Dynamic Race Detection for C++11

Christopher Lidbury
Imperial College London, UK
christopher.lidbury10@imperial.ac.uk

Alastair F. Donaldson
Imperial College London, UK
alastair.donaldson@imperial.ac.uk

Abstract

The intricate rules for memory ordering and synchronisation associated with the C/C++11 memory model mean that *data races* can be difficult to eliminate from concurrent programs. Dynamic data race analysis can pinpoint races in large and complex applications, but the state-of-the-art ThreadSanitizer (tsan) tool for C/C++ considers only sequentially consistent program executions, and does not correctly model synchronisation between C/C++11 atomic operations. We present a scalable dynamic data race analysis for C/C++11 that correctly captures C/C++11 synchronisation, and uses instrumentation to support exploration of a class of non sequentially consistent executions. We concisely define the memory model fragment captured by our instrumentation via a restricted axiomatic semantics, and show that the axiomatic semantics permits exactly those executions explored by our instrumentation. We have implemented our analysis in tsan, and evaluate its effectiveness on benchmark programs, enabling a comparison with the CDSChecker tool, and on two large and highly concurrent applications: the Firefox and Chromium web browsers. Our results show that our method can detect races that are beyond the scope of the original tsan tool, and that the overhead associated with applying our enhanced instrumentation to large applications is tolerable.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords data races, concurrency, C++11, memory models

1. Introduction

With the introduction of threads of execution as a first-class language construct, the C/C++11 standards (which we henceforth refer to as C++11 for brevity) give a detailed memory model for concurrent programs [19, 20]. A principal feature of this memory model is the notion of a data race, and that a program exhibiting a data race has undefined semantics. As a result, it is important for programmers writing multi-threaded programs to take care in not introducing data races.

The definition of a data race in C++11 is far from trivial, due to the complex rules for when synchronisation occurs between the various atomic operations provided by the language, and the memory orders with which atomic operations are annotated. Working out by hand whether a program is race-free can be difficult.

[Copyright notice will appear here once 'preprint' option is removed.]

Another subtlety of this new memory model is the *reads-from* relation, which specifies the values that can be observed by an atomic load. This relation can lead to non-sequentially consistent (SC) behaviour; such *weak* behaviour can be counter-intuitive for programmers. The definition of *reads-from* is detailed and fragmented over several sections of the standards, and the weak behaviours it allows complicate data race analysis, because a race may be dependent upon a weak behaviour having occurred.

The aim of this work is to investigate the provision of automated tool support for race analysis of C++11 programs, with the goal of helping C++11 programmers write race-free programs. The current state-of-the-art in dynamic race analysis for C++11 is ThreadSanitizer [43] (tsan). Although tsan can be applied to programs that use C++11 concurrency, the tool does not understand the specifics of the C++11 memory model: it can both miss data races and errors, and report false alarms. The example programs of Figure 1 illustrate these issues: Figure 1a has a data race that tsan is incapable of detecting; Figure 1b has an assertion that can only fail under non-SC behaviour and hence cannot be explored by tsan; Figure 1c is free from data races due to C++11 fence semantics, but is deemed racy by tsan. We discuss these examples in more detail in §2.1.

In light of these limitations, the main research questions we consider are: (1) Can synchronisation properties of a C++11 program be efficiently tracked during dynamic analysis? (2) How large a fragment of the C++11 memory model can be modelled efficiently during dynamic analysis? (3) Following from (1) and (2), can we engineer a memory model-aware dynamic race analysis tool that scales to large concurrent applications, such as the Firefox and Chromium web browsers? These applications can already be analysed using tsan, without the full extent of the C++11 memory model; our question is whether by modifying tsan to be fully aware of the memory model, we can still explore said applications.

The programs we wish to analyse can have hundreds of threads running concurrently, executing thousands of lines of code. They are thus out of scope for current analysers, such as CDSChecker [31, 32] and Cppmem [6], which are designed to operate on self-contained litmus tests and small benchmarks. It is in this regard that our aims differ significantly from those of prior work.

We approach these questions through a series of research contributions as follows:

1. Extending the vector clock algorithm for C++11 (§3) We extend the vector clock-based dynamic race detection algorithm to handle C++11 synchronisation accurately, requiring awareness of *release sequences* and fence semantics. Our extension allows accurate handling of programs like those of Figures 1a and 1c.

2. Exploring weak behaviours (§4) Many C++11 weak behaviours are due to the *reads-from* relation, which allows a load to read from one of several stores. We present the design of an instrumentation library that enables dynamic exploration of this relation, capturing a large fragment of the C++11 memory model so

that errors dependent on weak behaviours can be detected, such as the assertion failure of Figure 1b.

3. Operational model (§5) We formalise the instrumentation of §4 as an operational semantics for a core language. Unlike related works on operational semantics for C/C++11 that aim to capture the full memory model (see §8), our semantics is intended as a basis for dynamic analysis of real-world applications, thus trades coverage for feasibility of implementation.

4. Characterising our operational model axiomatically (§6) The practically-focussed design of our operational model means that not all memory model behaviours can be observed. To make this precise, we characterise the behaviours we eliminate via a single additional axiom to those of an existing axiomatic formalisation of C++11, and argue that this strengthened memory model is in correspondence with our operational model.

