Testing Concurrent Programs on Relaxed Memory Models

Jacob Burnim Koushik Sen Christos Stergiou
EECS Department, University of California, Berkeley, CA, USA
{jburnim,ksen,chster}@cs.berkeley.edu

ABSTRACT

High-performance concurrent libraries, such as lock-free data structures and custom synchronization primitives, are notoriously difficult to write correctly. Such code is often implemented without locks, instead using plain loads and stores and low-level operations like atomic compare-and-swaps and explicit memory fences. Such code must run correctly despite the relaxed memory model of the underlying compiler, virtual machine, and/or hardware. These memory models may reorder the reads and writes issued by a thread, greatly complicating parallel reasoning.

We propose RELAXER, a combination of predictive dynamic analysis and software testing, to help programmers write correct, highly-concurrent programs. Our technique works in two phases. First, RELAXER examines a sequentially-consistent run of a program under test and dynamically detects potential data races. These races are used to predict possible violations of sequential consistency under alternate executions on a relaxed memory model. In the second phase, RELAXER re-executes the program with a biased random scheduler and with a conservative simulation of a relaxed memory model in order to create with high probability a predicted sequential consistency violation. These executions can be used to test whether or not a program works as expected when the underlying memory model is not sequentially consistent.

We have implemented RELAXER for C and have evaluated it on several synchronization algorithms, concurrent data structures, and parallel applications. RELAXER generates many executions of these benchmarks with violations of sequential consistency, highlighting a number of bugs under relaxed memory models.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Reliability, Verification

Keywords

Active testing, concurrency, relaxed memory models

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '11, July 17–21, 2011, Toronto, ON, Canada
Copyright 2011 ACM 978-1-4503-0562-4/11/05 ...$10.00.

1. INTRODUCTION

In a multithreaded program, a *data race* occurs when two threads simultaneously access the same shared variable and at least one of the accesses is a write. Data races are the most common and notorious kind of concurrency bug. The conventional wisdom is that programmers should use synchronization mechanisms such as locks to ensure that their programs contain no data races, and a large body of research over the past thirty years has focused on detecting data races in parallel programs.

Yet software developers often write programs with intentional data races. Highly-concurrent libraries, such as lock-free data structures and custom synchronization operations, may contain intentional data races for performance. Such code uses compare-and-swap-like primitives or unsynchronized reads and writes to avoid the overhead of locks. Additionally, developers of large applications often introduce data races to avoid the cost of synchronization on certain frequent operations.

The presence of explicit data races, however, greatly increases the difficulties of writing correct parallel software. In addition to the normal challenges of reasoning about thread interleavings and of nondeterminism in testing and debugging, data races expose the programmer to the underlying *relaxed memory model* of the language and hardware. Under such relaxed memory models, a program can exhibit a wide range of counter-intuitive behaviors that can be quite difficult to reason about.

Consider the example program in Figure 1. The program has two data races—the write of `x` at Line 1 races with the read of `x` at Line 4, and the write of `y` at Line 3 races with the read of `y` at Line 2. Under sequential consistency, the final assertion will always hold. Either the statement at Line 2 or at Line 4 is the last to execute, so at least one of them must see an earlier write of 1. But, in the absence of any synchronization, most relaxed memory models permit the writes to `x` and `y` to be *delayed* or *reordered* so that they occur after the reads at Lines 2 and 4. Thus, the final values of `t1` and `t2` can both be 0. That is, these seemingly benign data races can lead to an error under a relaxed memory model.

Thus, there is a need for techniques and tools to aid developers in writing and testing high-performance parallel software with in-

```
Initially: x = y = 0;

thread1:            thread2:
1:  x = 1;          3:  y = 1;
2:  t1 = y;         4:  t2 = x;

 assert(t1 == 1 || t2 == 1);
```

Figure 1: Program with error under relaxed memory model TSO, but correct with sequentially-consistent memory.

tentional data races. While there exist many tools to help programmers find and reproduce data races in their programs, few address the challenges of ensuring program correctness when these races expose violations of sequential consistency. Recently, there have been a number of efforts to verify and model check concurrent programs on relaxed memory models [6, 20, 9, 28, 3, 10, 4]. Some of these techniques [9, 28, 3] encode a program as well as the axiomatic semantics of the underlying memory model as a constraint system and use a constraint solver to find bugs. Other techniques [6, 20, 10] explicitly explore the state space of a program to find bugs. More recently, SOBER [4] and [5] have been proposed for run-time monitoring of sequentially consistent executions to find violations of sequential consistency on "nearby" executions. Although these techniques scale and find bugs, they must be driven by a model checker to encounter executions which come close to violating sequential consistency. Moreover, violations of sequential consistency are not necessarily always bugs in, e.g., highly-concurrent data structures.

In this paper, we propose an *active testing* technique, RELAXER, to detect bugs under relaxed memory models. Active testing [22] is a two-phase analysis and testing approach for predicting and confirming concurrency bugs. Active testing techniques have been developed for detecting real data races [22, 13], atomicity violations [19, 13], and deadlocks [11]. In the first phase of active testing, a static or dynamic analysis is used to predict *potential* concurrency bugs in a test program—e.g. pairs of statements that could be data races. In the second phase, for each potential concurrency bug, the test program is executed with the analysis actively controlling the thread schedule to cause the bug to occur. A concurrency bug is reported to the user only if it is observed in some execution during the second phase—e.g. if a pair of potentially-racing statements actually race in a created execution.

Applying active testing in RELAXER to find relaxed memory model bugs poses several challenges. For the first phase of RELAXER, we must devise an analysis to predict violations of sequential consistency in a test program. We use the existing notion of a *happens-before cycle* to characterize a violation of sequential consistency and formalize a *potential happens-before cycle* as a sequence of potential data races. We propose a simple dynamic analysis that, given a sequentially-consistent execution of a test program, predicts such cycles. In the second phase, RELAXER attempts to create each potential happens-before cycle by running the test program while actively controlling not only the thread scheduler but also the underlying memory model. This enables RELAXER to create feasible potential cycles with high probability and to test if the resulting sequential consistency violations lead to real bugs.

In order to control the underlying memory model, the second phase of RELAXER requires an operational implementation of each relaxed memory model under consideration. In this paper, we consider three memory models, *Total Store Order* (TSO) [24] and *Partial Store Order* (PSO) [24], and *Partial Store Load Order* (PSLO)[16]. TSO allows to delay a store relative to subsequent loads on a processor. The operational model of TSO uses a store buffer for each processor. This memory model is quite similar to that of the x86 architecture. PSO additionally allows to delay stores relative to other stores to different addresses on the same processor. The operational model of PSO is implemented by using a separate buffer for each address per processor. These write-write reorderings are present in many architectures, including Alpha, IA64, and POWER. PSLO captures some of the behaviors of SPARC's *Relaxed Memory Order* (RMO) [24] and other very relaxed memory models like Alpha, IA64, and POWER. PSLO is like PSO, but additionally allows loads to be reordered to before previous loads and stores on the same processor. The operational model of PSLO is implemented by keeping a per-thread store history for each address.

RELAXER has several advantages:

- RELAXER provides conservative implementations of the operational semantics of various relaxed memory models. These operational models are often intuitive to understand and debug. With RELAXER, a programmer can plug-in an implementation of a relaxed memory model and observe the actual run-time behavior of a program under the model. This can be useful for testing and debugging.
- RELAXER can help distinguish harmful violations of sequential consistency from benign ones. RELAXER, after creating a happens-before cycle, completes the test execution to check for any crashes or functional errors. Further, it produces execution traces that can be used to help debug any errors found.
- RELAXER can quickly trigger memory model related bugs even though such bugs often happen only under rare schedules and memory operation reorderings. RELAXER does this by predicting potential happens-before cycles and guiding the execution of the program to create the cycles with high probability.

