AutoMO: Automatic Inference of Memory Order Parameters for C/C++11

Peizhao Ou and Brian Demsky
University of California, Irvine
{peizhaoo,bdemsky}@uci.edu

Abstract

Many concurrent data structures are initially designed for the sequential consistency (SC) memory model. Developers often then implement these algorithms on real-world systems with weaker memory models by adding sufficient fences to ensure that their implementation on the weak memory model exhibits the same executions as the SC memory model.

Recently, the C11 and C++11 standards have added a weak memory model to the C and C++ languages. Developing and debugging code for weak memory models can be extremely challenging. We present AutoMO, a framework to support porting data structures designed for the SC memory model to the C/C++11 memory model. AutoMO provides support across the porting process: (1) it automatically infers initial settings for the memory order parameters, (2) it detects whether a C/C++11 execution is equivalent to some SC execution, and (3) it simplifies traces to make them easier to understand. We have used AutoMO to successfully infer memory order parameters for a range of data structures and to check whether executions of several concurrent data structure implementations are SC.

1. Introduction

With the wide scale deployment of multi-core processors, software developers must write parallel software to leverage the benefits provided by additional cores. While it is relatively straightforward to use locks to protect concurrent data accesses, locks are often an impediment to writing code that effectively scales to many cores. A consequence of Amdahl's law is that even small regions of code that use coarse grain locking can significantly limit the overall speedup achieved from parallelism.

Careful data structure design can improve scalability by supporting multiple simultaneous operations and by reducing the time taken by individual operations. Researchers and practitioners have developed a wide range of concurrent data structures designed to meet these goals [12, 27, 30, 36–38, 48].

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 [30]. Traditionally, developers had to target their implementations to a specific platform and compiler as the implementations relied on low-level platform details and often required coding components in assembly.

In 2011, the C/C++ standardization committees extended the C/C++ language standards with support for low-level atomic operations [2, 3, 13], which allow experts to craft efficient concurrent data structures that avoid the overheads of locks. The new C/C++ memory model provides memory operations with weaker semantics than the sequential consistency (SC) memory model to support real-world processors and compiler optimizations.[1]

1.1 Implementing Data Structures

Designing data structures directly for a weak memory model is extremely difficult. Weak memory models admit a number of surprising and non-intuitive behaviors [10]. A number of researchers have hypothesized that the predominant development model for concurrent data structures on weak memory models is that algorithm experts first design data structures for the much stronger, more intuitive SC memory model [14, 17]. Developers then implement the SC data structure design on a weaker language or hardware memory model by adding sufficient fences or memory order con-

[1] We are somewhat relaxed in our usage of the abbreviation SC and also use it to mean sequentially consistent.

OOPSLA '15 date, City.
Copyright © 2015 ACM [to be supplied]. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

straints to ensure that behaviors that arise from relaxed memory models do not break the data structure.

Moreover, we expect that developers will commonly use the following development methodology:

1. Developers will find an already existing concurrent data structure design that solves the problem at hand or design one based on their intuitions. Note that such designs often assume the SC memory model.
2. The developers may make minor adaptations to the basic design to fulfill their needs.
3. The developers then attempts to tune the memory order parameters to ensure that their implementation only admits the SC executions assumed in the original design. Although C++11 provides SC in the absence of data races by default (all memory accesses use `memory_order_seq_cst`), overly restrictive memory order constraints can incur significant performance overheads, and thus it is typically preferable to use the weakest constraints that still guarantee correctness.

A key challenge is avoiding mistakes in Step 3. There is anecdotal evidence that getting Step 3 correct is difficult (while optimizing for performance). Solving the problem in Step 3 for the Chase-Lev deque was a subject of an academic paper [28], and the published code in that paper for the C11 memory model contained errors in the memory order constraints [40].

We present an algorithm that takes as input a data structure implementation and a set of test cases and then automatically infers memory order parameters for the data structure that ensure that all executions of these test cases are equivalent to executions under the SC memory model.

1.2 Debugging Data Structures

The C/C++ memory model is formalized in terms of a *reads-from* (*rf*) relation that maps stores to the loads that read-from them [2, 3, 9]. The reads-from relation is then constrained by a number of constraints that ensure cache coherence, define the semantics of synchronization, and implement various memory order constraints.

Researchers have developed a range of testing tools for exploring the behaviors of code under the C/C++ memory model including CDSChecker [40], CppMem [9], and Relacy [49]. These tools dump execution traces[2] that list the memory operations and tell which store each load reads from. These traces can be very difficult to understand as they contain non-intuitive behaviors including (1) loads that read from stores older than the last store to the location and (2) loads that read from stores that appear after the load.

When a bug is discovered in a data structure implementation, these tools provide the developer with an execution trace that exposes the bug. To debug the implementation, it is often important to understand whether the buggy behavior arises because of the relaxed memory model or the trace is allowed by the SC memory model. Moreover, if the developer ported a data structure designed for the SC memory model, the presence of any execution that is not allowed by the SC memory model is worth investigating further.

Unfortunately, it can be surprisingly difficult to figure out whether a given trace (even relatively short with tens of operations) is allowed by the SC memory model. Even if loads may read from stores other than the last prior store to the same location, it may be possible to permute the operations such that the trace is consistent with the SC memory model while maintains the same reads-from relation.

Even if parts of a trace are prohibited by the SC memory model, rewriting the trace to be mostly SC with only a few violations eliminates the need to jump all over the trace when examining which store a load reads from.

It is known that the problem of checking whether an execution is SC even when the reads-from mapping is given is NP-complete[24, 25], and hence the complexity of checking whether a C/C++ trace is allowed by the SC memory model is NP-complete if the order of stores to a given location is not known.

We present an algorithm that efficiently solves the problem for traces produced by real-world concurrent data structures. We prove the correctness of our algorithm and evaluate it on a number of C/C++ data structure implementations.

Our SC checking approach is not specific to the C/C++ memory model — we essentially check whether there exists an SC trace that is consistent with the reads-from relation. Therefore, our approach generalizes to all axiomatic memory models that are formalized in terms of a reads-from relation. Of course, for stronger memory models it may be possible to lower the complexity bound of checking whether a trace is allowed by the SC memory model.

1.3 Contributions

This paper makes the following contributions:

- **Memory Order Parameter Inference:** It presents an approach that automatically infers memory order parameters, automating one of the more difficult aspects of using the C/C++ memory model.
- **Sequential Consistency Trace Checking:** It presents a new technique that checks whether a given trace under the C/C++ memory model is consistent with the SC memory model. It primarily targets unit testing and debugging concurrent data structures implementations.
- **Formalization:** It proves the correctness of the SC trace checking algorithm.
- **Trace Simplification:** It explores several approaches for reordering traces to make it easier for developers to understand an execution. When a trace is not allowed by the SC memory model, our algorithm prints a more readable trace with fewer gratuitous SC violations and guar-

[2] We use trace to informally refer to the order in which these tools print out memory operations. The C/C++ memory model does not define a trace.

antees that the SC violations are not flagged in an obviously wrong place.

- **Evaluation:** It presents an evaluation of the algorithm on traces of several real-world data structures. It found two bugs that it automatically fixed for one benchmark, one incorrect claim in an academic paper and it infers no worse (sometimes better) memory order parameters than the original manually developed versions for most benchmarks (9 out of 11).

2. Overview

Choosing memory order parameters often involves trading off weaker semantics for improved performance. For example, a wide range of C/C++ data structures implementations admit some behaviors that are prohibited by the sequentially consistent memory model. For example, queue implementations often only provide release/acquire synchronization between the enqueuing and dequeuing threads. Non-SC behaviors can be observed in test cases involving multiple such queues.

A key observation of this work is that although many data structure implementations do admit non-SC executions, this often arises only when the data structures are composed with others. Many practical data structure implementations are *internally SC* — such data structure implementations can view their internal behavior as sequentially consistent and blame any non-SC behaviors that are exposed by compositions on the external data structures.

All executions involving a single instance of an internally SC data structure in isolation are equivalent to SC executions. The key insight is that internally SC suffices to avoid breaking the internals of a data structure design.

2.1 Inference of Memory Order Parameters

Developers often have given some thought to the corner cases for their implementations. They know that resizing of concurrent data structures, dequeuing the last element, or dequeuing from an empty queue are all cases that must be given careful consideration. They sometime even write unit tests to cover such corner cases.

Many developers do not understand the subtle details of the C/C++ memory model. Reading developer blogs or StackOverflow threads regarding the topic reveals numerous examples of sophisticated developers who do not understand the key elements of the C/C++ memory model.

We have designed AutoMO to address this issue. AutoMO takes as input a data structure and a set of unit tests. It then outputs a set of assignments to the memory order parameters that make all of the test case executions SC.

Figure 1 presents AutoMO's basic approach. The approach taken is structured as follows:

1. Initialize the memory order parameters using the input parameter assignments generated from the previous test case. If this is the first test case, then AutoMO initializes the memory order parameters to `relaxed`.
2. Run the unit test using the CDSChecker model checking tool to exhaustively explore the legal executions of the test case.
3. Check each execution to detect whether the execution is equivalent to some SC execution.
4. If the execution is not equivalent to some SC execution, determine one or more memory operations to blame for the non-SC behavior. While AutoMO is not always guaranteed to blame the correct memory operation, it is guaranteed to blame a memory operation that can be repaired. Blaming a set of memory operations that are all already specified to be `memory_order_seq_cst` is not helpful as they cannot be strengthened.
5. Leverage repair patterns to strengthen memory order parameter assignments to fix the problem. In general, problematic executions cannot always be fixed with a single repair action. However, AutoMO repair actions guarantee that they will eventually ensure that the same memory operations cannot be blamed again in the future. This suffices to guarantee that AutoMO will always converge on a repair for a given execution.
6. After a given test case only exhibits SC executions, move to different unit tests.
7. When all test cases are completed, AutoMO outputs the final parameter assignments.

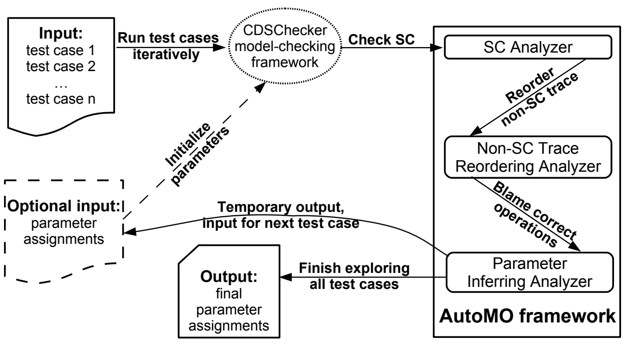

Input:
test case 1
test case 2
...
test case n
Run test cases iteratively
CDSChecker model-checking framework
Check SC
SC Analyzer
Reorder non-SC trace
Non-SC Trace Reordering Analyzer
Blame correct operations
Parameter Inferring Analyzer
Initialize parameters
Optional input: parameter assignments
Temporary output, input for next test case
Finish exploring all test cases
Output: final parameter assignments
AutoMO framework

Figure 1. AutoMO system overview

A key challenge behind AutoMO was developing an analysis that can quickly check whether an execution is SC, reorganize non-SC executions to be mostly-SC, and automatically discover operations that are likely to be responsible for introducing non-SC behaviors into an execution. These core technologies also have the potential to be useful for debugging or understanding concurrent data structures. Checking SC can identify executions that may be worth more careful inspection and presenting non-SC executions as mostly SC (plus blaming the appropriate operations) can greatly simplify understanding executions under weak memory models.