5. Implementation in ThreadSanitizer, and experiments (§7) We have implemented our race detection and memory model exploration techniques as an extension to the ThreadSanitizer (tsan) tool. We evaluate the effectiveness of our extended tsan by comparing it with the original tsan and with CDSChecker on small benchmarks, and with the original tsan for race analysis on the Firefox and Chromium web browsers. Our results show that our extension to tsan can find data races that the original cannot, and will run large-scale applications with a tolerable overhead. However, our results emphasise the open problem of how to explain and pinpoint the root cause of data races, as well as how to determine whether data races rely on non-SC behaviour to manifest.

2. Background

We provide a brief overview of C++11 concurrency and the C/C++11 memory model (§2.1), the vector clock algorithm for data race detection (§2.2), and ThreadSanitizer, a state-of-the-art race detection tool for C++ (§2.3).

2.1 C/C++11 Memory Model

The C/C++11 standards provide several low level atomic operations on atomic types, which allow multiple threads to interact: stores, loads, read-modify-writes (RMWs) and fences. RMWs will modify (e.g. increment) the existing value of an atomic location, storing the new value and returning the previous value atomically. Fences decouple the memory ordering constraints mentioned below from atomic locations, allowing for finer control over synchronisation.

Each operation can be annotated with one of six memory orderings: relaxed, consume, acquire, release, acquire-release and sequentially consistent. These control how operations are ordered between threads and when synchronisation occurs. Sequentially consistent ordering provides the strongest ordering guarantees: if all operations are annotated as sequentially consistent then, provided the program is free from data races, it is guaranteed to have sequentially consistent semantics. The rest of the orderings provide synchronisation when certain conditions are met, with relaxed providing minimal synchronisation. In line with many prior works [8, 31, 46], for simplicity we do not further consider the scarcely used consume ordering. We also omit a treatment of lock operations, which are already handled by tsan.

We follow the *Post-Rapperswil* formalisation of Batty et al. [5] in providing an overview of the memory model. Although recent works have condensed the formalisation [8, 46], the descriptive presentation of [5] provides a greater degree of intuition for designing our instrumentation framework in §4.

We start by defining a few basic types of operation. A *load* is an atomic load or RMW. An *acquire load* is a load with acquire,

```
void T1() {
  nax = 1;                                          // A
  x.store(1, std::memory_order_release);            // B
}
void T2() {
  if (x.load(std::memory_order_acquire) == 1) // C
    x.store(2, std::memory_order_relaxed);    // D
}
void T3() {
  if (x.load(std::memory_order_acquire) == 2) // E
    nax; // read from 'nax'                    // F
}
```

(a) The write from T2 can cause T1 to fail to synchronise with T3, resulting in a data race on `nax`; tsan cannot detect the race

```
void T1() {
  x.store(1, std::memory_order_relaxed);
  y.store(1, std::memory_order_relaxed);
}
void T2() {
  assert(!(y.load(std::memory_order_relaxed) == 1) &&
           x.load(std::memory_order_relaxed) == 0));
}
```

(b) The assertion can fail as T2 can observe the writes out of order; this is not possible under SC and so cannot be detected by tsan

```
void T1() {
  nax = 1;
  atomic_thread_fence(std::memory_order_release);
  x.store(1, std::memory_order_relaxed);
}
void T2() {
  if (x.load(std::memory_order_relaxed) == 1) {
    atomic_thread_fence(std::memory_order_acquire);
    nax; // read from 'nax'
  }
}
```

(c) T1 and T2 synchronise via fences, thus there is no data race; however, tsan reports a race (a false alarm)

Figure 1: Examples showing limitations of tsan prior to our work (the statement labels `A`–`F` in Figure 1a are for reference in our vector clock algorithm example)

acquire-release or sequentially consistent ordering. A *store* is an atomic store or RMW. A *release store* is a store with release, acquire-release or sequentially consistent ordering.

The model is defined using a set of relations and predicates. An overview is given throughout the rest of this subsection.

Pre-executions A program execution represents the behaviour of a single run of the program. These are shown as execution graphs, where nodes represent memory events. For example, $\mathbf{a}{:}\mathbf{W_{rel}x{=}1}$ is a memory event that corresponds to a relaxed write of 1 to memory location $\mathbf{x}$; $\mathbf{a}$ is a unique identifier for the event. The event types $\mathbf{W}$, $\mathbf{R}$, $\mathbf{RMW}$ and $\mathbf{F}$ represent read, write, RMW and fence events, respectively. Memory orderings are shortened to $\mathbf{rlx}$, $\mathbf{rel}$, $\mathbf{acq}$, $\mathbf{ra}$, $\mathbf{sc}$ and $\mathbf{na}$ for relaxed, release, acquire, release-acquire, sequentially-consistent and non-atomic, respectively. An RMW has two associated values, representing both the value read and written. For example, $\mathbf{b}{:}\mathbf{RMW_{ra}x{=}1/2}$ shows event b reading value 1 from and writing value 2 to $\mathbf{x}$ atomically. Fences have no associated values or atomic location; an example release fence event is $\mathbf{c}{:}\mathbf{F_{rel}}$.

Sequenced-before (*sb*) is an intra-thread relation that orders events by the order they appear in the program. Operations within an expression are not ordered, so *sb* is not total within a thread.

Additional-synchronises-with (*asw*) causes synchronization on thread launch, between the parent thread and the newly created

thread. Let a be the last event performed by a thread before it creates a new thread, and b be the first event in the created thread. Then $(a, b) \in asw$. Similarly, an asw edge is also created between the last event in the child thread and the event immediately following the join in the parent thread.