We have implemented RELAXER for C programs and applied it to two mutual exclusion algorithms and several concurrent data structures and parallel applications. Our implementation intercepts all loads, stores, fences, and other memory operations, providing debuggable simulations of these operations under various relaxed memory models. RELAXER rapidly predicts and then creates many executions for these benchmarks in which sequential consistency is violated. Further, RELAXER creates many executions in which violations of sequential consistency lead to program errors, highlighting bugs in the benchmarks under relaxed memory models.

2. OVERVIEW

In this section, we give an informal overview of RELAXER on a simple example.

Consider the parallel program in Figure 2. Under a sequentially consistent memory model, this program cannot encounter the ERROR at line 6. However, under many relaxed memory models provided by current multiprocessor systems, the ERROR can be encountered. We describe in Section 2.1 exactly how a violation of sequential consistency can lead to this error under the Partial Store Order (PSO) model. And in Section 2.2, we describe how RELAXER characterizes these violations using happens-before cycles and data races.

In Sections 2.3 and 2.4, we describe how RELAXER uncovers an execution of our example program exhibiting this ERROR. As an active testing technique, RELAXER consists of two phases. In the first phase (Section 2.3), RELAXER examines a sequentially consistent execution of the example program and *predicts* possible violations of sequential consistency under PSO. In the second phase (Section 2.4), RELAXER executes the test program under PSO, biasing the program's thread schedule and the underlying memory model to produce executions in which the predicted violations actually occur.

2.1 Error Under Partial Store Order (PSO)

Consider the parallel program in Figure 2. In reasoning about parallel programs, we typically assume *sequential consistency*. That is, we think of the program's execution as simply interleaving the operations of each thread. Under sequential consistency, the ERROR on Line 6 cannot occur, because there is only a single interleaving in which `thread2` reads `done=1` at Line 4. And in

```
Initially: x = y = done = 0;

thread1 {                 thread2 {
                          4:    if (done) {
1:   x = 1;               5:       if (x==0)
2:   y = 1;               6:          ERROR;
3:   done = 1;           7:       local = y;
                          8:    }
```

Figure 2: A program correct under sequential consistency, but with an error under relaxed memory model PSO.

this interleaving, `thread1` executes completely before `thread2`, and thus `x` will not be zero at Line 5.

Our example program has additional behaviors, however, under many relaxed memory models that are provided by current multiprocessor systems. For example, the Partial Store Order (PSO) model allows some stores by a thread to be *delayed*, only becoming visible to other threads at some later time, while the thread continues to execute statements. As we formalize in Section 4.2, we can understand PSO as if every thread maintained a FIFO *store buffer* for each memory address. Then, a possible execution of the program in Figure 2 is the following:

1. `thread1` executes Line 1, but rather than write `x=1` directly to shared memory, the store `x=1` is buffered in `thread1`'s store buffer for `x`.
2. `thread1` executes Lines 2 and 3, writing `y=1` and `done=1` to shared memory.
3. `thread2` executes Line 4, reading `done=1`.
4. `thread2` executes Line 5, reading `x=0`.
5. `thread1` *commits* its buffered store `x=1`. That is, it now writes `x=1` to shared memory so that it is globally visible.
6. `thread2` encounters the ERROR at Line 6.

Note that to fix such an error, it is necessary to introduce some kind of synchronization or memory barrier to prevent the writes of `x=1` and `done=1` from being reordered.

2.2 Happens-Before Cycles and Races

To convince ourselves that the above behavior is a violation of sequential consistency (SC), we note that, under SC, we could reason as follows about causality among the executed statements:

- The store `x=1` at Line 1 must have ***happened before***[1] the store `done=1` at Line 3, because the stores are from the same thread and `x=1` is performed first.
- The store `done=1` at Line 3 must have ***happened before*** the load of `done` at Line 4, because Line 4 reads the value written by Line 3.
- The load of `done` at Line 4 must have ***happened before*** the load of `x` at Line 5, because the loads are from the same thread and Line 4 is performed first.
- The load of `x` at Line 5 must have ***happened before*** the store `x=1` at Line 1, because Line 5 reads the initial value 0 for `x` rather than the value written by Line 1.

But these arguments imply a *happens-before cycle*: Line 1 happens before Line 3, which happens before Line 4, which happens before Line 5, which happens before Line 1. Such a cycle is a contradiction—Line 1 cannot have happened before itself—which shows that this behavior is ***not*** sequentially consistent.

[1]We formally define this *happens-before* relationship in Section 3.

Note that, in this *happens-before cycle*:

$$\text{Line 1} \longrightarrow_p \text{Line 3} \longrightarrow_c \text{Line 4} \longrightarrow_p \text{Line 5} \longrightarrow_c \text{Line 1}$$

the edges alternate between: (1) edges from one statement to another by the same thread (program order, indicated by $\rightarrow_p$), and (2) edges from one statement to another statement accessing the same variable in a different thread (conflict order, indicated by $\rightarrow_c$).

In particular, the $\rightarrow_c$ happens-before edges involve pairs of statements that form *data races*. This violation of sequential consistency is possible because of the two data races between `thread1` and `thread2`—one on `x` between Lines 1 and 5 and one on `done` between Lines 3 and 4. These races can lead to an SC violation because the variables involved are accessed in opposite order by the two threads: `thread1` writes `x` before `done`, but `thread2` reads `x` after `done`.

We show in Section 4 that this pattern of happens-before relationships, alternating between races and same-thread ordering, captures all violations of sequential consistency.

2.3 Phase I of RELAXER

In the first phase, RELAXER examines a sequentially consistent execution of our example program and uses a dynamic analysis to predict *potential* violations of sequential consistency. In the second phase, described in the next section, RELAXER will attempt to create executions actually exhibiting these violations.

Given a sequentially consistent trace of a test program, RELAXER's predictive dynamic analysis identifies program statements that could *potentially* form a happens-before cycle in a run under a relaxed memory model. As discussed in the previous section, such happens-before cycles are violations of sequential consistency.

Suppose that RELAXER observes the sequentially consistent execution where `thread1` runs completely and then `thread2` runs. RELAXER first performs a standard dynamic race detection analysis, finding three data races—pairs of accesses to the same memory location from different threads, with at least one access a write and without any synchronization ordering the accesses:

`1: x = 1` and `5: read x`
`2: y = 1` and `7: read y`
`3: done = 1` and `4: read done`

Although the observed execution cannot have any actual happens-before cycles, because it is sequentially consistent, RELAXER reports two *potential happens-before cycles*:

$$\text{Line 1} \longrightarrow_p \text{Line 3} \longrightarrow_r \text{Line 4} \longrightarrow_p \text{Line 5} \longrightarrow_r \text{Line 1}$$
$$\text{Line 2} \longrightarrow_p \text{Line 3} \longrightarrow_r \text{Line 4} \longrightarrow_p \text{Line 7} \longrightarrow_r \text{Line 2}$$

These *potential happens-before cycles* are like happens-before cycles, in that the $\rightarrow_p$ edges indicate happens-before between pairs of executed statements from the same thread. But the $\rightarrow_r$ are only *potential* happens-before—e.g., "Line 5 $\longrightarrow_r$ Line 1" indicates that Lines 1 and 5 are a data race and that, in some execution under a relaxed memory model such as PSO, it could be the case that Line 5 happens-before Line 1. (But in the observed SC execution, Line 5 does *not* happen-before Line 1.)

2.4 Phase II of RELAXER

For each of the two potential happens-before cycles from Phase I, RELAXER will attempt to create executions under the relaxed memory model PSO in which these cycles are actual happens-before cycles and thus violate sequential consistency.