3. Definitions

We begin with a brief summary of the relations that comprise the formalization of the C/C++ memory model. We then continue with several definitions that we make use of throughout the paper.

3.1 Summary of 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). 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.

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

release: `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst` – a store-release may form release/consume or release/acquire synchronization. When a load-acquire reads from a store-release, it establishes a happens-before relation between the store and the load.

consume: `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.

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

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

- **Sequenced-Before:** The evaluation order within a program establishes an intra-thread *sequenced-before* (*sb*) relation—a strict preorder of the atomic operations over the execution of a single thread.
- **Reads-From:** The *reads-from* (*rf*) relation consists of store-load pairs (X, Y) such that Y takes its value from 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 the synchronization that occurs when certain atomic operations interact across threads.
- **Happens-Before:** In the absence of memory operations with the `consume` memory ordering, the *happens-before* relation is the transitive closure of the union of the sequenced-before relation and the synchronizes-with relation.
- **Sequentially Consistent:** All operations that declare the `memory_order_seq_cst` memory order have a total ordering (*sc*) in the program execution.
- **Modification Order:** Each atomic object in a program has an associated *modification order* (*mo*)—a total order of all stores to that object—which informally represents an ordering in which those stores may be observed by the rest of the program.

Program executions directly observe the reads-from relation by observing the values that loads return. The synchronizes-with, happens-before, sequentially consistent, and modification order orderings constrain the reads-from relation and are only indirectly observable by the effect that they have on the reads-from relation. As the SC memory model is strictly stronger than the C/C++ memory model, if we can find an SC execution trace that is consistent with both the reads-from relation and the sequenced-before relation of a C11 trace, then any constraints related to the remaining relations will also be satisfied.

Specifically, the synchronizes-with and happens-before relations are trivially satisfied by any SC trace as executions in the SC memory model essentially behave as if every load synchronizes with the store from which it reads. The sequentially consistent relation is satisfied because the SC trace requires all operations to have a total order that is consistent with the reads-from and sequenced-before relations. A consistent modification ordering exists for each memory location as the SC trace ensures that a total ordering exists for all operations. Thus, the subsequence of the SC total ordering containing all operations for a given memory location gives a consistent modification order for that memory location. Hence, the rest of this paper can focus on the problem of finding an SC ordering that is consistent with just the sequenced-before ordering and the reads-from relation.

3.2 Formalizing Traces

Figure 2 presents a simplified version of the memory operations that can appear in the input C/C++ execution traces. C/C++ traces have two basic types of operations: *StoreOps* and *LoadOps*. It is possible for a single atomic operation to perform both a load and store — such *RMW* operations are members of both sets. The input trace τ specifies both the intrathread *sb* partial order and the reads-from *rf* partial order. We can safely assume that the order of operations in the input trace τ is consistent with the intrathread ordering *sb*, but in general it may not be consistent with the reads-from

$$
\begin{aligned}
s \in \mathit{StoreOps} &= \{\mathit{store}\} \times \mathit{Address} \times \mathit{Value} \cup \\
&\quad \{\mathit{rmw}\} \times \mathit{Address} \times \mathit{Value} \\
l \in \mathit{LoadOps} &= \{\mathit{load}\} \times \mathit{Address} \cup \\
&\quad \{\mathit{rmw}\} \times \mathit{Address} \times \mathit{Value} \\
\mathit{op} \in \mathit{Ops} &= \mathit{StoreOps} \cup \mathit{LoadOps}
\end{aligned}
$$

Figure 2. Sets of memory operations in the input trace

$$
\begin{aligned}
\mathit{sb}: &\quad \mathit{Ops} \times \mathit{Ops} \\
\mathit{rf}: &\quad \mathit{StoreOps} \times \mathit{LoadOps} \\
\tau = &\quad \tau(1); ...; \tau(i); ...; \tau(n), \\
&\quad \text{where } \tau(i) \in \mathit{Ops} \text{ and the trace } \tau \text{ is consistent with the} \\
&\quad \text{intrathread execution order.}
\end{aligned}
$$

Figure 3. Relationships that define the input trace

$$
\begin{aligned}
\mathcal{SC}(\tau) &= \forall i, s, 1 \le i, s \le n, \\
&\quad (\langle \tau(s), \tau(i) \rangle \in \mathit{rf} \Rightarrow s < i) \wedge \\
&\quad (\forall j. s < j < i, \tau(j) \notin \mathit{StoreOps} \vee \\
&\quad \mathtt{address}(\tau(j)) \neq \mathtt{address}(\tau(i)))) \\
\mathtt{preserves\_}\mathit{sb}(\tau) &= \forall i, j, 1 \le i, j \le n, \\
&\quad \langle \tau(i), \tau(j) \rangle \in \mathit{sb} \Rightarrow i < j
\end{aligned}
$$

Figure 4. Trace predicates. The first predicate checks that each loads reads from a store that precedes the load and that there are no stores to the same address between the original load and the store. The second predicates checks that the reordering preserves the intrathread ordering.

rf ordering. Figure 3 presents these two partial orders along with the trace τ. The notation $\tau(i)$ indicates the *ith* operation in the trace τ.

The output of the algorithm is an execution trace τ_{isc} that totally orders the *Ops* to be consistent with the SC memory model and the intrathread *sb* ordering. The algorithm functions by inferring a partial order *isc*, which is denoted as the ordering constraints that the SC memory model places on the operations and shown in Figure 8. Note that the order *isc* is distinct from the *sc* order defined by the C/C++11 memory model. Figure 4 presents the predicate $\mathcal{SC}$ that checks that each load in the trace reads from the last prior store to the same memory location and hence that trace is consistent with the SC memory model.

3.3 Checking Sequential Consistency

A key component of our approach is checking whether there exists a reordering ϕ of the operations in a C/C++ trace that is consistent with the sequenced-before relation such that the $\mathcal{SC}$ predicate is true for the reordered trace τ^{ϕ}. If such an ordering exists, we say that the original trace τ is sequentially consistent.

Definition 3.1. (Reordering) A *reordering* of a trace τ is a permutation ϕ on $\{1, ..., n\}$ and the reordered trace τ^{ϕ} is $\tau(\phi(i))$.

Definition 3.2. (SC) A trace τ is sequentially consistent (*SC*) if there exists a reordering ϕ such that $\mathcal{SC}(\tau^{\phi})$ and τ^{ϕ} is consistent with the original intrathread ordering *sb* (i.e., satisfies the predicate `preserves_`*sb*).

4. Example

Figure 5 presents a single-producer single-consumer queue example that we will use to illustrate our approach. Lines 1 through 8 define a `node` struct with an atomic field `index` and an atomic field `next` pointing to the next `node`. Lines 10 through 32 define the `spsc_queue` class to maintain a `head` and a `tail` pointer. The `enqueue()` method initializes a new node, reads the `tail` pointer, stores the new node to the tail's `next` field, and updates the `tail` pointer. The `dequeue()` method reads the `head`, reads its `next` field, and if it is not `NULL`, updates the `head` and returns the value of the `index` field.

This data structure is trivially correct under the SC memory model, however, with the memory order parameters in the Figure, it is buggy under C/C++11. Consider the test case from Lines 34 through 44. This test case has two threads. One thread updates an array element and enqueues an index of `0`, and the other thread tries to dequeue an item, and if successful, loads the array element of the dequeued index.

However, without establishing proper synchronization, this implementation can have non-SC behaviors and lead to the buggy behavior of reading uninitialized values. We use this as a running example throughout the paper to present how AutoMO checks whether traces are SC, simplifies non-SC traces and then automatically infers memory order parameters.

5. SC Analysis Algorithm

Our algorithm takes as input a C/C++11 execution trace and determines (1) whether the trace is SC and if so (2) generates a reordered trace that satisfies the $\mathcal{SC}$ predicate.

As an example, Figure 6 presents a trace of one of the executions of the test case shown in Figure 5. In this trace, while the atomic load in Operation 2 (from Line 25) reads from the atomic store in Operation 7 (from Line 19), the atomic load in Operation 5 (from Line 28) reads from an uninitialized value instead of from the atomic store in Operation 4 (from Line 4). We denote Operation 0 as the store of uninitialized values, and we assign `0` to uninitialized values. While Operation 2 (reads from a later store) and Operation 5 (reads from an old store) obviously violate the $\mathcal{SC}$ predicate in this trace, we cannot rely on this fact to trivially decide whether the execution is SC or not since in general it might be possible to shuffle the order of the statements such that their behavior is consistent with the SC memory model while still maintaining the same reads-from relation.

Therefore, we need a systematic approach to check whether traces are SC. We begin by computing the partial

```
struct node {
  node(int idx) {
    next.store(NULL, memory_order_relaxed);
    index.store(idx, memory_order_relaxed);
  }
  atomic<node*> next;
  atomic<int> index;
};

class spsc_queue {
  node *head, *tail;
  public:
  spsc_queue() {
    head = tail = new node(-1);
  }
  void enqueue(int idx) {
    node* n = new node(idx);
    // Store of next field should be release
    tail->next.store(n, memory_order_relaxed);
    tail = n;
  }
  bool dequeue(int *idx) {
    node *tmp = head;
    // Load of next field should be acquire
    node *n = tmp->next.load(memory_order_relaxed);
    if (NULL == n) return false;
    head = n;
    *idx = n->index.load(memory_order_relaxed);
    delete (tmp);
    return true;
  }
};

spsc_queue *q;
atomic_int arr[2];
void thrd1() { // Thread 1
  arr[1].store(1, memory_order_relaxed);
  q->enqueue(1); // Enqueue index 1
}
void thrd2() { // Thread 2
  int idx;
  if (q->dequeue(&idx))
    arr[idx].load(memory_order_relaxed);
}
```

Figure 5. A buggy single-producer single-consumer queue

#	Thread	Operation	Order	Addr	Value	rf
1	1	atomic store	relaxed	0x2080	0x1	
2	2	atomic load	relaxed	0x5c08	0x6020	7
3	1	atomic store	relaxed	0x6020	0	
4	1	atomic store	relaxed	0x6028	0x1	
5	2	atomic load	relaxed	0x5c08	0	0
6	2	atomic load	relaxed	0x2078	0	0
7	1	atomic store	relaxed	0x5c08	0x6020	

Figure 6. Original execution trace for example test case

order $\xrightarrow{isc}$ as the union of the reads-from partial order $\xrightarrow{rf}$, the sequenced-before partial order $\xrightarrow{sb}$, and the synchronization created by thread starts and joins. We then use a fixed-point algorithm combined with inference rules to infer additional edges in the partial order $\xrightarrow{isc}$.