The events, sb edges and asw edges form a *pre-execution*. In the program of Figure 1b, whether an event is created for the second read in `T2` depends on whether, under short-circuit semantics, it is necessary to evaluate the second argument to the logical `&&` operator. In most of the graphs we show, obvious relations like asw are elided to prevent the graphs from becoming cluttered. The values read by read events are unbound, as matching reads and writes comes at a later stage. As a result, only a select few pre-executions of a program lead to valid executions.

Presentation of Execution Graphs Throughout the paper we present a number of execution graphs, such as those depicted in Figures 2 and 3. These graphs are best viewed in colour. In each graph, events in the same column are issued by the same thread. We sometimes omit write events that give initial values to locations; e.g. in Figure 2 we label events starting with c, not showing events a and b that give initial values to locations $\mathbf{x}$ and $\mathbf{nax}$.

Witness Relations A single pre-execution, disregarding the event values, can give rise to many different executions, depending on the behaviours the program can exhibit. A pre-execution combined with a set of relations characterising the behaviour of a particular execution is referred to as a *candidate execution*. Not all pre-executions can be extended to a candidate execution, if, for example, a read cannot be matched with a write.

Reads-from (rf) shows which store each load reads from. For a store a and load b, $(a, b) \in rf$ indicates that the value read by b was written by a. In any given execution, there are usually many stores that a load can read from.

Modification-order (mo) is a total order over all of the stores to a single atomic location. Each location has its own order.

Sequentially-consistent (sc) order is a total order over all atomic operations in the execution marked with sequentially-consistent ordering. This removes a lot of the weak behaviours that a program could otherwise exhibit. For example, a sequentially consistent load will read from the last sequentially consistent store to the location, but not from an earlier sequentially consistent store.

The candidate set of executions is the set of pre-executions extended with the witness relations. At this stage, we still do not know which of the executions are allowable by the memory model.

Derived Relations Given a pre-execution and witness relations, a further set of relations can be derived that will allow us to see whether said execution follows the rules set out by the memory model.

A *release-sequence* (rs) represents a continuous subset of the modification order. It is headed by a release store, and continues along all stores to the same location. The rs is *blocked* when another thread performs a store to the location. An RMW from another thread will however continue the rs. Figure 2 shows a release sequence that is immediately blocked by a relaxed write from another thread.

A *hypothetical-release-sequence* (hrs) works in the same way as a release sequence, but is headed by both release stores and non-release stores. The rules for extending and blocking are the same as for release sequences. The hrs is used for fence synchronisation, discussed in §3.2.

Synchronises-with (sw) defines the points in an execution where one thread has synchronised with another. When a thread performs an acquire load, and reads from a store that is part of a release sequence, the head of the release sequence synchronises with the acquire load. Synchronisation is also caused by fences, discussed later in §3.2. An asw edge is also sw an edge.

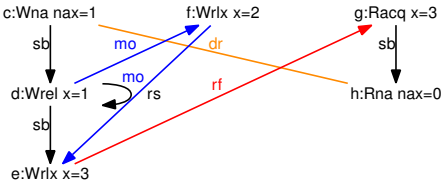


Figure 2: The release sequence headed by d is blocked by event f, causing a data race between c, the non-atomic write to **nax**, and h, the non-atomic read from **nax**; if the blocking event f is removed, there is no race

Happens-before (hb) is simply $(sb \cup sw)^+$ (where $+$ denotes transitive closure), representing Lamport's partial ordering over the events in a system [25]. Because an sw edge is also an hb edge, when thread A synchronises with thread B, every side effect that has occurred in A up to this point will become visible to every event issued by B from this point.

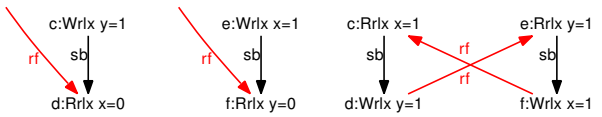
Data Races Now that we have defined the happens-before relation, we can give a formal definition of a *data race*, as described by the C/C++11 standard. A data race occurs between two memory accesses when at least one is non-atomic, at least one is a store, and neither happens before the other according to the hb relation. Figure 2 shows an execution with a data race, as there is no sw edge between the release store d and acquire load g, and therefore no hb edge between the non-atomic accesses c and h.

The presence of a data race is indicative of a program bug. The standard states that data races are undefined behaviour, and the negative consequences of data races are well known [1].

Consistent Executions The C++11 memory model is axiomatic—it provides a set of axioms that an execution must abide by in order to be exhibited by a program. A candidate execution that conforms to such axioms is said to be *consistent*. Inconsistent executions are discarded, as they should never occur when the program is compiled and executed. If any consistent execution is shown to have a data race, then the set of allowed executions is empty, leaving the program undefined.

There are seven axioms that determine consistency [5]. As we are not considering consume memory ordering and locks, some of these are fairly simple. The $well\_formed\_threads$ axiom states that sb must be intra-thread and a strict pre-order. The $well\_formed\_rf\_mapping$ axiom ensures that nothing unusual is happening with the rf relation, such as a load specified at one location reading from a store to another location, from multiple stores, or from a store whose associated value is different from the value read by the load. The $consistent\_locks$ axiom we do not consider, as locks have not been affected by our work. The last three axioms, $consistent\_sc\_order$, $consistent\_mo$ and $consistent\_rf\_mapping$, correspond with the formation of the sc, mo and rf relations. We cover these in detail in §4 when presenting our instrumentation library. The $consistent\_ithb$ axiom, without consume, simply requires hb to be irreflexive.

