qadeer, P. A. Nainar, T. Ball, G. Basler, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, 2008.

[35] S. Park and D. L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48, 1999.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computing Systems*, 15:391–411, Nov. 1997.

[37] D. Vyukov. Relacy race detector. `http://relacy.sourceforge.net/`, 2011 Oct.

[38] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. *ATVA LNCS*, (126–140), 2008.

[39] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software*, 2008.

[40] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 279–295, 2009.

# A. Fences

In addition to the atomic loads, stores, and read-modify-writes discussed in the body of the paper, 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 [3]) and have additional modification order constraints and synchronization properties, whose support we will discuss in this appendix.

### A.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.

- SC FENCES RESTRICT RF: Seq-cst fences impose restrictions on the "oldest" store from which a load may read (§29.3p4-6 [3]).
- SC FENCES IMPOSE MO: A pair of stores separated by seq-cst fences must form *mo* consistently with *sc* (§29.3p7 [3]). Notably, the C++ specification leaves out the COLLAPSED constraints that are presented here, but they are included in the formal model developed for [8]. 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.

### A.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 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 8 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 intra-thread ordering *sb* (to find prior fence-releases) but do not require querying the partially-ordered *mo*-graph.

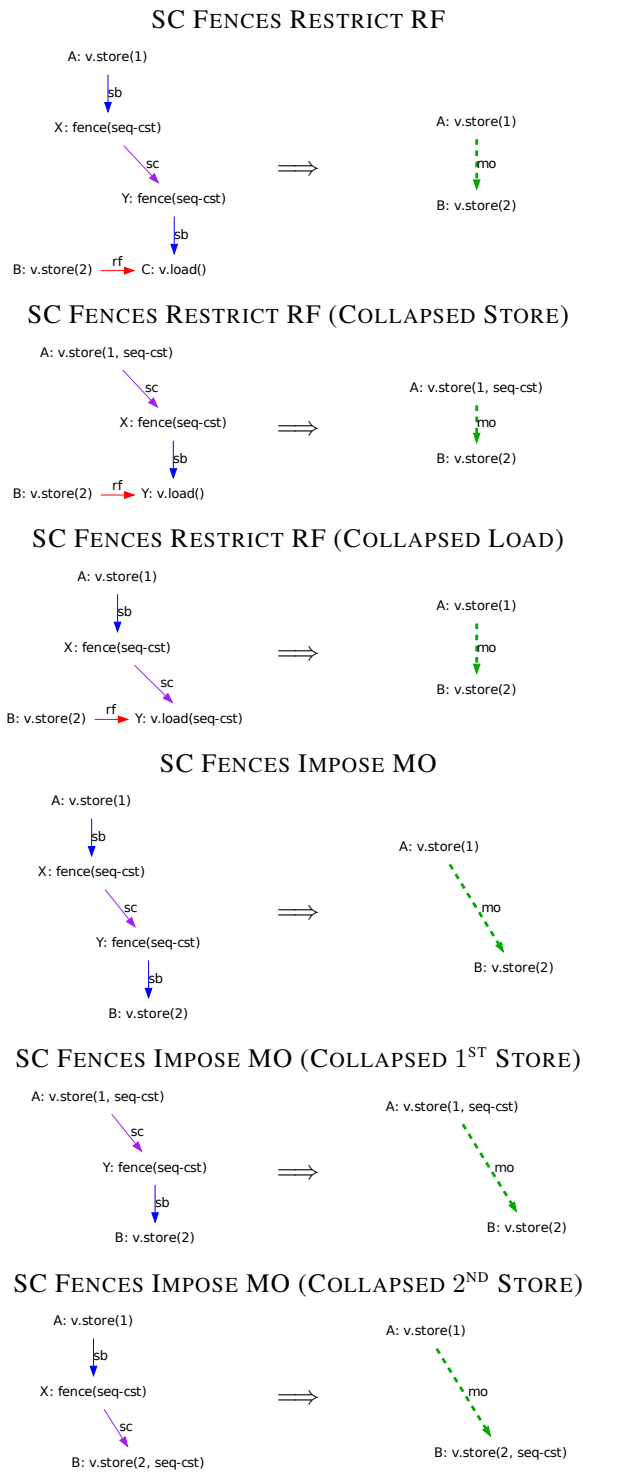


**Figure 7.** Fence modification order implications. On the left side of each implication, $A$, $B$, $C$, $X$, and $Y$ must be distinct.