As described in Section 2.1, under PSO threads are free to locally "buffer" writes to shared memory, deferring the actual write until

later. RELAXER uses this freedom, in combination with control over the thread scheduler, to create happens-before cycles.

For the first potential happens-before cycle, on Lines 1, 3, 4, and 5, RELAXER will begin executing our example program. But RELAXER intercepts every shared read and write of the program in order to control the program's thread schedule and to simulate the possible PSO behaviors. RELAXER's goal is to ensure that both Line 3 happens-before Line 4 and that Line 5 happens-before Line 1, creating a happens-before cycle and violating sequential consistency. Suppose `thread2` is first to execute a statement:

1. Just before `thread2` reads `done` at Line 4, RELAXER will pause `thread2`, because Line 3 is supposed to happen-before Line 4.
2. Then, when `thread1` writes `x=1` at Line 1, RELAXER intercepts this write and buffers it in `thread1`'s store buffer for `x`, because the read at Line 5 is supposed to happen-before this store.
3. RELAXER then lets `thread1` execute Lines 2 and 3 normally, and then allows `thread2` to resume.
4. `thread2` then executes Line 4, which see the write `done=1` from Line 3, and then executes Line 4, which does *not* see the write `x=1` from Line 1 because that write is still buffered. Line 4 thus reads 0 for `x`.
5. `thread1` then commits its buffered store `x=1`.

RELAXER has now created the predicted happens-before cycle:

$$\text{Line 1} \longrightarrow_p \text{Line 3} \longrightarrow_c \text{Line 4} \longrightarrow_p \text{Line 5} \longrightarrow_c \text{Line 1}$$

RELAXER then allows the program to continue its execution normally in order to observe whether any errors occur. The execution soon encounters the ERROR at Line 6. RELAXER outputs this trace as an execution possible under memory model PSO which not only violates sequential consistency (SC), but leads to a program error.

RELAXER similarly creates an SC violation involving the second potential happens-before cycle by delaying the write to `y`. This execution, however, does not lead to an error. If the test program is fully equipped with assertions to detect incorrect program behavior, then the programmer need not examine the execution traces when all assertions pass. RELAXER can thus test that programs behave as expected under various relaxed memory models.

3. FORMAL BACKGROUND

Before formally describing the RELAXER algorithm, we formally define (along the lines of [4]) execution traces of parallel programs, the happens-before relation discussed in the previous section, and sequential consistency.

3.1 Execution Traces

The axiomatic semantics of various memory models are described in terms of an execution *trace* of a program, which is a sequence of events. The events of a *trace* are:

- *ld*(l, p, a, v, i, c), denoting a load of the value v from memory address a by processor p. The load is the i^{th} instruction executed by processor p according to the program order and it returns the c^{th} value written to the address a. i is called the instruction index and c is called the commit index. l is the label of the instruction (same for below),
- *st*(l, p, a, v, i, c), denoting a store of the value v at the memory address a by processor p. The store is the i^{th} instruction executed by processor p according to the program order and it is the c^{th} value written to the address a.

We use corresponding projection functions $l(e)$, $p(e)$, $a(e)$, $v(e)$, $i(e)$, $c(e)$ for an event e, and use $o(e)$ to denote the type of e—i.e. *ld* or *st*. A trace is generated from a concurrent program execution by logging all loads and stores.

3.2 Sequential Consistency

In order to define a sequentially consistent execution, we define two relations on the set of events.

Events e and e' are related by the *program-order relation*, denoted by $e \rightarrow_p e'$, iff $t(e) = t(e')$ and $i(e) \leq i(e')$. That is, $e \rightarrow_p e'$ when either $e = e'$ or e and e' are events from the same thread and that thread issues e before e'

Two events e, e' are related by the *conflict-order relation*, denoted by $e \rightarrow_c e'$, iff $a(e) = a(e')$ and either (1) $o(e') = st$ and $c(e) < c(e')$, or (2) $o(e) = st$ and $o(e') = ld$ and $c(e) \leq c(e')$. Informally, $e \rightarrow_c e'$ means that e and e' are operations on the same memory location and e happened first from the point of view of that memory location.

The classic *happens-before relation* is defined as follows:

$$\rightarrow_{hb} = (\rightarrow_p \cup \rightarrow_c)$$

Then, we define a trace to be *sequentially consistent* in terms of the happens-before relation:

DEFINITION 1. *A trace T is* sequentially consistent *if $\rightarrow_{hb}$ is acyclic on the events in T.*

Note that this condition is equivalent to requiring the existence of a total ordering on the events in T that is consistent with the happens-before relation.

Given Definition 1 above, we can say that a trace generated under a relaxed memory model violates sequential consistency iff the trace contains a cycle under the classic happens-before relation $\rightarrow_{hb}$. Thus, such a *happens-before cycle* is a witness to the violation of sequential consistency.

4. ALGORITHM

In this section, we provide a full description of the RELAXER algorithm. We assume that we are given a concurrent test harness, a closed program with fixed inputs and assertions checking its output.

RELAXER works in two phases. In the first phase, described in Section 4.1, RELAXER runs the test harness on random schedules on the sequentially consistent memory model and observes the various memory load and store events. RELAXER analyzes these events to find sets of events such that the events in each set could form a cycle in the happens-before relation in some other potentially feasible schedule.

In the second phase, for each such set of events, RELAXER executes the program again and actively controls the schedule and memory operations so that the cycle in the happens-before relation gets created with reasonable probability. RELAXER also checks if the assertions in the harness could be violated in the controlled execution; if a violations happens, then RELAXER reports the error along with a concrete execution trace for the purpose of debugging.

The second phase of RELAXER is parameterized by the memory model on which we are testing. In particular, RELAXER requires operational semantics for a memory model so that it can simulate the model during testing. RELAXER's strategy for controlling the schedule and the behavior of memory operations is then specific to each memory model.

Thus, we describe in Section 4.2 our operation models for three relaxed memory models: TSO, PSO, and PSLO. Then in Section 4.3 we describe Phase II of RELAXER on each model.

Algorithm 1 Phase I of RELAXER

INPUTS: a trace
$i \Leftarrow 1$
$C^i \Leftarrow \{e, e' \mid e \rightarrow_p e'\}$
while $C^i \neq \emptyset$ **do**
  for each e in the trace and each c in C^i **do**
    if c, e is a chain
      if i is even and c, e is a potential $\rightarrow_{hb}$-cycle
        report $l(c, e)$
      else
        add c, e to C^{i+1}
  $i \Leftarrow i + 1$

4.1 Phase I of RELAXER

In Phase I, RELAXER predicts set of events which could be involved in a potential cycle of the happens-before relation.

For prediction, RELAXER generates a trace of the concurrent program under test by running it on a random thread schedule and on a sequentially consistent memory. Note that we do not address here the problem of executing the test program on a random schedule. There exists much prior work on generating thread schedules for testing concurrent software, including noise making [7, 25], active random scheduling [22, 13], or model checking [27, 17].