So long as an execution follows these axioms, it will be allowed. This leads to some interesting behaviours. We refer to a *weak* behaviour as one that would not appear under any interleaving of the threads using sequentially consistent semantics. To illustrate this, Figure 3 shows two such executions that arise from well-known litmus tests [3, 6, 9, 31]. In the load and store buffering examples, at least one of the reads will not read from the most recent write in mo, no matter how the threads are interleaved. In the



(a) Store buffering (b) Load Buffering

Figure 3: Example executions showing some of the common weak behaviours allowed by the C/C++11 memory model

load buffering example, one of the reads will read from a write that has not even been performed yet. Note that while these behaviours are allowed by the memory model, whether we observe them in practice depends practical issues such as the effect of compiler reorderings and properties of the hardware on which a program is executed.

2.2 Dynamic Race Detection

A dynamic race detector aims to catch data races while a program executes. This requires inferring various properties of the program after specific instructions have been carried out.

The vector clock (VC) algorithm is a prominent method for race detection that can be applied to multiple languages, including C++ with pthreads, and Java [15, 21, 28, 37, 38]. It aims to precisely compute the happens-before relation. Each thread in the program has an *epoch* representing its current logical time. A VC holds an epoch for each thread, and each thread has its own VC, denoted $\mathbb{C}_t$ for thread t. Each epoch in $\mathbb{C}_t$ represents the logical time of the last instruction by the corresponding thread that happens before any instruction thread t will perform in the future. The epoch for thread t, $\mathbb{C}_t(t)$, is denoted $c@t$.

VCs have an *initial value*, $\bot_V$, a *join* operator, $\cup$, and a *comparison* operator, $\leq$, and a per-thread increment operator, inc_t. These are defined as follows:

$$\bot_V = \lambda t.0 \qquad V_1 \cup V_2 \triangleq \lambda t.\max(V_1(t), V_2(t))$$
$$V_1 \leq V_2 \triangleq \forall t. V_1(t) \leq V_2(t)$$
$$inc_t(V) = \lambda u.\ \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u)$$

Upon creation of thread t, $\mathbb{C}_t$ is initialised to $inc_t(\bot_V)$ (possibly joined with the clock of the parent thread, depending on the synchronisation semantics of the associated programming language). Each atomic location m has its own VC, $\mathbb{L}_m$, that is updated as follows: when thread t performs a release operation on m, it *releases* $\mathbb{C}_t$ to m: $\mathbb{L}_m := \mathbb{C}_t$. When thread t performs an acquire operation on m, it *acquires* $\mathbb{L}_m$ using the join operator: $\mathbb{C}_t := \mathbb{C}_t \cup \mathbb{L}_m$. Thread t releasing to location m and the subsequent acquire of m by thread u simulates synchronisation between t and u. On performing a release operation, thread t's vector clock is incremented: $\mathbb{C}_t := inc_t(\mathbb{C}_t)$.

To detect data races, we must check that certain accesses to each location are ordered by hb, the happens-before relation. As all writes must be totally ordered, only the epoch of the last write to a location x needs to be known at any point, denoted W_x. As data races do not occur between reads, they do not need to be totally ordered, and so the epoch of the last read by *each* thread may need to be known. A full VC must therefore be used to track reads for each memory location, denoted $\mathbb{R}_x$ for location x; $\mathbb{R}_x(t)$ gets set to the epoch $\mathbb{C}_t(t)$ when t reads from x. To check for races, a different check must be performed depending on the type of the current and previous accesses. These are outlined as follows, where thread u is accessing location x, $c@t$ is the epoch of the last write to x and $\mathbb{R}_x$ represents the latest read for x by each thread; if any check fails then there is a race:

write-write: $c \leq \mathbb{C}_u(t)$ **write-read:** $c \leq \mathbb{C}_u(t)$
read-write: $c \leq \mathbb{C}_u(t) \wedge \mathbb{R}_x \leq \mathbb{C}_u$

Example We illustrate the VC-based race detection algorithm using the example of Figure 1a, for the thread schedule in which the statements are executed in the order `A`–`F`. Initially, the thread VCs are $\mathbb{C}_{\texttt{T1}} = (1,0,0)$, $\mathbb{C}_{\texttt{T2}} = (0,1,0)$, $\mathbb{C}_{\texttt{T3}} = (0,0,1)$, and we have $\mathbb{R}_{\texttt{nax}} = \mathbb{L}_x = \bot_V$. Because `nax` has not been written to, $W_{\texttt{nax}}$ has initial value $0@\texttt{T1}$, where the choice of `T1` is arbitrary: epoch 0 for any thread would suffice [15].