We use the partial order $\xrightarrow{isc}$ to reorder the trace of memory operations. Figure 7 presents the edges that comprise the $\xrightarrow{isc}$ relation for the example. Note that the load in Operation 5 should have an $\xrightarrow{isc}$ edge to the store in Operation 4 because SC constrains the load to read from the last store. In this graph, there exists a cycle ($4 \rightarrow 7 \rightarrow 2 \rightarrow 5 \rightarrow 4$), meaning that this execution is non-SC because it is impossible to generate a total order that is consistent with the $\xrightarrow{isc}$ relation when there exist a cycle.

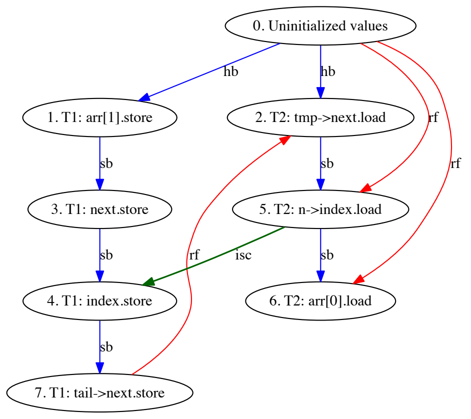


Figure 7. Edges in the $\xrightarrow{isc}$ relation for the example test case

5.1 Algorithm

The first component of the algorithm is a set of inference rules that build up the partial order $\xrightarrow{isc}$. Figure 8 presents these inference rules. The sequenced-before inference rule ensures that the *isc* partial order is consistent with *sb*. The reads-from inference rule and the write ordering inference rule together ensure that a load reads from the latest store to the given location. The join rule ensures that the end of a thread happens before the join operation returns. The thread creation rule ensures that creation of a new thread occurs before the new thread starts execution. Notably, for tools such as CDSChecker, CppMem, and Relacy, the reads-from information is provided and need not be extracted.

A simple topological sort of the $\xrightarrow{isc}$ relation generated by the inference rules in Figure 8 is not sufficient to compute a total SC execution order as the inference rules may not have established a total order on all of the stores to a given location. Figure 9 presents an example where the inference rules do not establish an ordering between the stores to `z`, yet ordering the store to `z` in Line 1 first yields traces that do not satisfy the predicate $\mathcal{SC}$.

Figure 10 presents our algorithm for checking whether an execution trace is allowed by SC and, if so, reordering the trace to be SC while maintaining the same intrathread order (*sb*). The algorithm begins by initializing a set of actions and then calling the SEARCHSC procedure to check whether the trace is SC. The SEARCHSC procedure begins by calling the UPDATESC procedure in Line 6 using the inference rules to compute the partial order $\xrightarrow{isc}$. In Line 10, those actions that are not ordered after any other actions in $\xrightarrow{isc}$ form a set (`searchset`) of candidate actions to incrementally build up the execution trace τ^{SC} in `seq`. If it selects a store

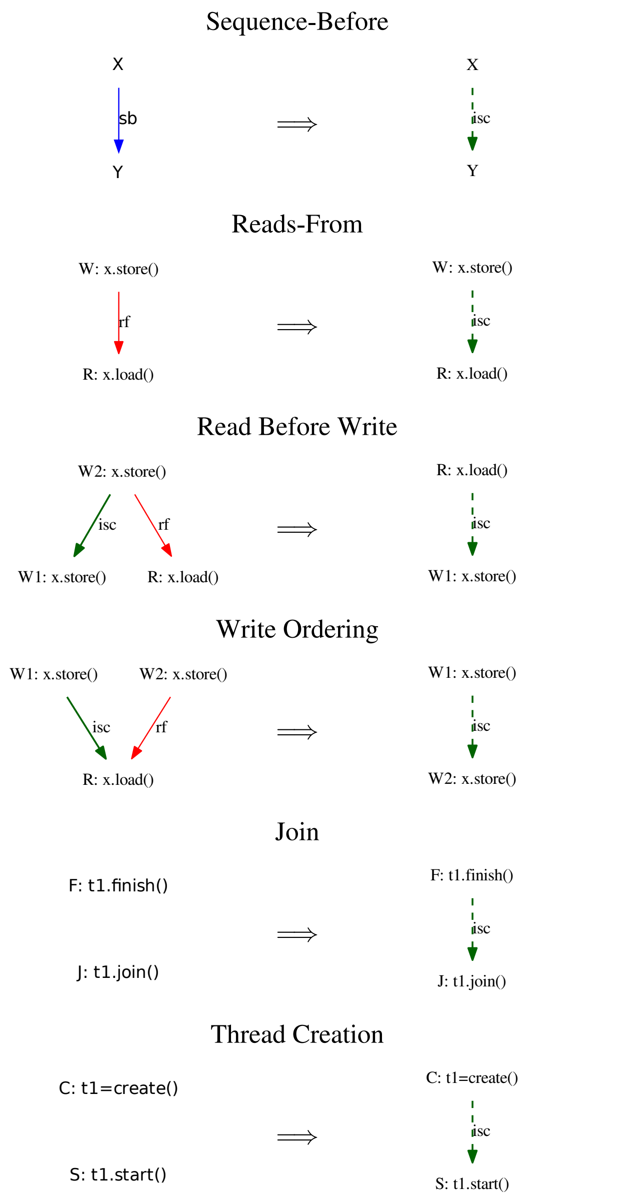


Figure 8. Implications for constructing the partial order $\xrightarrow{isc}$

```
Initially x=y=z=0.
T1:
 1: z.store(1, relaxed);
 2: x.store(1, relaxed);
 3: y.store(1, relaxed);
 4: r1=z.load(relaxed);//Reads from Line 1
T2:
 5: z.store(2, relaxed);
 6: x.store(2, relaxed);
 7: r2=x.load(relaxed); //Reads from Line 2
T3:
 8: z.store(2, relaxed);
 9: y.store(2, relaxed);
10: r3=y.load(relaxed); //Reads from Line 3
```

Figure 9. Example of a trace where search is required

```
 1: procedure CHECKSC
 2:     isc := {}
 3:     seq := {}
 4:     actions := Ops
 5:     return SEARCHSC(isc, seq, actions)
 6: end procedure

 1: function SEARCHSC(isc, seq, actions)
 2:     if actions = {} then
 3:         Output(seq)
 4:         return true
 5:     end if
 6:     isc=UPDATESC(isc)
 7:     if $isc = NULL$ then
 8:         return false
 9:     end if
10:     searchset = $\{a' \in actions \mid \neg\exists a'' \in actions.\langle a'', a'\rangle \in isc\}$
11:     for all $a \in searchset$ do
12:         seq' := seq ; a
13:         isc' := isc
14:         if $a \in StoreOp$ then
15:             for all $a'' \in actions \cap StoreOp$ do
16:                 if $\texttt{address}(a) = \texttt{address}(a'')$ then
17:                     isc' := isc' $\cup \{\langle a, a''\rangle\}$
18:                 end if
19:             end for
20:         end if
21:         if SEARCHSC(isc',seq',actions $\setminus\{a\}$) then
22:             return true
23:         end if
24:     end for
25:     return false
26: end function

 1: function UPDATESC(isc)
 2:     while $\exists$ a rule application $r$ that adds a new edge $e$ to isc do
 3:         if adding $e$ to isc does not create a cycle then
 4:             isc := isc $\cup \{e\}$
 5:         else
 6:             return NULL
 7:         end if
 8:     end while
 9:     return isc
10: end function
```

Figure 10. Algorithm for checking whether a trace is SC

operation, Lines 15 through 19 add edges to the $\xrightarrow{isc}$ relation between the current store operation a and all other store operations to the same location in the `actions` set. These edges may cause other inference rules to add additional edges to the $\xrightarrow{isc}$ relation. Line 21 then recursively calls the SEARCHSC procedure to reorder the remainder of the trace. If at any point, the $\xrightarrow{isc}$ relation contains a cycle the search algorithm backtracks.

If the $\xrightarrow{isc}$ relation does not uniquely specify the next operation to add to `seq`, the algorithm uses backtracking-based search to explore all possibilities for the next operation using the loop in Line 11.[3]

[3] Note that the algorithm can be further optimized to prune from the search (1) all loads and (2) stores that are ordered before all other stores to the same location.

The partial order $\xrightarrow{isc}$ can be implemented using clock vectors for efficiency [34]. Our implementation takes this approach. A clock vector is a map from threads to the operation identifiers. More precisely, if the clock vector for operation o contains the operation identifier o_t for thread t, then o_t appears before o in the $\xrightarrow{isc}$ relation. We prove the correctness of the algorithm in Appendix A.

5.2 Discussion

The constructions used for complexity proofs in [24, 25] are somewhat contrived in that they blindly perform stores to the same memory location without any mechanism to prevent conflicts that cause backtracking. Real-world concurrent data structures often either: (1) use RMW operations to update a single location thus implicitly ordering the RMW operations or (2) use another mechanism typically involving a load to make sure that no other thread will perform a conflicting store and thereby establish an order for the stores. Thus our rules for generating the $\xrightarrow{isc}$ relation were able to infer a strong enough ordering to avoid backtracking for our benchmark concurrent data structures.

6. Simplifying Non-SC Traces

In previous sections, we have described an algorithm for checking whether a trace is SC and proved its correctness. However, in addition to checking whether a trace is SC, it can be helpful to present a non-SC trace as mostly SC and then mark the parts of the trace that violate SC. In our experience, understanding a trace with a handful of reads that violate SC is far easier than understanding a trace where loads appear to read from almost arbitrary stores (e.g. the raw traces produced by CDSChecker) because simplified traces can save developers from jumping all over the traces while reasoning about the cause of the SC violations.

Recall the example trace and the $\xrightarrow{isc}$ graph from Figure 6 and Figure 7. Although there exists a cycle in the $\xrightarrow{isc}$ partial order, if we reorder the original trace by dropping the $\xrightarrow{isc}$ edge from Operation 5 to Operation 4, we produce an execution trace in which only Operation 5 reads from an old value (showed in Figure 11). We can see that such a reordered trace can be easier to understand than the original trace with two messy violations (Operation 2 and 5). More importantly, by simplifying the trace, we can see that it now blames only one SC violation (Operation 5) which is the correct SC violation in the trace.