We focus only on the problem of predicting potential cycles given an execution trace. For such a trace, RELAXER looks for a sequence of events, not necessarily adjacent in the trace, that form a potential $\rightarrow_{hb}$-cycle:

DEFINITION 2. *A sequence of events $e_1, e'_1, e_2, e'_2, \ldots e_n, e'_n$ is called a $\rightarrow_{hb}$-*chain *iff*

- $\forall j \in [1, n]$, $e_j \rightarrow_p e'_j$,
- $\forall j, k \in [1, n]$, $p(e_j) \neq p(e_k)$,
- $\forall j \in [1, n-1]$, e'_j *and* e_{j+1} *are in potential data race, i.e.* $p(e'_j) \neq p(e_{j+1})$, $a(e'_j) = a(e_{j+1})$, *and* $o(e'_j) = st \vee o(e_{j+1}) = st$

DEFINITION 3. *A sequence of events $e_1, e'_1, e_2, e'_2, \ldots e_n, e'_n$ is called a potential $\rightarrow_{hb}$-*cycle *iff it is a $\rightarrow_{hb}$-chain and e'_n and e_1 are in potential data race.*

Note that in a potential $\rightarrow_{hb}$-cycle, for any $j \in [1, n-1]$, either $e'_j \rightarrow_c e_{j+1}$ or $e_{j+1} \rightarrow_c e'_j$ in the trace. Similarly, either $e'_n \rightarrow_c e_1$ or $e_1 \rightarrow_c e'_n$. Note that the potential $\rightarrow_{hb}$-cycle cannot be a real cycle on the happens-before relation of the current trace as the trace is being generated by running the program on a sequentially consistent memory model. However, since e'_n and e_1 are in potential data race and $\forall j \in [1, n-1]$, e'_j and e_{j+1} are in potential data race, we could expect that one can execute the program in a controlled way on a relaxed memory model where $e'_n \rightarrow_c e_1$ and $\forall j \in [1, n-1]$, $e'_j \rightarrow_c e_{j+1}$ and thereby creating a real $\rightarrow_{hb}$-cycle. In Phase II, RELAXER tries to create such cycles by executing the program in a controlled way on a relaxed memory model.

```
m : array[Adr] of 𝕍

l:store_SC(p,a,v):
    m[a] = v

l:load_SC(p,a):
    return m[a];

l:CAS_SC(p,a,oldv,newv):
    if (m[a] == oldv)
      m[a] = newv
```

Figure 3: Operational Model of SC

For each sequence of events $c = e_1, e'_1, e_2, e'_2, \ldots e_n, e'_n$ that forms a potential $\rightarrow_{hb}$-cycle, RELAXER records the sequence $l(e_1), l(e'_1), l(e_2), l(e'_2), \ldots l(e_n), l(e'_n)$. Let us denote this sequence of events by $l(c)$. At the end of Phase I, RELAXER reports a set of sequences of instruction labels. RELAXER uses the reports from Phase I in Phase II to try to create those cycles and to check if there could be an error due to the created cycle.

Given a trace, RELAXER computes all potential $\rightarrow_{hb}$-cycles as follows: Let C^n be the set of all $\rightarrow_{hb}$-chains of length n. RELAXER computes potential $\rightarrow_{hb}$-cycles by iteratively computing $C^1, C^2, C^3, C^4 \ldots$ and finding potential $\rightarrow_{hb}$-cycles in those sets. This iterative algorithm is listed in Algorithm 1.

Note that in the algorithm, we do not add a $\rightarrow_{hb}$-chain to C^{i+1} if it is a potential hb-cycle. This ensures that we do not report $\rightarrow_{hb}$-cycles that can be decomposed into simpler cycles. Moreover, in Algorithm 1, a potential $\rightarrow_{hb}$-cycle of length $2k$ gets reported k times. For example, if $e_1, e'_1, e_2, e'_2, \ldots, e_n, e'_n$ is reported as a potential $\rightarrow_{hb}$-cycle, then $e_2, e'_2, \ldots, e_n, e'_n, e_1, e'_1$ is also reported as a cycle. We remove all such duplicates.

4.2 Operational Models

In this section, we describe our operational models for the three relaxed memory models, TSO, PSO, and PSLO, to which RELAXER has been applied. Note that our operational implementations are a conservative approximation of the full relaxed memory models. Every behavior our operational semantics generate is allowed by the full memory model, so we report only true bugs and violations of sequential consistency.

In these models, a program consists of a set of parallel processes that issue memory operations such as loads, stores, atomic compare-and-swaps, and memory barriers. A memory model implements these operations, deciding what values are returned to a process whenever it performs a load.

Sequential Consistency (SC).

As a point of comparison, we present in Figure 3 a simple operational model for sequential consistency (SC). In SC, memory is modeled as an array `m` mapping addresses `a` ∈ **Adr** to

```
m  : array[Adr] of 𝕍
B  : array[Proc] of FIFOQueue of Adr×𝕍

l:store_TSO(p,a,v):
    B[p].addLast((a,v))

store^c_TSO(p):
  (a,v) = B[p].removeFirst()
  m[a] = v

l:load_TSO(p,a):
    if (B[p].contains((a,*))
      (a,v) = last element (a,*) of B[p]
      return v
    else
      return m[a]

l:CAS_TSO(p,a,oldv,newv):
    membar^sl_TSO(p)
    if (m[a] == oldv)
      m[a] = newv

l:membar^sl_TSO(p):
    while (not B[p].empty())
      store^c_TSO(p)
```

Figure 4: Operational Model of TSO

data values v ∈ $\mathbb{V}$. $\texttt{store}_{SC}$(p,a,v) stores value v at m[a], $\texttt{load}_{SC}$(p,a) returns the value stored at m[a], and $\texttt{CAS}_{SC}$(p,a,oldv,newv) is an atomic compare-and-swap on m[a]. Each function in our operational model executes atomically.

Total Store Order (TSO).

The TSO memory model [24] allows stores to be reordered past later loads, but maintains a total order over stores. To maintain intra-thread consistency, when a store is reordered past a load of the same address, the load still sees the stored value.

Our operational model for TSO is given in Figure 4. We add a FIFOQueue B[p] as a store buffer for each process. Instead of directly updating memory, $\texttt{store}_{TSO}$(p,a,v) enqueues the pair (a,v) to the local buffer B[p]. $\texttt{store}^{c}_{TSO}$(p) is an internal transition that commits to global memory the oldest buffered store for process p. The memory model can non-deterministically call $\texttt{store}^{c}_{TSO}$(p) any number of times at any state of the execution. $\texttt{load}_{TSO}$(p,a) reads from main memory unless there is a pending store to a in B[p], in which case it returns the value of the most recent pending store.

We implement store-load memory barrier $\texttt{membar}^{sl}_{TSO}$(p) by committing all pending stores. Following the SPARC v9 description [24] of TSO, we do not allow compare-and-swaps to be reordered under TSO. We enforce this by including a store-load memory fence in $\texttt{CAS}_{TSO}$(p,a).

Partial Store Order (PSO).

The PSO memory model [24] is similar to TSO, but further allows stores to reordered past later stores to different addresses.

Our operational model for PSO is given in Figure 5. As in SC and TSO, PSO maintains a global memory m. PSO also maintains a store buffer B_a[p][a] for each process p and address a. $\texttt{store}_{PSO}$(p,a,v) enqueues v to the buffer B_a[p][a]. $\texttt{store}^{c}_{PSO}$(p,a) is an internal transition that commits to global memory the oldest buffered store of process p to address a. The memory system can non-deterministically call $\texttt{store}^{c}_{PSO}$(p,a) for any address a any number of times in any state of the execution. $\texttt{load}_{PSO}$(p,a) returns the value from the oldest pending store in B_a[p][a] or from main memory if no stores are buffered.

We conservatively implement both store-store and store-load memory barriers in PSO by committing all pending stores. This is too strict—these fences require only that pending stores commit before later stores commit or later loads are issued. Thus, our operational model does not quite capture all possible PSO behaviors.