Statement `A` writes to `nax`, which has not been accessed previously, no race check is required. After `A`, $W_{\texttt{nax}} := 1@\texttt{T1}$, because `T1`'s epoch is 1. After `T1`'s release store at `B`, $\mathbb{L}_{\texttt{x}} := \mathbb{L}_{\texttt{x}} \cup \mathbb{C}_{\texttt{T1}} = (1,0,0)$, and $\mathbb{C}_{\texttt{T1}} := inc_{\texttt{T1}}(\mathbb{C}_{\texttt{T1}}) = (2,0,0)$. After `T2`'s acquire load `C`, $\mathbb{C}_{\texttt{T2}} := \mathbb{C}_{\texttt{T2}} \cup \mathbb{L}_{\texttt{x}} = (1,1,0)$. The race analysis state is not updated by `T2`'s store at `D` since relaxed ordering is used.

After `T3`'s acquire load at `E`, $\mathbb{C}_{\texttt{T3}} := \mathbb{C}_{\texttt{T3}} \cup \mathbb{L}_{\texttt{x}} = (1,0,1)$. Thread `T3` then reads from `nax` at statement `F`, thus a race check is required between this read and the write issued at `A`. A **write-read** check is required, to show that $c \leq \mathbb{C}_{\texttt{T3}}(t)$, where $W_{\texttt{nax}} = c@t$. Because $W_{\texttt{nax}} = 1@\texttt{T1}$, this simplifies to $1 \leq \mathbb{C}_{\texttt{T3}}(\texttt{T1})$, which can be seen to hold. The execution is thus deemed race-free.

In Section 3.1 we will revisit the example, showing that our refinements to the VC algorithm to capture the semantics of C++11 release sequences identify a data race in this execution.

2.3 ThreadSanitizer

ThreadSanitizer (tsan) is an efficient dynamic race detector tool aimed at C++ programs [43]. The tool originally targeted C++03 programs using platform-specific libraries for threading and concurrency, such as pthreads. The tool was designed to support C++11 atomic operations, but does not fully capture the semantics of the C++11 memory model when tracking the happens-before relation. This imprecision was motivated by needing the tool to work on large legacy programs, for which performance and memory consumption are important concerns, and the tsan developers focused on optimising for the common case of release/acquire synchronisation.

The tool performs a compile-time instrumentation of the source program, in which all (atomic and non-atomic) accesses to potentially shared locations, as well as fence operations, are instrumented with calls into a statically linked run-time library. This library implements the VC algorithm outlined in §2.2. Shadow memory is used to keep track of accesses to all locations. This will store up to four shadow words per location. For a given location this allows tsan to detect data races involving one of up to four previous accesses to the location. On each access to the location, all the shadow words are checked for race conditions, after which details of the current access are tracked using a shadow word, with a previous access being evicted pseudo-randomly if four accesses are already being tracked. Older accesses have a higher probability of being evicted. As only four of the accesses are stored, there is a chance for false negatives, as shadow words that could still be used can be evicted.

Limitations of tsan Recall from §2.1 that under certain conditions, a release sequence can be blocked. In tsan, release sequences are never blocked, and all will continue indefinitely. This creates an over-approximation of the happens-before relation, which leads to missed data races as illustrated by the example of Figure 1a. On the other hand, tsan does not recognise fence semantics and their role in synchronisation, causing tsan to under-approximate the happens-before relation and produce false positives. The example of Figure 1c illustrates this: tsan will not see the synchronisation between the two fences and so will report a data race on `nax`.

The tsan instrumentation means that every shared memory atomic load and store leads to a call into the instrumentation library, the functions of which are protected by memory barriers. These barriers mean that tsan is largely restricted to exploring only sequentially consistent executions. Only data races on non-atomic locations can lead to non-SC effects being observed. If a program has data races that can only manifest due to non-SC interactions between atomic operations (such as in the example of Figure 1b), tsan will not detect the race even if the instrumented program is executed on a non-SC architecture, such as x86, POWER or ARM.

3. Data Race Detection for C++11

The traditional VC algorithm outlined in §2.2, and implemented in tsan, is defined over simple release and acquire operations, and is unaware of the more complicated synchronisation patterns of C++11. Our first contribution is to provide an updated VC algorithm that properly handles C++11 synchronisation. Throughout this section we show where the original VC algorithm falls short, and explain how our updated algorithm fixes these shortcomings. We summarise the overall algorithm, presenting our new extensions as a set of inference rules, in §3.3.

3.1 Release Sequences

As described in §2.1, release sequences are key to synchronisation in C++11. An event a will synchronise with event b if a is a release store and b is an acquire load that reads from a store in the release sequence headed by a. We explain why this is not captured accurately by the existing VC algorithm, and how our new algorithm fixes this deficiency.

Blocking Release Sequences Recall the execution of Figure 2. The release sequence started by event d is blocked by the relaxed write at event f. The effect is that when event g reads from event e, no synchronisation occurs, as the release sequence headed by event c does not extend to event e. In the original VC algorithm, synchronisation *does* occur, as the VC for a location is never cleared; thus it is as if release sequences continue forever.

To adapt the VC algorithm to correctly handle the blocking of release sequences, we store for each location m the id of the thread that performed the last release store to m. Let $\mathbb{T}_m$ record this thread id. When a thread with id t performs a release store to m, the contents of the VC for m are over-written: $\mathbb{L}_m := \mathbb{C}_t$, and t is recorded as the last thread to have released to m: $\mathbb{T}_m := t$. This records that t has started a release sequence on m. Now, if a thread with id $u \neq \mathbb{T}_m$ performs a relaxed store to m, the VC for m is cleared, i.e. $\mathbb{L}_m := \bot_V$. This has the effect of blocking the release sequence started by $\mathbb{T}_m$.