#	Thread	Operation	Order	Addr	Value	rf
1	1	atomic store	relaxed	0x2080	0x1	
3	1	atomic store	relaxed	0x6020	0	
4	1	atomic store	relaxed	0x6028	0x1	
7	1	atomic store	relaxed	0x5c08	0x6020	
2	2	atomic load	relaxed	0x5c08	0x6020	7
5	2	atomic load	relaxed	0x5c08	0	0
6	2	atomic load	relaxed	0x2078	0	0

Figure 11. Reordered trace to be mostly SC.

6.1 Approach to Reordering Trace

Our first naïve approach to this problem was to modify the algorithm to allow the $\xrightarrow{isc}$ relation to have cycles and then topologically sort the $\xrightarrow{isc}$ relation into strongly connected components (SCCs). We initially attempted this approach and it has two problems: (1) the inference rules typically expand the cycles in the $\xrightarrow{isc}$ relation to cover operations completely unrelated to the actual SC violation and (2) a cyclic $\xrightarrow{isc}$ relation does not provide any ordering between operations in the same SCC leading to traces that arbitrarily (and confusingly) order operations.

6.2 Forcing *isc* to be Acyclic

Our next refinement was to modify the basic algorithm from Figure 10 to continue building the $\xrightarrow{isc}$ relation even after discovering that the $\xrightarrow{isc}$ relation contains cycles if there are no more backtracking points. However, the modified version never actually adds an edge to the $\xrightarrow{isc}$ relation to realize the cycle. The algorithm then prints out the execution sequence `seq` and flags any loads that read from a store other than the last prior store to the given location.

This approach generates a mostly SC execution trace and makes explicit the few violations of SC that are present. We found these traces were easier to understand because we could think of most of the trace as SC (i.e., most loads read from the last store to the same location, and loads that violate this property were clearly marked) and only focus on the few operations that actually violate SC.

6.3 Preserving *sc* and *hb*

Although this refinement significantly improves the output, it can still be confusing. Consider the common idiom — store buffering (SB) example code shown in Figure 12. In any SC execution, one of the stores in either Line 3 or Line 1 must execute first, and the load performed by the other thread must see the value of that store. In the execution shown, both loads read from the initial values, and therefore it cannot be represented as an SC execution and the *isc* relation will contain a cycle.

If the algorithm breaks this cycle by dropping the *isc* edge from the sequentially consistent load in Line 2 to the sequentially consistent store in Line 3, it will generate the reordered trace in Figure 13.

If a developer looks at this trace with the goal of making the code only have SC executions, it can be very confusing because it shows the sequentially consistent load from `y` in Line 2 returning an old value. However, the load and store to `y` have the `memory_order_seq_cst` memory order and the initial store to `y` happens before the load and the store. So it is not possible to strengthen the operations on `y` and indeed the problematic behavior arises due to the operations on the variable `x`.

[5] The highlighted row indicates that the load operation reads from a store that was not the last store operation in the trace to that location.

```
Initially x=y=0.
T1:
 1: x.store(1, relaxed);
 2: r1=y.load(seq_cst);//Reads from initial value.
T2:
 3: y.store(1, seq_cst);
 4: r2=x.load(relaxed);//Reads from initial value.
```

Figure 12. Code for confusing SC example (store buffering)

#	Thread	Operation	Order	Addr	Value	rf
3	2	atomic store	seqcst	0x60a8	1	
4	2	atomic load	relaxed	0x60a0	0	init
1	1	atomic store	relaxed	0x60a0	1	
2	1	atomic load[5]	seqcst	0x60a8	0	init

Figure 13. A reordered trace for the confusing SC example

```
Initially x=y=0.
T1:
 1: r1=x.fetch_add(2, relaxed);//Reads from Line 4
 2: y.store(1, relaxed);
T2:
 3: r2=y.load(relaxed);//Reads from Line 2
 4: r3=x.fetch_add(2, relaxed);//Reads from Line 6
T3:
 5: r4=x.load(acquire);//Reads from Line 1
T4:
 6: x.store(1, release);
```

Figure 14. Code for confusing *hb* example

#	Thread	Operation	Order	Addr	Value	rf
1	1	atomic rmw[5]	relaxed	0x1060	3	4
6	3	atomic load	acquire	0x1060	5	1
2	1	atomic store	relaxed	0x1064	1	
3	2	atomic load	relaxed	0x1064	1	2
5	4	atomic store	release	0x1060	1	
4	2	atomic rmw	relaxed	0x1060	1	5

Figure 15. A reordered trace for the confusing *hb* example

Figure 14 presents an example involving happens before that yields a confusing trace. The modification orders for the add operations on `x` and the store and load operations on `y` in Threads 1 and 2 are not compatible. The SC analysis processes the operations on `y` first and adds a corresponding *isc* edge. This prevents adding the *isc* edge to `x`. The end result is the generation of the trace shown in Figure 15 that does not respect happens-before — the load `acquire` appears before the corresponding store `release`.

The problems in the two examples arise because the *isc* relation does not necessarily contain all of the *sc* or *hb* edges from the original execution. A key question is whether a reordered trace will always respect both the original *sc* and *hb* edges. Since *hb* is a subset of the transitive closure of *rf* and *sb* (plus thread joins and mutexes), a reordered trace will always respect *hb*.

The *sc* order presents a challenge — a reordered trace may not always respect the *sc* order as non-sequentially consistent loads and stores can interact with sequentially consistent operations. However, since *sc* is only indirectly observable via the *rf* behavior, there is no need for *isc* to be strictly consistent with *sc* for SC traces because an alternative *sc* order may produce a reordered trace that is consistent with the observed *rf* behaviors. However, as shown in Figure 12, if *isc* is cyclic we do not want to produce a trace where SC operations are falsely blamed for the non-SC behavior. Thus our approach is to add all *hb* edges to the *isc* partial order and to prioritize *sc* edges such that the *isc* inference rules add the *sc* edges to the *isc* partial order before adding any other edges.

7. Inferring Memory Order Parameters

Under the C/C++11 memory model, inferring the order parameters to obtain SC behaviors is essentially a search problem. In the absence of consume operations, memory order parameters for atomic operations can be only one of the following: `memory_order_relaxed`, `memory_order_release`, `memory_order_acquire`, `memory_order_acq_rel` and `memory_order_seq_cst`. A naïve approach that enumerates all possible memory order parameters is guaranteed to discover all the possible inferences of parameters that ensure SC behaviors for a specific test case. However, this approach obviously leads to an impractical exponential search space. Fortunately, there exist heuristics for strengthening parameters for the purpose of only admitting SC behaviors. These heuristics may not always achieve the optimal memory order parameters, but they are guaranteed to repair any SC violation. AutoMO uses a search-based approach combined with heuristics to fix the non-SC behaviors to reduce the search space. Figure 16 shows the core search algorithm.

The idea of our algorithm is that AutoMO iteratively infers parameters test case by test case. It takes an optional input of parameter assignments. If no input is provided, AutoMO begins the inference process by setting all order parameters to `memory_order_relaxed` (Line 4). For each test case, AutoMO maintains a set of possible inferences initialized by the input parameter assignments (Line 7), and for each potential candidate, it uses CDSChecker to explore traces and applies the SC analysis algorithm to check whether there exists any non-SC trace. If so, AutoMO calls the function `StrengthenParam` (Line 13) to find out potential repairs and insert them to the `candidates` set. It is possible for a given repair to be made redundant. Thus, AutoMO calls the `WeakenOrderParams` routine (Line 15) to find out potential weaker results. We discuss later how AutoMO strengthen order parameters (Line 24) in Section 7.1, and the `WeakenOrderParams` routine in Section 7.3. Note that AutoMO uses the temporary output as the input for the next test case (Line 18). After exploring all test cases, AutoMO returns the final inference results (Line 20).

The challenge then becomes the following: given an SC violation in a reordered trace, how can we discover potential repairs that eliminate the violation? For example, with the reordered trace in Figure 11, we can prevent the SC violation

```
function INFERPARAMS(testcases, initialParams)
    inputParams := initialParams
    if inputParams is empty then
        inputParams := the weakest parameters
    end if
    for all test case $t$ in testcases do
        candidates := inputParams
        results := {}
        while candidates is not empty do
            Candidate $c$ := pop from candidates
            run CDSChecker with $c$ and check SC
            if $\exists$ SC violation $v$ then
                STRENGTHENPARAM($v$, $c$, candidates)
            else
                results += WEAKENORDERPARAMS($c$)
            end if
        end while
        inputParams := results
    end for
    return results
end function
procedure STRENGTHENPARAM(v, c, candidates)
    while $\exists$ a fix $f$ for violation $v$ do
        possible_repairs := strengthen $c$ with fix $f$
        candidates += possible_repairs
    end while
end procedure
```

Figure 16. Algorithm for inferring order parameters

in Operation 5 by specifying `memory_order_release` in Line 19 and `memory_order_acquire` in Line 25 of the SPSC example shown in Figure 5. Thus, when the load of the `next` field reads its value from the corresponding store of the `next` field, it establishes a *happens-before* relationship (release/acquire synchronization), eliminating the possibility of Operation 5 reading from an uninitialized value. We next discuss how AutoMO strengthen parameters.

7.1 Inference Rules

As discussed above, when AutoMO discovers a non-SC trace for a test case, the goal of the inference algorithm is to figure out a weakest strengthening to the memory order parameters that will disallow those traces. A *weakest strengthening* for a test case is a strengthening of parameters that disallows the non-SC behavior but will admit some non-SC behavior if any parameter instance of that strengthening is weakened. Note that since the parameter assignments are finite, a weakest strengthening must exist but is not necessarily unique.

This problem can be viewed as detecting and eliminating the cycles in $\xrightarrow{isc}$. The first step is to discover which atomic operations are responsible for cycles in $\xrightarrow{isc}$. In Section 6, we show that our analysis can reorder a non-SC trace to preserve *sc* and *hb* while it indicates where a bad reads-from edge happens, and hence we can use the simplified trace to discover the non-SC violations in the original trace.

Figure 17 shows the two universal patterns that cover all non-SC behavior in reordered traces. The *Stale Read* Pattern covers the case in which a load takes its value from an old store rather than the most recent store in the reordered trace. The *Future Read* Pattern covers the case in which a load takes its value from a store that is ordered after the load.

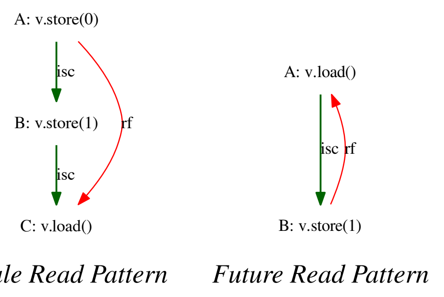


Figure 17. Cycle patterns for non-SC behaviors

Fortunately, for either pattern of SC violation, two repair approaches exist: 1) eliminating the *reads-from* edge; or 2) strengthening memory order parameters to reorder the trace in a new way.