Following the SPARC v9 description [24] of PSO, stores can be delayed past compare-and-swaps on different addresses. But pending stores to the same address are committed before the CAS in $\texttt{CAS}_{PSO}$(p,a). Our operational model can be easily changed for instances of PSO in which CASs have stronger ordering properties.

```
m   : array[Adr] of 𝕍
Bₐ  : array[Proc][Adr] of FIFOQueue of 𝕍

l:store_PSO(p,a,v):
    Bₐ[p][a].addLast(v)

store^c_PSO(p,a):
  if (not Bₐ[p][a].empty())
    m[a] = Bₐ[p][a].removeFirst()

l:load_PSO(p,a):
    if (not Bₐ[p][a].empty())
      return Bₐ[p][a].getLast()
    else
      return m[a]

l:CAS_PSO(p,a,oldv,newv):
    while (not Bₐ[p][a].empty())
      store^c_PSO(p,a)
    if (m[a] == oldv)
      m[a] = newv

l:membar^ss_PSO(p) or l:membar^sl_PSO(p):
    foreach a ∈ Adr
      while (not Bₐ[p][a].empty())
        store^c_PSO(p,a)
```

Figure 5: Operational Model of PSO

Partial Store Load Order (PSLO).

The PSLO memory model was proposed by [16] to capture some of the behaviors of RMO [24]. PSLO allows loads to be reordered to before stores and loads of different addresses. This models both speculative issuing of conditional loads (as in RMO) and speculative data-dependent loads (not allowed in RMO).

Our operational model for PSLO is given Figure 6. PSLO is very similar to PSO except that it also maintains a per-process, per-address set S, called *store history*, of past stores. Like in PSO,

```
m   : array[Adr] of 𝕍
Bₐ  : array[Proc][Adr] of FIFOQueue of 𝕍
S   : Set[Proc][Adr] of 𝕍

l:store_PSLO(p,a,v):
    Bₐ[p][a].addLast(v)

store^c_PSLO(p,a):
  if (not Bₐ[p][a].empty())
    m[a] = Bₐ[p][a].removeFirst()
    S[p][a].clear()
    foreach p' ∈ Proc
      S[p'][a].add(m[a])

l:load_PSLO(p,a):
    if (not Bₐ[p][a].empty())
      return Bₐ[p][a].getLast()
    else
      v = a value from S[p][a]
      remove from S[p][a] values earlier
          than v
      return v

l:CAS_PSLO(p,a,oldv,newv):
    while (not Bₐ[p][a].empty())
      store^c_PSLO(p,a)
    if (m[a] == oldv)
      m[a] = newv
    S[p][a].clear()
    foreach p' ∈ Proc
      S[p'][a].add(m[a])

l:membar^ss_PSLO(p):
    foreach a ∈ Adr
      while (not Bₐ[p][a].empty())
        store^c_PSLO(p,a)

l:membar^ll_PSLO(p):
    foreach a ∈ Adr:
      S[p][a] = singleton(m[a])

l:membar^sl_PSLO(p):
    membar^ll_PSLO(p)
    membar^ss_PSLO(p)
```

Figure 6: Operational Model of PSLO

```
B^t : array[Proc] of FIFOQueue of ℕ

l:before_store_TSO(p,a,v):
    if (l ∈ c_o and rand()<P)
      B^t[p].addLast(time() + delay)
    else
      B^t[p].addLast(time())

l:before_membar^sl_TSO(p,a):
    while (not B^t[p].empty())
      waitUntil(B^t[p].getFirst())
      B^t[p].removeFirst()

l:after_any_TSO(p):
    while (not B^t[p].empty()
        and B^t[p].getFirst() < time())
      B^t[p].removeFirst()
      store^c_TSO(p)
```

Figure 7: Phase II for TSO

$\texttt{store}_{PSLO}\texttt{(p,a,v)}$ enqueues a write and $\texttt{load}_{PSLO}\texttt{(p,a)}$ returns the most recent pending store if one exists in the local write buffer. But when there is no pending store, $\texttt{load}_{PSLO}\texttt{(p,a)}$ returns a value chosen non-deterministically from the set `S[p][a]` of past stores and removes all older stored values. (Intuitively, this models reordering the load back to just after that store committed.)

As in PSO, $\texttt{store}^{c}_{PSLO}\texttt{(p,a)}$ commits to the global memory the oldest pending store of process `p` to address `a`. In addition, it clears the set of past stores `S[p][a]`—because no later read of `a` by `p` can be reordered before this store. Then for each process `p'`, it adds the newly-committed value `m[a]` to the set `S[p'][a]`.

We conservatively implement store-store and load-load fences by committing all of `p`'s pending stores in $\texttt{membar}^{ss}_{PSLO}\texttt{(p)}$ and resetting all of `p`'s store histories in $\texttt{membar}^{ll}_{PSLO}\texttt{(p)}$. We conservatively implement a store-load fence by performing both of these actions. As in PSO, a compare-and-swap first commits pending stores to the same address and cannot be buffered.

4.3 Phase II of RELAXER

Given operational models for TSO, PSO, and PSLO, we can now describe the RELAXER Phase II algorithm on these models.

In Phase II, RELAXER is given a potential $\rightarrow_{hb}$-cycle predicted by Phase I. This cycle $c = l_1, l_2, \ldots, l_n$ is a list of an even number n of statement labels l_i. Our goal in Phase II is to execute the test program in a controlled way so that its trace contains events $e_1, \ldots, e_n$ such that, for $1 \leq j \leq n$: (1) $l(e_j) = l_j$, (2) if j is odd, $e_j \rightarrow_p e_{j+1}$, and (3) if j is even, $e_j \rightarrow_c e_{(j+1)\%n}$.

If these conditions are satisfied, then the $e_1, \ldots, e_n$ form a happens-before cycle on the statements $l_1, \ldots, l_n$.

RELAXER focuses on the requirement (3) above, biasing the thread scheduling and controlling the store buffers and histories to try to create the conflict ordering $e_j \rightarrow_c e_{(j+1)\%n}$ for each even j.

Informally, we aim to cause $e_j \rightarrow_c e_{(j+1)\%n}$ for even j by:

(ODD) When RELAXER encounters an instruction labeled $l_{(j+1)\%n}$, such that $(j+1)\%n$ is odd, it aims to *delay* the instruction so that some other thread can first execute a racing statement l_j. This essentially means buffering $l_{(j+1)\%n}$ if it is a store or pausing the thread for some time before executing $l_{(j+1)\%n}$ if it is a load.

(EVEN) When RELAXER encounters an instruction labeled l_j, such that j is even, it aims to let l_j execute quickly, so that it completes before some other thread executes the racing statement $l_{(j+1)\%n} \in c_o$. In TSO or PSO, when l_j is a write this means immediately committing it (unless some other write is pending). In PSLO, when l_j is a load this means reading the earliest value from the store history.

```
B^t_a : array[Proc][Adr] of FIFOQueue of ℕ

l:before_store_PSO(p,a,v):
    if (l ∈ c_o and rand()<P)
      B^t_a[p][a].addLast(time() + delay_S)
    else
      B^t_a[p][a].addLast(time())

l:before_load_PSO(p,a):
    if (l ∈ c_o and rand()<P)
      waitUntil(time() + delay_L)

l:before_CAS_PSO(p,a,oldv,newv):
    if (l ∈ c_o and rand()<P)
      waitUntil(time() + delay_L)

l:before_membar^ss_PSO(p):
l:before_membar^sl_PSO(p):
   while(∃a such that not B^t_a[p][a].empty())
     waitUntil(B^t_a[p][a].getFirst())
     B^t_a[p][a].removeFirst()

l:after_any_PSO(p):
    foreach a ∈ Adr
      while (not B^t_a[p][a].empty()
          and B^t_a[p][a].getFirst() < time())
        B^t_a[p][a].removeFirst()
        store^c_PSO(p,a)
```

Figure 8: Phase II for PSO

We now formally describe RELAXER's Phase II strategy for each memory model. Let c_o and c_e denote the sets of odd-numbered $\{l_i \mid i \in [1,n] \wedge i \text{ odd}\}$ and even-numbered $\{l_i \mid i \in [1,n] \wedge i \text{ even}\}$ labels from the predicted cycle.