Example revisited Recall from Section 2.2 our worked example of the VC algorithm applied to schedule `A`–`F` of Figure 1a. Revising this example to take release sequence blocking into account, we find that the relaxed store by `T2` at `D` causes $\mathbb{L}_{\mathtt{x}}$ to be set to $\bot_V$. As a result, the acquire load by `T3` at `E` yields $\mathbb{C}_{\mathtt{T3}} := \mathbb{C}_{\mathtt{T3}} \cup \mathbb{L}_{\mathtt{x}} = (0, 0, 1)$. This causes the **write-read** race check on `nax` to fail at `F`, because $W_{\mathtt{nax}} = 1@\mathtt{T1}$ and $\mathbb{C}_{\mathtt{T3}}(\mathtt{T1}) = 0$. Thus a race is detected, as required by the C++11 memory model.

Read-Modify-Writes RMWs provide an exception to the blocking rule: an RMW on location m does *not* block an existing release sequence on m. Each RMW on m with release ordering starts a *new* release sequence on m, meaning that an event can be part of multiple release sequences. If a thread t that started a release sequence on m performs a non-RMW store to m, the set of currently active release sequences for m collapses to just the one started by t. In Figure 4, release sequences from the left and middle threads

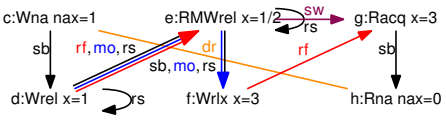


Figure 4: The release sequence started by d and continued by e is blocked by f; thus d does not synchronise with g, so c races with h

are active on event e, before a relaxed store by the middle thread causes all but its own release sequence to be blocked.

To represent multiple release sequences on a location m, we make $\mathbb{L}_m$ join with the VC for each thread that starts a release sequence. An acquiring thread will effectively acquire all of the VCs that released to $\mathbb{L}_m$ when it acquires $\mathbb{L}_m$. This is not enough however. Consider the case of collapsing release sequences when a thread t that started a release sequence on m performs a relaxed non-RMW store. We require the ability to replace $\mathbb{L}_m$ with the VC that t held when it started its release sequence on m, but this information is lost if t's VC has been updated since it performed the original release store. To preserve this information, we introduce for each location m a *vector of vector clocks* (VVC), $\mathbb{V}_m$, that stores the VC for each thread that has started a release sequence on m.

How $\mathbb{V}_m$ is updated depends on the type of operation being performed. If thread t performs a non-RMW store to m, $\mathbb{V}_m(u)$ is set to $\bot_V$ for each thread $u \neq t$. If the store has release ordering, $\mathbb{V}_m(t)$ and $\mathbb{L}_m$ are set to $\mathbb{C}_t$; as a result, t is the only thread for which there is a release sequence on m. If instead the store has relaxed ordering, $\mathbb{V}_m(t)$ is left unchanged, and $\mathbb{L}_m$ is set to $\mathbb{V}_m(t)$, i.e. to the VC associated with the head of a release sequence on m started by t, or to $\bot_V$ if t has not started such a release sequence.

Suppose instead that t performs an RMW on m. If the RMW has relaxed ordering then there are no changes to $\mathbb{L}_m$ nor $\mathbb{V}_m$ and all release sequences continue as before. If the RMW has release ordering, $\mathbb{V}_m(t)$ is updated to $\mathbb{C}_t$, and the VC for t is joined on to the VC for m, i.e. $\mathbb{L}_m := \mathbb{L}_m \cup \mathbb{C}_t$. By updating $\mathbb{L}_m$ in this manner, we ensure that when a thread acquires from m, it synchronises with all threads that head a release sequence on m.

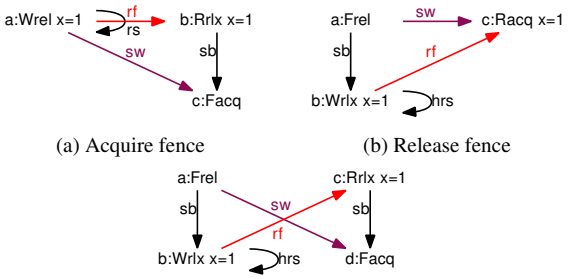
In practice, recording a full VVC for each location would be prohibitively expensive. In our implementation (§7.1) we instead introduce a mapping from thread ids to VCs that grows on demand when threads actually perform RMWs.

3.2 Fences

A fence is an atomic operation that does not work on any particular location. It is annotated with a memory ordering like other atomic operations, and thus can be a release fence and/or acquire fence. Fences with SC ordering have special meaning, discussed in §4.5. As discussed above, fences are not handled in tsan: programs such as that of Figure 1c will not be properly instrumented, leading to false positives.

The three cases of synchronisation with fences are shown in Figure 5. Acquire fences will synchronise if a load sequenced before the fence reads from a store that is part of a release sequence, even if the load has relaxed ordering, as shown in Figure 5a. Release fences use the hypothetical release sequence, described in §2.1. A release fence will synchronise if an acquire load reads from a hypothetical release sequence that is headed by a store sequenced after the fence, as shown in Figure 5b. Release fences and acquire fences can also synchronise with each other, shown in Figure 5c.