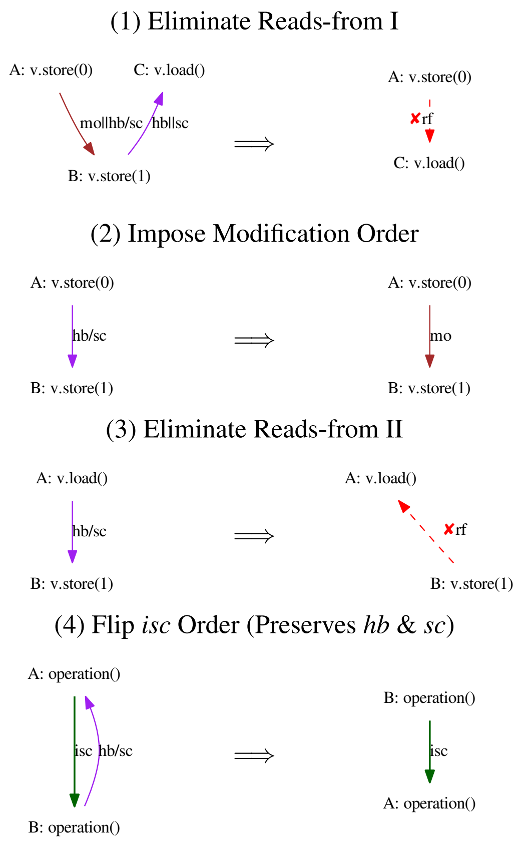


Figure 18. Inference rules for non-SC traces

Figure 18 presents a set of rules that we can use to strengthen memory order parameters for memory accesses, and Appendix B describes the rules to support fences. Each row means if the operations satisfy the condition on the left, then the property on the right holds. We denote *mo* as the *modification-order* relation, *hb* as the *happens-before* relation, *sc* as the total order of operations with `memory_order_seq_cst` order, *rf* as the *reads-from* relation and *isc* as the order in the reordered trace. For example, Rule 1 means that if: 1) operation A is *modification order* before operation B, and operation B *happens before* C; or 2) operation A *happens before* or is *sc* before operation B, and operation B is *sc* before operation C, then it ensures that operation C is not allowed to read from operation A. Note that due to partial order reduction in CDSChecker, the *modification order* relation is a subset of the union of the *happens-before* and *sc* relation.

We present the detailed reasons why each rule holds in case readers are interested. We derive Rule 1 from §1.10p18 [3] (write-read coherence) and §29.3p3 [3] (SC constraint on loads), Rule 2 from §1.10p15 [3] (write-write coherence) and §29.3p3 [3] (SC constraint on *mo*), and Rule 3 from §1.10p17 [3] (read-write coherence) and §29.3p3 [3] (SC constraint on loads). For Rule 4, the *isc* edge specifically means the order in the reordered trace. As discussed in Section 6, if operation A is ordered before B in the reordered trace, and we can enforce either *hb* or *sc* edge from B to A, AutoMO will flip the order of A and B since the reordered trace is guaranteed to preserve *hb* and *sc*.

AutoMO searches on the rule applications to generate memory order assignments that repair the SC violation. We next discuss some cases of how AutoMO can apply these rules to iteratively eliminate the SC violations for the two patterns shown in Figure 17.

Stale Read Pattern: For this pattern, we have a load operation C that reads from store operation A, and there exists at least one store operation B that is between A and C in the reordered trace. Note that A, B, and C are operations on the same memory location. Thus, we can apply Rule 1 by imposing *mo* from A to B and imposing either *hb* or *sc* from B to C such that C is no longer allowed to read from A. If A is not *modification order* before B, we then apply Rule 2, which leads to imposing either *hb* or *sc* from A to B. Therefore, we can end up imposing a combination of either *hb* or *sc* from A to B and from B to C. If between A and C there exist other store operations (B') that are on the same memory location as B, AutoMO applies the same rule on them and potentially generate more possible repairs.

Future Read Pattern: For this pattern, we have a load operation A and a store operation B that is ordered after A in the reordered trace, and A reads from B. AutoMO has two ways to repair this violation: 1) eliminate the reads-from edge by applying Rule 3, meaning that it imposes either *hb* or *sc* edge from A to B if possible; 2) flip the order of A and B in the reordered trace by applying Rule 4 such that the reordered trace does not blame this exact same violation again. By flipping the order of A and B, it can either fix the violation or expose another new violation.

These rules boil down to strengthening specific *isc* edges to either *hb* or *sc*. Without considering the C/C++ fences, imposing *sc* requires both operations to have stronger parameters, i.e., `memory_order_seq_cst`, than imposing *hb*. Imposing *hb* between Operations A and B requires that there exist a path in the graph of *reads-from* and *sequence-before* edges from A to B. We then either 1) strength according to the definition of release sequence (§1.10p7 [3]) if possible; or 2) strengthen store and load operations along this path to `memory_order_release` and `memory_order_acquire`. Whenever not necessary, AutoMO does not impose the stronger *sc* order between operations.

7.1.1 Correctness of Repair Approaches

Although our repair approach is limited to the test cases provided, it does guarantee that the executions of the given test cases are SC. Each time AutoMO detects an SC violation, the violation is visible in the trace as one of the two patterns shown in Figure 17.

For the *Stale Read* pattern, we have the blamed load (operation C), the store it reads from (operation A), and the last preceding store (operation B) to the same memory location. According to the semantics of the C/C++11 memory model, the following two conditions cannot hold at the same time: 1) A is *modification order* before B; and 2) B is *sc* before C or *happens before* C. If the first condition is false, the *hb* and *sc* relation between A and B must not have been established (since the *modification order* relation must be consistent with the *hb* and *sc* relation), and thus AutoMO will be able to apply Rule 2 to enforce the *hb* or *sc* relation between A and B. If the second condition is false, AutoMO will be able to apply Rule 1 to enforce *hb* or *sc* between B and C.

For the *Future Read* pattern, we have the blamed load (operation A) and the future store (operation B). The semantics of the C/C++11 memory model requires that A does not *happen before* B and that A is not *sc* before B. Thus, AutoMO will be able to apply Rule 3 to enforce *hb* or *sc* between A and B to eliminate that reads-from edge. Also, as the reordered trace preserves the *hb* and *sc* relation, B does not *happen before* A and is not *sc* before A in such a trace. Therefore, AutoMO will also be able to apply Rule 4 to enforce *hb* or *sc* between A and B.

Since the reordered trace is consistent with both *hb* and *sc*, it is guaranteed that repair rules can always be applied to the memory operations that have been for the SC violation (and that after the repair actions are fully performed that the same operations cannot be repetitively blamed for the SC violation).

As a result, AutoMO's repair actions will always strengthen some memory order parameter in each rule application. Since there are a finite number of memory order strengthenings before all memory parameters become `memory_order_seq_cst` (and the trace becomes trivially SC), this process must terminate. Since the algorithm runs on all provided test cases incrementally, it provides SC for all provided test cases.

7.1.2 Different Parameter Assignments

As shown above, there can exist different parameter assignments that provide SC for the given test cases. AutoMO keeps track of the parameter assignments that can be generated by applying the inference rules, and outputs those that are not strictly stronger than others. It can become very complex to consider the run-time effects of different parameter choices because they differ from platform to platform and may even depend on the choice of compiler. For example, under x86, operations naturally have the release/acquire semantics, so a relaxed load has minimal advantage over an acquire load.

7.2 Normal Memory Accesses

We use CDSChecker as the underlying model checker to check for non-SC traces and it differentiates between normal memory access and atomic memory access. A key aspect of assigning memory orders to atomics is to ensure that normal memory accesses do not race. As we do not require any compiler frontend that could be used to instrument these accesses, we need the developer to manually instrument the accesses. We solve the instrumentation problem as follows. Developers expose non-atomic accesses to our inference tool by using special wrapper functions, which are similar to C/C++11 atomic operations with a special parameter `memory_order_normal`. For example, the statement "`x = 1`" would be rewritten as "`x.store(1, memory_order_normal)`". Our tool then ensures that we establish synchronization between conflicting normal memory accesses. This instrumentation could conceptually be performed automatically at the cost of requiring the developer to use a specific compiler frontend.

7.3 Weakening Memory Order Parameters

Although iteratively applying the inference rules to fix SC violations eventually infers order parameters that guarantee SC for the corresponding test case, we may end up inferring overly strong parameters. This can happen if a later repair attempt makes the initial repair unnecessary. We solve this issue by introducing a routine to weaken order parameters.

More specifically, after inferring a preliminary result for the test case, we explore all possible parameter assignments that are strictly weaker than that result while no weaker than the corresponding input parameter assignment for that test case. If there exist any weaker parameter assignments that are SC, we weaken the preliminary result. Note that the complexity of the weakening process for a test case is independent of the total number of parameters to be inferred but only depends on the strengthened parameters, whose number in our experience is small.

7.4 Allowing Non-SC Behaviors

While most data structures are internally SC, in some cases it can be desirable to allow a few controlled SC violations. In our experience, a few data structures have SC violations that do not affect correctness because the data structures detect SC violations and retry the operation. Such code snippets often occur in spin loops that perform a CAS operation on exit. In this case, we only need to check that the other operations have SC behaviors to ensure correctness.

Therefore, we provide a simple annotation framework in AutoMO that users can use to specify the region of code that allows SC violations. We then extend the SC analysis algorithm such that it does not infer $\xrightarrow{isc}$ edges for load operations that are allowed to violate SC, and extend the parameter inference algorithm such that it does not repair allowed SC violations in the reordered trace. However, if those loads can be strengthened to prevent other SC violations, namely introducing cycles elsewhere, AutoMO may strengthen the parameters of such loads to to eliminate cycles. Note that this mechanism mainly provides means for more advanced users to provide more information to obtain further optimizations. Novices may very well not use this functionality and will simply obtain an SC implementation.

7.5 Implementation of the Inference Framework

As discussed above, AutoMO requires a model checker that can exhaustively enumerate executions allowed by the C/C++11 memory model and output a trace with the reads-from mapping and the *sc* and *hb* relations. We implemented AutoMO as a backend analysis of CDSChecker. We extended the memory order parameters to support special wildcard parameters to indicate which parameters AutoMO should infer. To use AutoMO, instead of using a concrete memory order parameter, a developer writes a C/C++ atomic operations with a special wildcard memory order. For example, a load operation, which requires one actual memory order parameter, can be written as "`x.load(wildcard(1))`", to indicate the load operation uses the first wildcard parameter. A given wildcard parameter should only be used for one atomic operation, e.g., the next operation would use `wildcard(2)`. After executing given test cases, AutoMO outputs a set of assignments to the wildcards that ensure all executions are equivalent to SC. Developers can then run a script that automatically replaces the wildcards with the corresponding inferred parameters.