Our Phase II algorithms are presented in Figures 7, 8, and 9. The algorithms are described as modifications to the operational memory models. Functions such as `before_load` and `before_store` are executed before each call by a test program to the corresponding memory model function (i.e. **`load`** or **`store`**). Similarly, `after_any` is called immediately after any call to a memory model operation. These functions execute atomically with the underlying memory model function, except where they explicitly call `waitUntil`. Further, the Phase II algorithm for PSLO completely replaces $\textbf{load}_{PSLO}$.

Note that the Phase II algorithms depend on both a notion of *time*—via the `time` function—and a way to pause a running thread until some later time—via the `waitUntil` function. The algorithms work with wall time or a logical time incremented with each executed instruction—we use wall time for simplicity.

Each Phase II algorithm maintains an extra buffer $\texttt{B}^t$ or $\texttt{B}^t_a$, similar to the buffers the `B` and $\texttt{B}_a$, respectively. Whenever a store is buffered, a corresponding commit time is computed and queued in $\texttt{B}^t$ or $\texttt{B}^t_a$ to record when the store should be committed. The `after_any(p)` functions use these buffers to commit any oldest pending stores whose times have been reached. Thus `after_any` ensures that we immediately commit any pending store which is not explicitly delayed or is behind an earlier commit that was delayed.

Note that each decision to delay a store, pause a thread, or load an old value is gated by a condition: `(rand() < P)`. That is, we only perform the delay, etc., with probability `P`, which we set to $\texttt{P} = 0.5$ in our experiments. This helps RELAXER explore all possible dynamic occurrences of static predicted cycles.

```
B_a^t : array[Proc][Adr] of FIFOQueue of ℕ

l: before_store_PSLO(p,a,v):
     before_store_PSO(p,a,v)

l: before_load_PSLO(p,a):
     before_load_PSO(p,a):

l: before_CAS_PSLO(p,a,oldv,newv):
     before_CAS_PSO(p,a)

l: before_membar^ss_PSLO(p):
l: before_membar^sl_PSLO(p):
    foreach a ∈ Adr
      B_a^t[p][a].clear()

l: after_any_PSLO(p):
     after_any_PSO(p)

l:load_PSLO(p,a):
    if(not B_a[p][a].empty())
      return B_a[p][a].getLast()
    else if(|S[p][a]|>1 and l ∈ c_e
            and rand()<P)
      return earliest value from S[p][a]
    else
      S[p][a] = singleton(m[a])
      return m[a]
```

Figure 9: Phase II for PSLO

Total Store Order.

For TSO, the statements in c_o will always be stores, so there is no need to pause before loads. In keeping with principle (ODD), we buffer stores in c_o for some small time `delay`. Further, we pause before store-load barriers until it is time to commit all pending stores, so that no purposely-delayed store is committed early.

Partial Store Order.

For PSO, a load or CAS could also be in c_o. So, in keeping with (ODD), we pause for $\texttt{delay}_L$ before such statements. As in TSO, we pause before forcibly committing any writes at a barrier. We select $\texttt{delay}_L < \texttt{delay}_S$—i.e. delay writes longer than pausing before stores—to ensure that a write in c_o from one thread does not commit while we are paused before a load in c_o in another thread.

Partial Store Load Order.

We treat PSLO the same as PSO except for two differences. First, when we encounter a load in c_e, as discussed in (EVEN), we read the earliest allowed value. Second, there is no need to pause at memory barriers—even if a store $l_{(j+1)\%n}$ has committed, a later load l_j can still read an earlier value from its thread's store history.

5. EVALUATION

In evaluating RELAXER, we want to validate two hypotheses: (1) RELAXER can predict and create real sequential consistency violations. Further, for a predicted and feasible happens-before cycle, RELAXER creates the cycle with high probability. (2) RELAXER can create violations of sequential consistency leading to real bugs.

5.1 Implementation

In order to evaluate our claims, we implemented RELAXER for C. RELAXER intercepts all of the loads and stores executed by a test program via source instrumentation with CIL [18]. We manually replace calls to synchronization primitives such as pthread locks, GCC atomic built-ins, etc., with calls to RELAXER stubs that translate the operations into loads, stores, CASs, and memory barriers. (Following [24] and [3], we treat lock acquires as load-load barriers and lock releases as store-store barriers.) Together, this gives each memory operation a unique static label.

RELAXER then provides implementations for each memory operation for our four memory models: SC, TSO, PSO, and PSLO. In addition to simulating the desired memory model, these implementations generate a trace of an executing test program and perform the controlled scheduling of the Phase II algorithms. The cycle prediction of Phase I is run off-line on generated execution traces.

Column 2 of Table 1 reports the average running time in seconds of Phase I of RELAXER for each benchmark—i.e., the time to execute once the benchmark's test script (defined below), predicting cycles via Algorithm 1. Columns 3 through 5 report average running times in seconds for Phase II for TSO, PSO, and PSLO—i.e., for one predicted cycle, the cost of one execution of the test script, guiding the thread schedule and memory model. All experiments were run on an 8-core Intel Xeon 2.0 Ghz system with 8 GB of RAM and running Ubuntu 10.04.

5.2 Benchmarks

We evaluate RELAXER on ten benchmarks, listed in Table 1. Benchmarks `dekker` and `bakery` are Dekker's algorithm and Lamport's bakery algorithm [14], both classic solutions to the mutual exclusion problem. The next five benchmarks are concurrent data structures—we use the implementations from [3]. The benchmarks are `msn`, a non-blocking queue, `ms2`, a two-lock queue, `lazylist`, a list-based concurrent set, `harris`, a non-blocking set, and `snark`, a non-blocking double-ended queue. We also evaluated RELAXER on three parallel application benchmarks: `pfscan`, a parallel text scanning application, `aget`, a parallel FTP client, and `ctrace`, a tracing library for multithreaded programs.

Neither the mutual exclusion or the data structure benchmarks are closed programs. To apply RELAXER to them, we need test harnesses to exercise the benchmark code. Along the lines of [3], we manually constructed a handful of test harnesses for each benchmark. For `dekker` and `bakery`, our test harnesses contain two threads that use the provided mutex to protect a critical section. Each thread executes its critical section three times and assertions check that the two threads are never concurrently in their critical sections. For the concurrent data structures, each test harness runs multiple data structure operations in parallel and asserts that the results are correct. We check the result by comparing against a set of all possible results when the test harness is run with each data structure operation run atomically. These harnesses are a proxy for the kinds of parallel unit tests we expect a developer would use with RELAXER. For the application benchmarks, a test harness runs the application with a fixed input. The test script for each benchmark runs its harness 100 times (mutual exclusion and data struture benchmarks) or 20 times (application benchmarks).

We report experimental results only for the largest test harnesses we tried for each data structure. The results are similar for the smaller harnesses, except that they have fewer predicted, confirmed, and buggy SC violations. These harnesses are:

- For queues `msn` and `ms2`, one threads performs two enqueues, two dequeues, then an enqueue while a second thread performs two dequeues, two enqueues, then a dequeue.
- For sets (sorted lists) `lazylist` and `harris`, insert 3 and then 7, and then in three parallel threads: (1) insert -1 then remove 2, (2) remove 3, insert 1, then check membership of 7, and (3) check membership of 3 and then insert -5.
- For double-ended queue `snark`, two parallel threads both performing: push-right, push-left, pop-right, then pop-left.