In order to allow the VC algorithm to handle fence synchronisation, the VC from whence a thread performed a release fence must be known, as this VC will be released to $\mathbb{L}_m$ if the thread then does a relaxed store to m. When a thread performs a relaxed load, the



(a) Acquire fence (b) Release fence

(c) Acquire and release fences

Figure 5: Synchronisation caused by fences

VC that would be acquired if the load had acquire ordering must be remembered, because if the thread then performs an acquire fence, the thread will acquire said VC. To handle this, for each thread t we introduce two new VCs to track this information: the *fence release clock*, $\mathbb{F}_t^{rel}$, and the *fence acquire clock*, $\mathbb{F}_t^{acq}$. We then extend the VC algorithm as follows. When thread t performs a release fence, $\mathbb{F}_t^{rel}$ is set to $\mathbb{C}_t$; when t performs an acquire fence, $\mathbb{F}_t^{acq}$ is joined on to the thread's clock, i.e. $\mathbb{C}_t := \mathbb{C}_t \cup \mathbb{F}_t^{acq}$. When a thread t performs a relaxed store to m, $\mathbb{F}_t^{rel}$ is joined on to $\mathbb{L}_m$. If t performs a relaxed load from m, $\mathbb{L}_t$ is joined on to $\mathbb{F}_t^{acq}$.

To illustrate fence synchronisation, consider the four operations shown in the execution fragment in Figure 5c. Let events a, b, c and d be carried out in that order. After a, $\mathbb{F}_t^{rel} = \mathbb{C}_t$. After b, $\mathbb{L}_x = \mathbb{F}_t^{rel}$. After c, $\mathbb{F}_u^{acq\prime} = \mathbb{F}_u^{acq} \cup \mathbb{L}_x$. Finally, after d, we have $\mathbb{C}'_u = \mathbb{C}_u \cup \mathbb{F}_u^{acq\prime} \geq \mathbb{C}_u \cup \mathbb{F}_t^{rel} = \mathbb{C}_u \cup \mathbb{C}_t$. Thus we have synchronisation between a and d.

3.3 Algorithm

Our extended VC algorithm, combining the original VC algorithm of [15] with the techniques described in §3.1 and §3.2 for handling release sequences and fences, is summarised by the inference rules of Figure 6. We omit the rules for reads and writes on non-atomic locations, which are unchanged.

For each thread t the algorithm records a vector clock $\mathbb{C}_t$, and fence release and acquire clocks, $\mathbb{F}_t^{rel}$ and $\mathbb{F}_t^{acq}$ (see §3.2). For each variable m, both a vector clock $\mathbb{L}_m$ and vector of vector clocks $\mathbb{V}_m$ (see §3.1) are recorded. We use $\mathbb{C}$, $\mathbb{F}^{rel}$ and $\mathbb{F}^{acq}$, and $\mathbb{L}$ and $\mathbb{V}$, to denote these clocks across all threads and locations, respectively.

Observe that $\mathbb{F}^{rel}$ and $\mathbb{F}^{acq}$ are only significant when relaxed ordering is used; they do not introduce any new information in the presence of release and acquire semantics. The fence VCs are also never stored in the VVC, because if a thread performs a relaxed store requiring the VVC to collapse, $\mathbb{F}^{rel}$ will need to be joined onto the VC for the location regardless.

For clarity, many optimisations to the algorithm, incorporated in our implementation (see §7.1) are omitted from the presentation of Figure 6. Appendix A in the extended version of the paper presents the optimised algorithm [27]. As an example, the VVC does not need to be used until there are two active release sequences.

4. Exploring Weak Behaviours

The fact that the C++11 memory model allows non-SC behaviours poses a problem for data race detection techniques: a tool such as tsan that only considers SC executions will not be able to explore

STATE:

$\mathbb{C} : Tid \to VC$
$\mathbb{L} : Var \to VC$
$\mathbb{V} : Var \to (Tid \to VC)$

$\mathbb{F}^{rel} : Tid \to VC$
$\mathbb{F}^{acq} : Tid \to VC$

STORES and RMWs:

[RELEASE STORE]

$$\frac{\mathbb{L}' = \mathbb{L}[x := \mathbb{C}_t] \qquad \mathbb{V}' = \mathbb{V}[x := \emptyset[t := \mathbb{C}_t]]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{store_{rel}(x,t)} (\mathbb{C}, \mathbb{L}', \mathbb{V}', \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

[RELAXED STORE]

$$\frac{\mathbb{L}' = \mathbb{L}[x := \mathbb{V}_x(t) \cup \mathbb{F}_t^{rel}] \qquad \mathbb{V}' = \mathbb{V}[x := \emptyset[t := \mathbb{V}_x(t)]]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{store_{rlx}(x,t)} (\mathbb{C}, \mathbb{L}', \mathbb{V}', \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

[RELEASE RMW]

$$\frac{\mathbb{L}' = \mathbb{L}[x := \mathbb{L}_x \cup \mathbb{C}_t] \qquad \mathbb{V}' = \mathbb{V}[x := \mathbb{V}_x[t := \mathbb{C}_t]]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{rmw_{rel}(x,t)} (\mathbb{C}, \mathbb{L}', \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

[RELAXED RMW]

$$\frac{\mathbb{L}' = \mathbb{L}[x := \mathbb{L}_x \cup \mathbb{F}_t^{rel}]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{rmw_{rlx}(x,t)} (\mathbb{C}, \mathbb{L}', \mathbb{V}', \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