8. Evaluation

In this evaluation, we focus on three aspects of AutoMO: 1) how efficient is our algorithm? and 2) how do the results compare to the manual versions of the data structures? and 3) as a component of AutoMO, how efficient is the SC analysis

Benchmark	# Wildcard	Inference time (sec)
Chase-Lev	40	536.322
SPSC	7	0.015
Barrier	5	0.019
Dekker	12	396.756
MCS lock	9	4.056
MPMC	8	0.143
M&S queue	20	4.808
Linux RW lock	16	24.982
Seqlock	8	0.095
Concurrent hashtable	13	0.016
Treiber stack	8	0.018

Figure 19. Benchmark results of inference algorithm

algorithm (since it can also be useful for debugging as a separate part)?

We have implemented our algorithm as an analysis plugin for the CDSChecker model checker, and ran our experiments on an Ubuntu 14.04 Linux machine with an Intel Core i7 3770 processor.

To test our algorithm on real-world code, we used CDSChecker's benchmark suite along with three additional benchmarks. The benchmark suite includes six data structure implementations—a synchronization barrier, a mutual exclusion algorithm, a contention-free lock, and two different types of concurrent queues, and a work stealing deque [28]. Additionally, the benchmark suite contains a port of the Linux kernel's reader-writer spinlock from its architecture-specific assembly implementation and the Michael and Scott queue from its original C and MIPS source code [38]. We have added three additional benchmarks — a seqlock, a concurrent hashtable, and the Treiber stack.

8.1 Performance of Inference Algorithm

Figure 19 presents the results of the inference algorithm. The second column shows the number of operations that require inference, and the third column shows the time taken to finish the inference for each benchmark in seconds. We can see that 8 out of our 11 benchmarks finish within 5 seconds, and the benchmark that takes the longest time (Chase-Lev Deque) finishes with 536.322 seconds (less than 9 minutes). These results show that our inference algorithm is efficient and can finish in a reasonable amount of time for real-world data structures.

8.2 Inference Results Compared to Manual Versions

In this section, we briefly describe each data structure, our test clients for both the inference algorithm and the SC analysis, and the inference results. Due to the absence of formal techniques that can prove that the data structures only exhibit SC behaviors under any execution, we manually review the correctness of the inference results.

Chase-Lev Deque: This implementation was taken from a peer-reviewed, published C11 adaptation of the Chase-Lev deque [28]. It utilizes relaxed operations (for efficiency) while utilizing fences and release/acquire synchronization to establish order. While the paper proves that an ARM implementation is correct, it does not contain a correctness proof for its C11 implementation. A bug was discovered in the published version in the deque resize implementation [40].

This benchmark has three API methods: `push`, `take` and `steal`, and we use 6 test clients to infer the parameters as follows: 1) there is a main thread with one `push` method call and a stealing thread with one `steal` method call; 2) the deque is initialized with three items, and a main thread (with one `take` method call) and a stealing thread (with two `steal` method calls) race for the elements; 3) the deque is initialized with three items, and a main thread (with one `take` method call) and two stealing threads (each with one `steal` method call) race for the elements; 4) a main thread has three `push` and two `take` method calls, and a stealing thread with one `steal` method call; 5) the deque is initialized with one item, and a main thread with one `take` method call and a stealing thread with one `steal` method call race for it; and 6) the deque is initialized with one element, and a main thread has one `take`, `push` and another `take` method calls along with two stealing threads that each have one `steal` method call. We also use the fourth test client of the above six to test the SC analysis for both the original buggy version and a bug fixed version of the Chase-Lev Deque.

We ran AutoMO on these test clients. Upon manual review, the inferred result appears to be correct. The inference result requires three stronger parameters, one load to be `memory_order_acquire` and two stores to be `memory_order_release`. Our result also infers two weaker parameters as `memory_order_relaxed`, one is a load with `memory_order_acquire` and the other is a CAS with `memory_order_seq_cst` in the original paper. The interesting difference is that we inferred that the load operation right before a fence (`memory_order_seq_cst`) should be `memory_order_relaxed` rather than `memory_order_acquire`. The stronger `memory_order_acquire` parameter in [28] is redundant because the fence already suffices to generate the necessary synchronization. We contacted the paper's authors and they confirmed that the stronger parameter is not necessary and that they believe AutoMO's version to be correct.

This shows that AutoMO can be useful in practice by inferring close enough parameters since in less than 10 minutes it can infer suitable assignments for a data structure whose porting effort justified a research paper.

SPSC queue: The single-producer, single-consumer queue allows concurrent access by one reader and one writer [7]. In the CDSChecker's benchmark suite, there are two versions, a buggy version and a bug-fix version. Both implementations utilize methods `signal` and `wait` to communicate between enqueuers and dequeuers, and the buggy version can potentially miss a signal. We use a test client with two threads—one to enqueue a single value and the other to dequeue it and verify the value.

We ran AutoMO on the buggy version, and AutoMO inferred an assignment in which four operations were memory_order_seq_cst. The bug-fix version fixed the bug by changing a plain load with memory_order_relaxed to a fetch_add operation with memory_order_seq_cst, and it also has four operations with memory_order_seq_cst. We reviewed the buggy version and found that our inference result is the optimal way to repair the bug without changing the operations because the stronger parameters (memory_order_seq_cst) are necessary to eliminate the possibility of missing a signal.

Barrier: Barrier 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 test client utilizes two threads with a non-atomic shared memory operation executed on either side of the barrier, one in each thread.

This implementation utilizes memory_order_seq_cst order for 5 operations. However, our results show that we only require one operation to be memory_order_release, one operation to be memory_order_acquire and one operation to be memory_order_acq_rel. After reviewing the code, we found that the original manual choice of order parameters were overly strong and our inference result is sufficient to ensure the correctness property that the invocation of each wait method *happens before* the response of any other wait method. The reason is as follow:

For the first $N-1$ threads, they perform a fetch_add on the variable nwait (the number of currently waiting threads) with the parameter memory_order_acq_rel, and thus threads T_i and T_j ($1 \leq i < j \leq N$) establish synchronization. The last thread increments the variable step (the number of barrier synchronizations completed so far) by performing a fetch_add (memory_order_release) operation, and then the first $N-1$ threads leave its spinning loop by loading the updated value from step (memory_order_acquire). Therefore, the last thread also synchronizes with the first $N-1$ threads. As a result, all participating threads synchronize with each other.

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.

For the parameter inference, we use two clients: 1) two identical threads that update a normal memory location for once; and 2) one thread updates a normal memory location for once, while the other thread updates that location twice. We also use the first one to test the SC analysis. AutoMO infers the exact same result as the manual versions for this benchmark.

MCS lock: This contention-free lock implements the algorithm proposed by Mellor-Crummey and Scott (known as an *MCS lock*) [5, 36]. The lock queues waiting threads like a concurrent queue in a linked-list fashion. In the test client, we use two threads, each of which alternates between reading and writing the same variable, releasing the lock in between operations. We use a test driver with two threads, each of which alternates between reading and writing the same variable, releasing the lock in between operations.

AutoMO infers two strictly weaker parameter assignments than the original manually annotated benchmark. There are two operations with memory_order_acquire in the original parameter assignment, while our two inference results only require one of the two parameters to be memory_order_acquire. After careful review, we found both of our parameter assignments are correct, and the reason is as follows:

When a thread, followed by waiting threads, releases the lock, it sets the gate field of the next waiting node to let it acquire the lock. In order for that waiting thread to see the the update-to-date value of the gate field, that lock method must synchronize with the unlock method. Our review discovered that in the unlock method, either of the load of the m_tail variable or the load of the next field can be assigned the memory_order_acquire order to establish the synchronization. As a result, AutoMO infers two parameter assignments, meaning that the manual version was overly strong and AutoMO infers better results.

MPMC queue: This multiple-producer, multiple-consumer queue allows concurrent access by multiple readers and writers [6]. Note that the original implementation admits non-SC traces due to retries. We use a test client that runs two identical threads, each of which first enqueues an item and then dequeues an item.

This benchmark has eight memory operations, and AutoMO infers stronger parameters (memory_order_seq_cst) for four of them. Although our inferred parameters are not ideal in this case, AutoMO can still help users who do not have intimate knowledge about the C/C++ memory model because compared to the naïve approach of specifying memory_order_seq_cst parameter for all operations, our inference result infers four weaker parameters.

M&S queue: This benchmark is an adaptation of the Michael and Scott lock free queue [38] to the C/C++ memory model. The port uses relaxed atomics when possible. We used the following 3 test clients for this benchmark: 1) two threads each with one enqueue method call and one thread with one dequeue method call; 2) one thread with one enqueue method call and two threads each with one dequeue method call; and 3) one thread has one enqueue method call while the other thread calls the methods enqueue, dequeue, and enqueue in order. The third test client covers the scenario where a node is dequeued, recycled, and enqueued back to the queue again. We used the first one to evaluate the SC analysis.

We ran AutoMO with the above three test clients, and it infers the same parameters as the manual version except that two operations have stronger parameters, which are `memory_order_acquire` and `memory_order_release` rather than the original `memory_order_relaxed`. After careful review, we found that both are bugs in the original benchmark, and our result repaired them by imposing the stronger parameters. We briefly explain the two bugs as follows.

One bug exists in the `dequeue` method, in which synchronization is not established for the load of the `tail` so that the load of the `next` field of the `head` can read an out-of-date value (e.g. `NULL`), and that can potentially return arbitrary values. Another bug is in the `enqueue` method, where an initialized node is inserted into the queue without proper synchronization on the `next` field (pointing to the next node).

Linux reader-writer lock: A reader-writer lock allows either multiple readers or a single writer to hold the lock at any time—but no reader can share the lock with a writer. The test client for the parameter inference has two threads that use lock, trylock and unlock to protect the read/write of normal memory accesses. Specifically, the test client for the SC analysis has two threads where one reads the variable under a reader lock, and the other writes to the variable under the protection of a writer lock. AutoMO infers the exact same result as the manual version for this benchmark.

Seqlock: Seqlocks are used in Linux to avoid writer starvation. We implemented it with C/C++11 atomics and utilized relaxed operations when possible. We run two writing threads and one reading thread.

We then ran the inference algorithm on Seqlock and obtained two possible inference results, one of which has the exact same parameters as the original. The other one has just one different `acquire` load operation. After careful review, we found that both results are correct for the following reason. In the `write` method, it spins by loading from the `seq` variable (the global sequence number) and trying to increment it with a CAS operation if the value of the `seq` variable is an even number. We can use either operation to establish synchronization between `write` methods, and the manual version is just one option of the two. However, AutoMO provides both options.

Concurrent hashtable: We ported a concurrent hashtable implemented in Java by Doug Lea from [29] to C/C++. For the sake of simplicity, we only consider the fundamental primitive API methods `put` and `get`. We use two test clients for the parameter inference: 1) two threads update two different keys and then look up the keys updated in another thread; and 2) same as the first test client except that the updated keys have been initialized. Also, we use the first test client to test the SC analysis.