Table 1: Results of RELAXER experimental evaluation.

Benchmark	Approx. LoC	Average Runtime (seconds) phase 1	phase 2			Cycles Predicted	Cycles Confirmed			# of Bugs			Empirical probability of confirming a cycle		
			TSO	PSO	PSLO		TSO	PSO	PSLO	TSO	PSO	PSLO	TSO	PSO	PSLO
dekker	20	1.1	1.9	2.0	2.5	112	47	45	69	39	38	65	0.69	0.81	0.84
bakery	30	2.3	4.4	3.6	5.4	222	36	75	100	33	68	96	0.85	0.84	0.82
msn	80	2.0	1.2	1.4	1.7	459	0	117	144	0	117	144	-	0.84	0.72
ms2	60	1.0	1.8	2.4	1.8	75	0	2	5	0	2	5	-	1.00	0.57
lazylist	120	1.9	2.7	2.9	3.1	192	0	8	10	0	8	9	-	0.96	0.62
harris	160	4.4	2.8	3.4	3.8	172	0	54	49	0	48	49	-	0.35	0.68
snark	150	6.3	1.6	1.0	1.2	1800	0	647	404	0	419	191	-	0.60	0.59
pfscan	1000	9.1	-	-	-	0	-	-	-	-	-	-	-	-	-
aget	1200	26.1	2.6	2.7	3.2	27	0	2	2	0	2	2	-	0.83	0.80
ctrace	1400	10.4	-	-	-	0	-	-	-	-	-	-	-	-	-

5.3 Experimental Setup

We run each of the test scripts described in the previous section once under sequential consistency (SC) and record the execution traces. These schedules are generated via noise making [7, 25]—each SC memory operation contains a small, random sleep. These traces are fed to Phase I of RELAXER and we record all predicted happens-before cycles of length 4. We focus on length-4 cycles because they capture sequential consistency violations involving two threads and ordering of memory operations on two addresses.

For Phase II of RELAXER, for each predicted cycle, we average results over 30 trials of each benchmark's test script under each of TSO, PSO, and PSLO. We record which cycles we are able to create in a real execution, and which of these confirmed sequential consistency violations lead to program errors—i.e. incorrect output, assertion failures, or memory errors.

5.4 Experimental Results

Table 1 summarizes the results of these experiments. For each benchmark listed in Column 1, the number of unique cycles predicted by Phase I is reported in Column 6. Columns 7, 8, and 9 report the number of these predicted cycles that RELAXER was able to create under each of the three memory models. That is, a predicted cycle is "confirmed" if, in at least one of the 30 trials, RELAXER created a real violation of sequential consistency among the static statements in the cycle.

The number of predicted and confirmed cycles validates the first part of our Hypothesis (1): RELAXER can predict and then create many real violations of sequential consistency in our benchmarks. Note that RELAXER correctly reports zero potential cycles for benchmarks `pfscan` and `ctrace`—these benchmarks use sufficient synchronization to maintain sequential consistency.

In the last three columns of Table 1, we estimate the average empirical probability with which one run of Phase II of RELAXER creates a *feasible* cycle. That is, we average over all feasible cycles c the quantity: (# of trials in which c is created)/(# of trials) . As we cannot know which cycles are truly feasible, we assume a cycle c is infeasible under a memory model if no RELAXER trial on that model ever creates c.

These probability estimates provide evidence for the second part of Hypothesis (1). RELAXER can reliably create most of these feasible cycles by running Phase II only a handful of times.

Columns 10, 11, and 12 of Table 1 report the number of confirmed sequential consistency violations from Columns 7-9 which lead in at least one trial to a program error. That is, RELAXER was able to cause a program error by forcing the statements in a given predicted happens-before cycle to actually witness a violation of sequential consistency.

The number of observed program errors provides evidence for Hypothesis (2): RELAXER can produce real bugs involving relaxed memory models. We examine some of these bugs in greater detail.

Partial Initialization Bug(s).

Figure 10 lists simplified enqueue and dequeue methods from benchmark `ms2`. The queue is a linked list with different locks `H` and `T` for protecting the head and tail. When empty, the list contains a single, "dummy" element.

RELAXER reports two buggy cycles for this benchmark: $(2, 5, 10, 14)$ and $(3, 5, 10, 10)$. In both cases, under PSO a write in `dequeue` to a field of a newly-allocated `node` can be delayed until after the `node` is added to the list at line 5. This allows an `enqueue` from another thread to read a random, uninitialized value for `nhd->data` or `n->next`, leading to an incorrect return value or a memory error.

RELAXER found cycles exhibiting this *partial initialization* bug in every data structure benchmark. To prevent such bugs, store-store memory fences are needed—e.g., before line 5 in `ms2`.

Under PSLO, many of these bugs are possible even with added store-store fences. A thread is permitted to read the out-of-date, uninitialized value of `nhd->data` at line 14 even after reading at line 10 the more recent value of `n->next`. This models a processor speculating `n->next` and loading `n->next->data` before the load of `n->next` completes. Load-load fences are needed to prevent such bugs. (Note that cycle $(3, 5, 10, 10)$ is not in PSLO $\setminus$ PSO because `unlock` is a load-load fence.)

Read-After-Delayed-Write Bug(s).

Figure 11 lists the core of the `dekker` benchmark. The intention is that only one thread at a time can pass through to its critical section. When RELAXER predicts and creates the cycle $(1, 2, 11, 12)$ under the TSO memory model, it buffers the writes to `flag0` and `flag1`. Thus, the reads at lines 2 and 12 see `flag0` $= 0$ and `flag1` $= 0$. This both violates sequential consistency and lets the two threads erroneously enter the critical section at the same time.

This is a well know issue in Dekker's algorithm—to function

```
 struct Q {
    lock H, T;
    node *hd, *tl;
 }

 enqueue(Q, data):
1: n = new_node();
2: n->data = data;
3: n->next = NULL;
4: lock(Q->T);
5: Q->tl->next = n;
6: Q->tl = n;
7: unlock(Q->T);
```

```
 struct node {
    node* next;
    void* data;
 }
 dequeue(Q):
 8: lock(Q->H);
 9: n = Q->hd;
10: nhd = n->next;
11: if (nhd) {
12:   Q->hd = nhd;
13:   unlock(Q->H);
14:   return nhd->data
15: }
16: unlock(Q->H);
17: return NOT_FOUND;
```

Figure 10: Core of `ms2` benchmark.

```
        Initially flag0 = flag1 = 0
   thread0:                thread1:
 1: flag0 = 1;          11: flag1 = 1;
 2: while (flag1)       12: while (flag0)
 3:   if (turn) {       13:   if (!turn) {
 4:     flag0 = 0;      14:     flag1 = 0;
 5:     while (turn)    15:     while(!turn)
 6:       ;             16:       ;
 7:     flag0 = 1;      17:     flag1 = 1;
 8:   }                 18:   }
    // Critical             // Critical
    // section              // section
 9: turn = 1;           19: turn = 0;
10: flag0 = 0;          20: flag1 = 0;
```

Figure 11: Core of `dekker` benchmark.

correctly on a relaxed memory model such as TSO, a store-load fence must be placed between lines 1 and 2 and between lines 11 and 12. RELAXER similarly finds errors via cycles $(1, 2, 17, 12)$ and $(7, 2, 11, 12)$, showing that store-load fences are needed immediately after lines 7 and 17. RELAXER finds similar known issues in the Bakery algorithm, where mutual exclusion is violated if certain groups of stores are delayed past several later loads. No TSO bugs were found for the other benchmarks—we believe they are free of such read-after-delayed-write bugs.

5.5 Discussion

As RELAXER is a testing tool, it produces no false warnings. However, it can have false negatives—i.e., fail to report bugs. We next discuss some sources of this unsoundness.