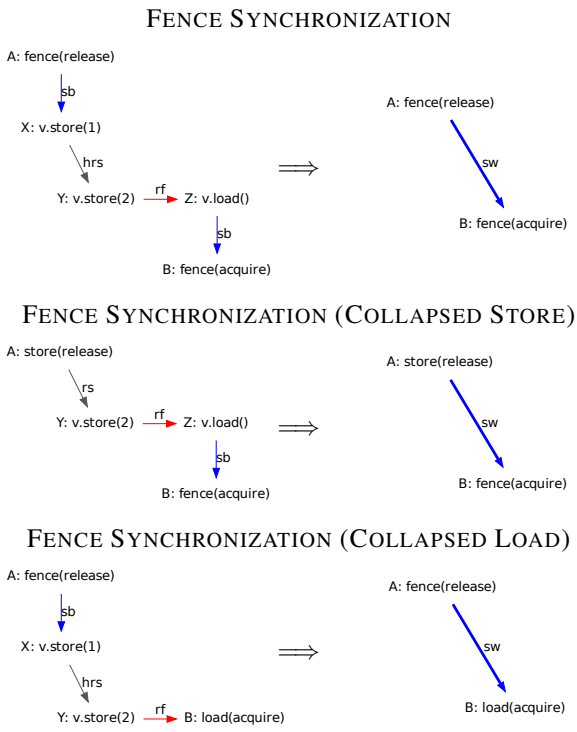


**Figure 8.** Fence synchronization implications

### A.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 A.2. So, whenever CDSCHECKER 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 4.3, 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.

## B. Satisfaction Cycles

An issue with the C/C++ specification is that it allows programs to observe various types of out-of-thin-air values through the use of *satisfaction cycles*, in which the effects of an action justify performing the action in the first place, producing unintuitive results. The problem of satisfaction cycles has already been recognized by other researchers and can make it difficult to prove properties of code in many cases [9]. It is clear that the specification authors would like to rule out satisfaction cycles, and they make an attempt to rule these out with requirement §29.3p9 that states:

> An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that:
>
> - if an evaluation B observes a value computed by A in a different thread, then B does not happen before A, and
> - if an evaluation A is included in the sequence, then every evaluation that assigns to the same variable and happens before A is included.

Despite efforts to disallow all out-of-thin-air values, in the end the C/C++ authors concede that some sub-optimal behaviors are not ruled out by the specification. They then simply state that implementations "should not allow such behavior" without providing the details of exactly what this means.

For instance, the specification mentions that the formal requirements allow `r1 = r2 = 42` in the following program fragment:

```
1  atomic<int> x(0), y(0);
2
3  void threadA() {
4     int r1 = x.load(memory_order_relaxed);
5     if (r1 == 42)
6        y.store(r1, memory_order_relaxed);
7  }
8  void threadB() {
9     int r2 = y.load(memory_order_relaxed);
10    if (r2 == 42)
11       x.store(42, memory_order_relaxed);
12 }
```

But because CDSCHECKER operates as a runtime framework, it only sees the program statements that are actually executed. So while it will explore all behaviors that do not involve a satisfaction cycle, CDSCHECKER cannot guarantee that it explores behaviors where the behavior circularly justifies itself. One such example is where the only justification for taking a conditional branch is hidden behind the branch. Thus, CDSCHECKER will never see `r1 = r2 = 42` in the above example. This variation from the formal reading of the C/C++ specification is desirable since it prevents CDSCHECKER from warning developers about program behavior which is forbidden by the C and C++ specifications ("implementations should not allow such behavior") and which should never be produced by a compiler or processor.

Satisfaction cycles in general make verifying software intractable. It remains an open question of how to best constrain the C/C++ memory model to disallow satisfaction cy-

cles while still allowing common compiler optimizations and achieving good performance on all architectures.

With constraints to rule out satisfaction cycles, the correctness of CDSCHECKER follows from an induction on the evaluation sequence.

## C. 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 $X \overset{rf}{\not\to} A$ whenever there exists a store $B$ such that

$$A \overset{hb}{\longrightarrow} B \wedge B \overset{mo}{\longrightarrow} X.$$

Allowing $X \overset{rf}{\to} A$ 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, CDSCHECKER would send a future value, not realizing the cycle until it established the *rf* edge concretely. Similarly, we do not send a future value from store $B$ to load $A$ if $A \overset{hb}{\longrightarrow} B$.

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 CDSCHECKER 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 $Z$ does exist, however, we delay sending the future value until $Z$'s promise is resolved—breaking the cycle while still allowing non-cyclic dependences 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*$(X)$.