Initially, AutoMO infers stronger parameters than needed for this benchmark. The result requires a stronger parameter (`memory_order_seq_cst`), while it only needs `memory_order_acquire` to establish proper synchronization to ensure it reads the entire list of entries for each bucket. By reviewing the code, we found that even if the load of the first element of the entry list reads an old value, it does not affect correctness since it is fixed by a later clean-up routine. We then leverage the annotation framework provided by AutoMO to annotate that this operation is allowed to have an SC violation.

By adding this information, AutoMO can infer the weakest order parameters that guarantee correctness. It infers `memory_order_seq_cst` for the load and store of the `value` variable, ensuring that the clients observe SC behaviors when there are two threads that each update one key and look up for the other key. It infers the `acquire` semantics for the load of the first element of the list, ensuring that it obtains an intact list. This shows that AutoMO can be useful for data structures that allow non-SC behaviors if users can provide the knowledge of which SC violations are tolerable.

Treiber stack: The Treiber stack [47] allows concurrent push/pop operations in a non-blocking fashion. We implemented it to the C/C++11 memory model and tested it with three threads, two of which push an item and one of which pops an item. AutoMO infers the exact same result as the manual versions for this benchmark.

8.3 Writing Test Clients

To effectively use AutoMO, developers need to provide test cases that fully exercise the data structure, and here we discuss our insights for writing these test cases. In general we started with clients that generate a variety of small normal usage scenarios. We then examine the data structure to determine potential corner cases (e.g., for a deque these might include resizes, operations on empty deques, and operations that race for the last element) and write test cases to exercise these behaviors. Although writing test cases is not trivial, developers often have to at least consider these cases in the design of the original SC data structure and thus have a good intuition for potential corner cases.

8.4 Performance of SC Analysis

As discussed above, the SC analysis alone can be useful for developers to understand traces. In order to evaluate the performance of our SC analysis, we ran the SC checking algorithm alone in AutoMO in this section.

8.4.1 Results

Figure 20 presents the results. For each benchmark, we record the total number of executions whose behavior was consistent with the memory model (# Feasible), the number of those traces that were not SC, the time taken to run the SC analysis for all feasible traces, the total model checking

Benchmark	# Feasible	# Non-SC	SC Analysis time (s)	Total time (s)	Avg. Trace Length	SC Analysis time per trace (s)
Chase-Lev (buggy)	65	24	.0037	.11	68	5.7×10^{-5}
Chase-Lev (correct)	49	1	.0017	.04	75	3.5×10^{-5}
SPSC (buggy)	10	2	.0006	.01	26	5.7×10^{-5}
SPSC (correct)	15	0	.0008	.01	29	5.2×10^{-5}
Barrier	7	0	.0004	.01	23	5.8×10^{-5}
Dekker	2,313	0	.0756	8.27	52	3.3×10^{-5}
MCS lock	12,609	0	.5767	4.08	65	4.6×10^{-5}
MPMC queue	11,306	6,764	1.0497	9.09	49	9.3×10^{-5}
M&S queue	114	0	.0051	.06	55	4.4×10^{-5}
Linux RW lock	1,348	0	.0325	11.84	30	2.4×10^{-5}
Seqlock	9,124	0	.2669	2.91	38	2.9×10^{-5}
Concurrent hashtable	66	11	.0051	0.02	89	7.7×10^{-5}
Treiber stack	29	0	.0013	0.02	54	4.5×10^{-5}

Figure 20. Benchmark results. Note that all of the non-SC traces for MPMC are due to retries. When restricted to yield-free executions, MPMC only exhibits SC executions.

time, and the average trace length. The key points are that the time taken to check whether traces are SC is a small fraction of the total model checking time for all of our benchmarks.

Although checking whether traces are allowed under the SC memory model is NP-complete, in practice the algorithm ran very fast. In fact the SC analysis never even backtracked on any of our benchmarks or our set of litmus tests. The only code for which we have observed backtracking is a specific test case that implements the variable setting component used in a proof that the problem is NP-complete. It appears that for real-world code, the search component of the algorithm (backtracking) is typically not utilized and hence the runtime of the algorithm is typically polynomial.

8.4.2 Longer Traces

The benchmarks we exhaustively tested have fairly short traces. Our algorithm primarily targets helping developers unit test and debug concurrent data structures implementations and thus we only expect it to see relatively short traces. To test our algorithm on longer traces, we modified CDSChecker to produce a fixed number of traces and modified the SPSC queue to repeatedly enqueue and dequeue. We then varied the number of repeats to generate traces of varying lengths.

By performing 500 repeated enqueues and dequeues, we generated traces with an average of 5,945 operations and the SC analysis took an average of 0.0059 seconds per execution (averaged over 500 executions). By performing 900 enqueues and dequeues, we generated traces with an average of 10,645 operations and the SC analysis took an average of 0.0122 seconds per execution (averaged over 500 executions).

9. Related Work

Researchers have formalized the C++ memory model [9]. A number of tools have been developed to test the behaviors of C/C++ code under the C/C++ memory model. The CppMem tool is built directly from the formalized specification with the goal of allowing researchers to explore implications of the memory model. It explores all legal modification orders and reads-from relations and therefore must search a significantly larger search space than CDSChecker which only explores the space of legal reads-from relations. The Nitpick tool translates the memory model constraints into SAT problems and then uses a SAT solver to find legal executions [11]. The Relacy race detector [49] explores thread interleavings and memory operation reorderings for C++11 code. The CDSChecker tool [40] uses partial order reduction techniques to unit test C/C++ code. All of these tools would benefit from using the algorithm presented in this paper to present traces to the user. MemSAT [46] is designed to help debug and reason about relaxed memory models with axiomatic specifications. However, we cannot use these techniques to verify our inference results since they are also limited to the provided test cases and cannot verify that the data structures are robust under any execution.

A number of tools [20–22, 33, 45] have been developed to detect data races for programs that use lock-based concurrency, yet they do not extend to low-level atomics. As a complement to these tools, our work ensures that it can automatically infer memory order parameters that ensure data-race-freedom for provided test cases when developers provide sufficient atomic operations whose parameters can be strengthened.

Researchers have designed useful techniques [16, 19, 31, 44] for automatic parallelization. These techniques are primarily targeted to convert sequential code into parallel (multi-threaded or vectorized) code with the purpose of utilizing multiple processors. Our work is orthogonal in that it seeks to automate the process of tuning memory order parameters of C/C++11 data structures to provide SC.

Researchers have explored the complexity of checking whether a trace is SC in the context of testing shared memory implementations [18]. Earlier work established that the complexity of checking SC under the assumption that the reads-from mapping is known is NP-Complete [24]. Although our algorithm has polynomial complexity for traces it can handle without backtracking and we have not observed backtracking for our benchmarks, this result shows that there exist traces for which it has exponential time complexity.

Although the problem of verifying TSO is NP-hard, researchers have developed polynomial time techniques for approximately checking whether an execution is allowed by the TSO memory model [43]. In the absence of the full algorithm, our UPDATESC procedure presented in Figure 10 can be viewed as analogous polynomial-time algorithm for approximately checking whether a trace is SC.

Researchers have developed a nice property called triangular-race freedom (TRF) [41] that can precisely characterize programs with SC memory accesses on TSO. However, TRF is built upon TSO which has a total store order and allows only relatively few reorderings. The C/C++ memory model is much weaker and allows more reorderings, making an analogous result challenging. Besides, our algorithm checks a property closer to *resultSC*. [8] has developed formalization that checks SC assuming the modification order is known, which our algorithm does not rely upon.

Researchers have built tools to verify executions against axiomatic rules [26]. The approach taken by Gopalakrishnan *et al.* is to translate memory model axioms into a SAT problem instance and then to use a SAT solver to check whether the execution is consistent with the memory model.

CheckFence [15] explores executions of relaxed memory models. It does this by bounding loop iterations and translating the program's behavior into a SAT formula and solving the formula. This approach uses extensive static analysis of code to simplify the SAT formula.

Researchers have developed verification techniques for code that admits only SC executions under relaxed memory models such as TSO and PSO [14, 17, 23, 32]. The basic idea is to develop an execution monitor that can detect whether non-SC executions exist by examining only SC executions. Our work builds upon this work in two aspects: (1) it supports the C/C++ memory model; (2) it can still provide useful information even for code that admits non-SC executions; and (3) it automatically infers necessary order parameters to admit only SC executions. Researchers have also developed hardware support [35, 39, 42] for checking SC. Our approach differs in that it seeks to determine memory order parameters that suffice to ensure that a data structure only exhibits SC behaviors.

[32] presents a framework which can test a program on a given memory model to expose violations of a given specification, and synthesize a set of necessary ordering constraints that prevent these violations by leveraging a SAT solver. It only shows that its specification language can specify hardware memory models (TSO and PSO). However, the C/C++11 memory model is much more complicated than hardware models since it introduces more tricky ordering constraints (atomic operations and fences with memory order parameters) and allows more reorderings. Our work on the other hand not just supports the C/C++ memory model, but also provides useful information on non-SC traces for the purpose of understanding and debugging.

10. Conclusion

The C/C++ memory model makes it possible to write efficient, portable low-level concurrent data structure implementations. Many concurrent data structures are initially designed for the SC memory model, and porting them to the C/C++ memory model can be extremely challenging. We present AutoMO, a framework that provides support across the porting process: (1) it automatically infers initial settings for the memory order parameters, (2) it detects whether a C/C++11 execution is equivalent to some SC execution, and (3) it simplifies traces to make them easier to understand. We have evaluated AutoMO by using it to successfully infer memory order parameters for a range of data structures.

Acknowledgments

This project was partly supported by a Google Research Award and the National Science Foundation under grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786. We would also like to thank the anonymous reviewers for their helpful comments.

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] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. In *TOPLAS*, 2014.

[9] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[10] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[11] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *PPDP*, 2011.

[12] H. Boehm. Can seqlocks get along with programming language memory models? In *MSPC*, 2012.

[13] H. J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[14] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, 2008.

[15] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[16] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *CC*, 1986.

[17] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS*, 2011.

[18] J. F. Cantin, M. H. Lipasti, and J. E. Smith. The complexity of verifying memory coherence and consistency. *TPDS*, 16(7): 663–671, July 2005. ISSN 1045-9219.

[19] M. K. Chen and K. Olukotun. The jrpm system for dynamically parallelizing java programs. In *ISCA*, 2003.

[20] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *PLDI*, 2007.

[21] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[22] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, 2009.

[23] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *SPAA*, 1991.

[24] P. B. Gibbons and E. Korach. The complexity of sequential consistency. In *IPDPS*, 1992.

[25] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4), August 1997.

[26] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *CAV*, pages 401–413, 2004.

[27] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.

[28] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *PPoPP*, 2013.

[29] D. Lea. EDU.oswego.cs.dl.util.concurrent.ConcurrentHashMap in the EDU.oswego.cs.dl.util.concurrent package. `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/ConcurrentHashMap.html`, .