First, RELAXER could fail to predict a feasible cycle. This could happen if the test harness provided is not sufficient to predict all program behaviors. In this paper we manually generated tests. Further, we used a very coarse predictive analysis to get better coverage of all feasible cycles.

Second, even if RELAXER predicts all feasible cycles, it may not be able to confirm all such cycles. There are a couple of reasons behind this. In order to create a difficult cycle: (1) a store may be required to remain buffered for a long time (unlikely in practice), and (2) a complex thread schedule may be required. For example, consider the predicted cycle $(9,\ 10,\ 12,\ 15)$ for benchmark `dekker`. This cycle is feasible, but it requires `thread0` to buffer a write to `turn` while `thread1` enters, runs, and exits its critical section and then while both threads simultaneously try to enter their critical sections again. We were not able to confirm several predicted cycles in our benchmarks, both because our coarse predictive analysis reported many infeasible cycles and because we missed some difficult-to-create cycles. (Note that we have determined manually that all 25 unconfirmed predicted cycles for `aget` are infeasible.)

Third, even if RELAXER confirms all feasible cycles, it may not be able to classify all cycles as buggy. A key reason behind this is that some of the cycles are benign and cannot lead to a bug under any circumstances. We analyzed `dekker` for benign cycles and we found that some confirmed cycles, which were not reported as buggy by RELAXER, were indeed benign. We discuss such a benign cycle in some detail below.

Benign `dekker` *Cycle.*

Consider potential happens-before cycle $(9,\ 2,\ 14,\ 15)$ for benchmark `dekker`, listed in Figure 11. This happens-before cycle is created in the trace shown in Figure 12. Initially, `thread0` is in its critical section, and `thread1` is waiting to enter. `thread0` exits its critical section then attempts to reenter. The write of 0 to `flag1` at line 14 is buffered, so that the read of `flag1` at line 2 sees an old value. Note that this cycle is feasible even with all of the necessary store-load fences added (e.g., one between lines 1 and 2).

```
     Initially flag0 = 1, turn = 0
   thread0:            thread1:
                   14: flag1 = 0 [delayed]
                   15: read turn (0)
 9: turn = 1
10: flag0 = 0
    ...
 1: flag0 = 1
 2: read flag1 (1)
          -- SC violated --
```

Figure 12: Trace of `dekker` with a benign violation of sequential consistency: $14 \rightarrow_p 15 \rightarrow_c 9 \rightarrow_p 2 \rightarrow_c 14$**.**

This violation is benign because `thread0` will wait for `thread1` to modify `turn` and `thread1` will eventually see the write of 1 to `turn` and enter its critical section. More generally, we can see that there is no need for any global ordering between lines 14 and 15 or, similarly, lines 4 and 5. (Along the same lines, it is safe for a processor, compiler, or run-time to reorder the writes at lines 9 and 10 or lines 19 and 20.)

By quickly confirming buggy cycles, RELAXER can help separate buggy cycles from benign cycles like the above example.

6. RELATED WORK

This work applies active random testing [22, 13] to predict and reproduce violations of sequential consistency in parallel programs. Several other recent techniques have been proposed to confirm potential bugs in parallel programs using random testing. Havelund et al. [2] uses a directed scheduler to confirm that a potential deadlock cycle could lead to a real deadlock. Similarly, ConTest [7] uses the idea of introducing noise to increase the probability of the occurrence of a variety of parallel bugs. CTrigger [21], instead of trying out all possible schedules, uses trace analysis to systematically identify (likely) feasible unserializable interleavings for the purpose of finding atomicity violations.

RELAXER relies on operational definitions of relaxed memory models for its testing, and we give conservative approximations for the TSO, PSO, and PSLO models. A number of researchers [4, 1, 23, 15] have developed operational definitions for TSO, PSO, and other relaxed memory models.

Recently, Flanagan and Freund proposed to use an adversarial memory to discover at run-time if a data race in a Java program could be harmful under Java's relaxed memory model [8]. For a read operation involved in a data race, the technique checks if there exists a different return value allowed by the Java Memory Model, that would cause the program to exhibit a bug.

As mentioned in Section 1, there have been a number of efforts to verify concurrent programs under relaxed memory models through explicit state model checking [6, 20, 10] or bounded model checking [9, 28, 3, 26], by translating both a test program and an axiomatic specification of a memory model into SAT. Given a finite-state program and a correctness/safety specification, Fender [12] infers the least-strict memory fences needed to meet the specification. [1] proves several decidability results for verification of finite-

```
      Initially: x = y = 0
thread1              thread2
1:  x = 1;           6:  f1();
2:  r1 = y;          7:  lock(L);
3:  lock(L);             ...
    ...              8:  unlock(L);
4:  unlock(L);       9:  y = 1;
5:  f2();           10:  r2 = x;
```

Figure 13: Program with rare SC violation under TSO

state concurrent programs under different relaxed memory models.

Run-time monitoring algorithms such as SOBER [4] and [5] scale well in practice. They need to be driven by a model checker, however, in order to find all violations of sequential consistency. For example, consider the program in Figure 13. If `f1()` and `f2()` are large, expensive functions, then the first thread will acquire lock `L` before the second thread in almost all executions. If a monitoring algorithm sees only such runs, it cannot detect the potential violation of sequential consistency. But if RELAXER sees any run of this program, it will predict the cycle $(1, 2, 9, 10)$ and can then direct the program's executions to create the sequential consistency violation. In addition, RELAXER is easily adapted to any memory model for which we can develop an operational approximation, such as PSO and PSLO, while SOBER can currently be applied only to TSO and [5] only to TSO and PSO.

Acknowledgments

We would like to thank Krste Asanović, Pallavi Joshi, Chang-Seo Park, and our anonymous reviewers for their valuable comments. This research is supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. The second author is supported in part by an Alfred P. Sloan Foundation Fellowship. Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

7. REFERENCES

[1] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *The 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, 2010.

[2] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD*, pages 41–50, 2006.

[3] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, 2007.

[4] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Computer Aided Verification (CAV)*, 2008.

[5] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

[6] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, 1993.

[7] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[8] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *Programming language design and implementation (PLDI)*. ACM, 2010.

[9] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Computer-Aided Verification (CAV)*, 2004.

[10] T. Q. Huynh and A. Roychoudhury. Memory model sensitive bytecode verification. *FMSD*, 31(3):281–305, 2007.

[11] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[12] M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE Computer Society, 2010.

[13] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multi-threaded programs through active randomized testing. In *ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*, 2010.

[14] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[15] S. Mador-Haim, R. Alur, and M. M. Martin. Generating litmus tests for contrasting memory consistency models. In *Computer Aided Verification (CAV)*, July 2010.

[16] S. Mador-Haim, R. Alur, and M. M. K. Martin. Specifying relaxed memory models for state exploration tools. In *(EC)^2: Workshop on Exploting Concurrency Efficiently and Correctly*, 2009.

[17] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Programming language design and implementation (PLDI)*. ACM, 2007.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and transformation of C Programs. In *Conference on compiler Construction (CC)*, 2002.

[19] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145. ACM, 2008.

[20] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[21] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *14th international conference on Architectural support for programming languages and operating systems*, pages 25–36. ACM, 2009.

[22] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[23] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[24] SPARC Inc. *The SPARC architecture manual (v.9)*. 1994.

[25] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Workshop on Runtime Verification (RV'02)*, volume 70 of *ENTCS*, 2002.

[26] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking axiomatic specifications of memory models. In *Programming Language Design and Implementation (PLDI)*. ACM, 2010.

[27] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engn.*, 10(2):203–232, 2003.

[28] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2004.