[30] D. Lea. util.concurrent.ConcurrentHashMap in java.util.concurrent the Java Concurrency Package. `http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html`, .

[31] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL*, 1997.

[32] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.

[33] 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 *ISCA*, 2010.

[34] F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1989.

[35] A. Meixner and D. J. Sorin. Dynamic verification of sequential consistency. In *ISCA*, 2005.

[36] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1), February 1991.

[37] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, 2002.

[38] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[39] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware support for detecting sequential consistency violations dynamically. In *MICRO*, 2012.

[40] B. Norris and B. Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *OOPSLA*, 2013.

[41] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP*, 2010.

[42] X. Qian, J. Torrellas, B. Sahelices, and D. Qian. Volition: Scalable and precise sequential consistency violation detection. In *ASPLOS*, 2013.

[43] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. Fast and generalized polynomial time memory consistency verification. In *CAV*, 2006.

[44] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *ICS*, 2007.

[45] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 15:391–411, Nov. 1997.

[46] E. Torlak, M. Vaziri, and J. Dolby. Memsat: Checking axiomatic specifications of memory models. In *PLDI*, 2010.

[47] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[48] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, 1995.

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

A. Correctness of SC Analysis Algorithm

This section proves that our algorithm outputs that a trace is SC iff the trace can be reordered to satisfy the $\mathcal{SC}$ predicate. We begin the proof by showing in Lemma A.1 that the inference rules for the $\xrightarrow{isc}$ relation will not generate any cycles when applied to an SC trace. This proof holds even when we include the ordering of operations that the SC trace provides into the $\xrightarrow{isc}$ relation. We then combine this lemma with the observation that our algorithm enumerates all orderings of writes that are not already ordered by the

$\xrightarrow{isc}$ relation in Theorem A.3. This suffices to show that if an input trace can be shuffled to be SC, then our algorithm will generate an acyclic $\xrightarrow{isc}$ relation and thus output that the trace is SC.

Lemma A.1 (Acyclicity of the $\xrightarrow{isc}$ relation for Sequentially Consistent Executions)**.** The $\xrightarrow{isc}$ relation generated for a sequentially consistent trace is acyclic.

Proof. We prove this property by showing that the $\xrightarrow{isc}$ relation is a subset of the order of statements in the SC execution.

As the *sb* order is included in the SC execution order, all edges added by the sequenced-before rule to the $\xrightarrow{isc}$ relation are trivially consistent with the SC order.

The reads from rule only adds edges to the $\xrightarrow{isc}$ relation that are already in the SC order as the $\mathcal{SC}$ constraint $s < i$ implies that loads always appear in SC after the stores that they read from.

The read before write rule only adds edges to the $\xrightarrow{isc}$ relation that are in the SC order as the $\mathcal{SC}$ constraint shows that any operation that appears after W2 and before R must either not be a store or have a different address than W2. Therefore, W1 must appear after R in the SC order.

The write ordering rule only adds edges to the $\xrightarrow{isc}$ relation that are in the SC order as the $\mathcal{SC}$ constraint implies that any operation that appears after W2 and before R must either not be a store or have a different address than W2. Therefore, W1 must appear before W2 in the SC order. □

We also need to show that if our algorithm outputs that a trace is SC, then the trace really can be reordered to satisfy the predicate $\mathcal{SC}$. We begin by showing in Lemma A.2 that if the $\xrightarrow{isc}$ relation orders all writes to the same memory location and the $\xrightarrow{isc}$ relation is acyclic, then the execution can be shuffled to satisfy the predicate $\mathcal{SC}$. Theorem A.3 then observes that our algorithm by construction ensures that the $\xrightarrow{isc}$ relation orders all writes to the same memory location. It then follows directly from the lemma that if the algorithm outputs that a trace is SC, that the trace can be reordered to satisfy the predicate $\mathcal{SC}$.

Lemma A.2 (Correspondence between the $\xrightarrow{isc}$ relation and Sequentially Consistent Executions)**.** If all StoreOps to a given location are totally ordered by the $\xrightarrow{isc}$ relation, and the $\xrightarrow{isc}$ relation is acyclic, then a topological sort of the $\xrightarrow{isc}$ relation produces a trace τ^{SC} that satisfies the predicate $\mathcal{SC}$.

Proof. As the $\xrightarrow{isc}$ relation is assumed to be a DAG, we know that it has at least one topological sort. Take the topologically sort of the $\xrightarrow{isc}$ relation. Consider a store S with the index s in the topological sort and a load L with the index i that reads from S.

By the application of the reads-from rule, we know that all loads must read from a store that appears earlier in the topological sorted order and therefore $s < i$.

We next need to prove that $\forall j. s < j < i, \tau(j) \notin$ *StoreOps* $\vee$ $\texttt{address}(\tau(j)) \neq \texttt{address}(\tau(i))$. We prove this by contradiction. Consider an arbitrary operation O with index j that is larger than s and smaller than i. If the operation is not a store, the predicate is trivially true. Therefore, assume that the operation is a store to the same location as S. Given that we assumed that the $\xrightarrow{isc}$ relation totally orders stores to the same location, we have $S \xrightarrow{isc} O$. By the read before write rule, we also have $R \xrightarrow{isc} O$ and therefore O must appear after R in the topological sort. This contradicts $j < i$. □

Theorem A.3 (Algorithm Correctness)**.** The procedure CHECKSC correctly checks whether a given execution is allowed by SC.

Proof. We have to prove both: (1) that if the CHECKSC states that an execution is in SC then the execution is SC and (2) that if an execution is in SC, that CHECKSC returns true.

The procedure CHECKSC adds edges to the $\xrightarrow{isc}$ relation between all writes to the same location that are not already ordered by the $\xrightarrow{isc}$ relation. Therefore, by Lemma A.2, if CHECKSC discovers that the $\xrightarrow{isc}$ relation is acyclic, then the trace in `seq` must satisfy the predicate $\mathcal{SC}$.

If the execution trace is allowed by SC, then there must exist a total ordering of writes such that the $\xrightarrow{isc}$ relation is acyclic by Lemma A.1. If the procedure CHECKSC orders writes in the same order as in SC, then the $\xrightarrow{isc}$ relation that it computes will be acyclic and therefore it will identify the trace as SC. Consider two writes to the same location. If the $\xrightarrow{isc}$ relation orders them, by Lemma A.1 it must order them in the same order as SC. The procedure CHECKSC will then naturally process them in that order.

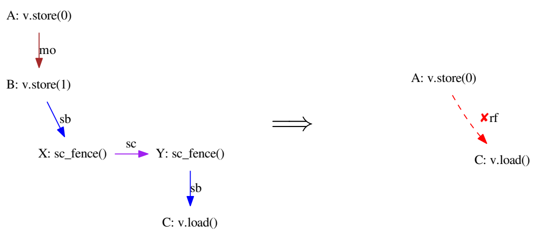
If the writes are not ordered by the $\xrightarrow{isc}$ relation, then the backtracking algorithm will try both orders and hence order them in the same way as SC. Therefore, CHECKSC will return that the execution is allowed by SC. □

B. Memory Order Parameters Inference Rules with Fence

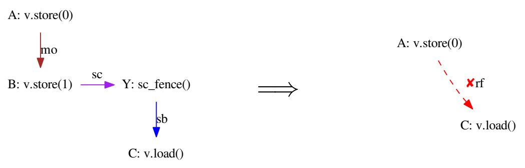
In the body of this paper, we discuss a set of rules that can be used to strengthen order parameters to memory accesses so that the SC violations can be eliminated without considering fences. C and C++ defines an atomic fence operation, which loosely imitates the low-level fence instructions provided by processors for ordering memory accesses and can in some cases allow developers to write code more efficiently.

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]). Each memory order imposes different modification order constraints and synchronization properties. We will discuss the inference rules involving fence operations to infer order parameters in this appendix. When developers use AutoMO to infer parameters for data structures that have fences, we assume that they attach some wildcard order parameters to fence operations, and then AutoMO figures out the proper parameters for each. Figure 21 and Figure 22 together show the rules involving fence operations that can be used to eliminate SC violations.

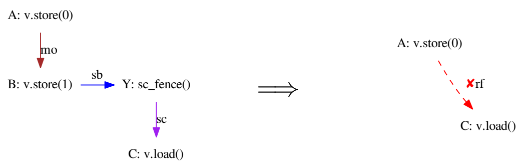
Eliminate Reads-from III (A) (Fence)



Eliminate Reads-from III (B) (Fence)



Eliminate Reads-from III (C) (Fence)



Impose Modification Order (D) (Fence)

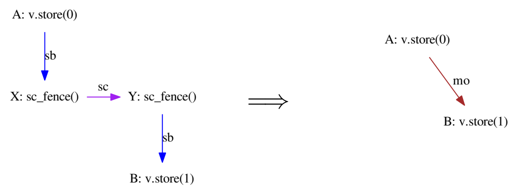


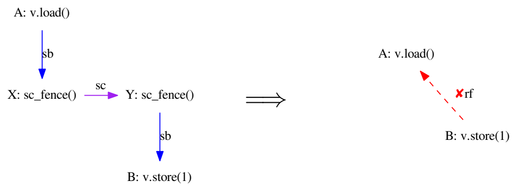
Figure 21. Fence inference rules for non-SC traces (part I)

Eliminate Reads-From III: We derive these three rules (Rule A, B and C) from §29.3p4, 29.3p5 and 29.3p6 [3]. To summarize, these rules require that operation C must read from any operation that is later than operation A in modification order, and thus the *reads-from* edge can be eliminated.

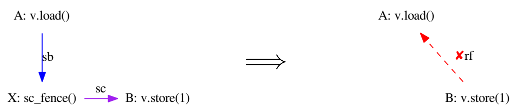
Impose Modification Order (Fence): §29.3p7 [3] requires explicitly that operation A will be later than operation B in modification order.

Eliminate Reads-From IV: According to §29.3p3 [3], *sc* order must be consistent with *happens-before*. In Rule E,

Eliminate Reads-from IV (E) (Fence)



Eliminate Reads-from IV (F) (Fence)



Eliminate Reads-from IV (G) (Fence)

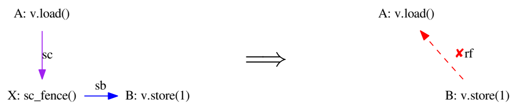


Figure 22. Fence inference rules for non-SC traces (part II)

for example, if operation A reads from operation B, then according to §29.8 fence Y happens before fence X, and it has a conflict with the *sc* edge from X to Y. Therefore, A is not allowed to read from B. Similar rules apply to Rule F and Rule G.

To extend AutoMO to support inferring order parameters for fences with the above discussed rules, we extended the search for possible repairs such that it also searches for repairs involving fence operations when possible. Since a fence operation with `memory_order_relaxed` parameter means a no-op, any inference results that have a relaxed fence imply that a fence operation is not needed at that location.