LOADS (an RMW also triggers a LOAD rule initially)**:**

[ACQUIRE LOAD]

$$\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \cup \mathbb{L}_x]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{load_{acq}(x,t)} (\mathbb{C}', \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

[RELAXED LOAD]

$$\frac{\mathbb{F}^{acq\prime} = \mathbb{F}^{acq}[t := \mathbb{F}_t^{acq} \cup \mathbb{L}_x]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{load_{rlx}(x,t)} (\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq\prime})}$$

FENCES:

[RELEASE FENCE]

$$\frac{\mathbb{F}^{rel\prime} = \mathbb{F}^{rel}[t := \mathbb{C}_t]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{fence_{rel}(t)} (\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel\prime}, \mathbb{F}^{acq})}$$

[ACQUIRE FENCE]

$$\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \cup \mathbb{F}_t^{acq}]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{fence_{acq}(t)} (\mathbb{C}', \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

Figure 6: Semantics for tracking the happens-before relation with loads, stores, RMWs and fences

these additional behaviours. For example, tsan cannot detect errors associated with non-SC executions of the program of Figure 1b.[1]

To address this, we now present the design of a novel library that allows a program to be instrumented, at compile time, with auxiliary state that can enable exploration of a large fragment of the non-SC executions allowed by C++11. The essential idea is as follows: every atomic store is intercepted, and information relating to the store is recorded in a *store buffer*. Every atomic load is also intercepted, and the store buffer is queried to determine the set of possible stores that the load may acceptably read from.

By controlling the order in which threads are scheduled and the stores from which atomic load operations read, our instrumentation enables exploration of a large set of non-SC behaviours. Our

[1] It is possible that compiler optimisations applied at instrumentation-time might induce non-SC behaviours, e.g. by reordering memory accesses. In this case, tsan would explore SC behaviours of the transformed program.

buffering-based approach has some limitations, for example it does not facilitate a load reading from a store that has not yet been issued; we formalise the exact fragment of the memory model covered by our technique in §6.2. We use this instrumentation as a basis for extending the tsan tool for detection of data races arising from non-SC program executions by randomising the stores that are read from by atomic loads (see §7).

We now give an overview of our instrumentation. In §5 we formalise the instrumentation using an operational semantics.

4.1 Preliminaries

As stated in §2.1, we follow closely the Post-Rapperswil memory model presentation of Batty et al. [5] in the design of our instrumentation library. We use the notation "§PRX" to refer to section X of the Post-Rapperswil formalisation.

Going back to the witness relations described in §2.1, it is these relations that differentiate one run of a program from another. We wish to be able to explore all the possible arrangements of these relations, while pruning those that are inconsistent. For example, consider a program that has a single location written to four times, split between two threads. There are 24 (4!) ways in which the mo relation can be arranged, although only 6 of these will be consistent.

As we will see in this section, the different arrangements of mo and sc can be handled by exploring different thread schedules. It is the rf relation that is difficult to explore, as this requires us to know all the stores that each load could read from. We will therefore introduce the notion of a *software store buffer*.

We assume throughout that the operations issued by a thread are issued in program order; this is a standard constraint associated with instrumentation-based dynamic analysis. Under this assumption, the operations of each thread are ordered by the sb relation. We treat this as an axiom, and refer to it as **AxSB**. We also assume that the order in which sequentially consistent operations are carried out conforms with the sc relation, which we refer to as **AxSC**. In fact, as we will see in §6.2, the order in which we carry out operations conforms to all of the relations, and therefore each relation conforms to every other relation. We will be brief on axioms that require showing conformance with certain relations, but nonetheless, these will be useful in showing that our instrumentation follows the C++11 memory model.

4.2 Post-Store Buffering

Consider the case where a thread performs an atomic store to an atomic location. Depending on the state of the thread and the memory order used, atomic loads should be able to read from this store, even if there has been an intervening store to the same location. We will therefore record the atomic stores to each location in a buffer, allowing the instrumentation library to search though and pick a valid store to read from.

Our approach to instrumenting stores is as follows. On intercepting a store to location m, the VC updates described in §3 are performed, to facilitate race checking. The value to be stored to m is then placed in the store buffer for m. Each individual store in the store buffer is referred to as a *store element*, and contains a snapshot of the state of the location at the time the store was performed. This snapshot includes the meta-data required to ensure that each load can be certain that reading from the store will lead to a consistent execution. We explain the meta-data that constitutes a store element throughout this section, guided by the C++11 consistency axioms. We then formally define the store buffer in §5.

4.3 Consistent Modification Order (§PR6.17)

The consistent mo axiom states: (1) mo is a strict total order over all the writes to each location. (2) That hb restricted to the writes at a location is a subset of mo. (3) Restricting the composition of (sb ; $\mathbf{F}_{sc}$; sb) to the writes at a location is a subset of mo.

The store elements for a location is an ordered list, with each store to m creating a store element at the back. This represents mo for the location as a strict total order, satisfying (1).

To satisfy (2), we need to show that mo conforms with hb, which is the transitive closure of sb and sw, thus we need to show conformance with each of sb and sw. We already know from the **AxSB** axiom that mo conforms with sb. Synchronisation follows the rf relation (and sb when fences are involved), and as a load can only read from a store already in the store buffer, mo must conform with rf. So (2) is satisfied. The agreement between *mo* and *hb* shown here is also referred to as coherence of write-writes (CoWW).

As we have **AxSB** and **AxSC**, (3) holds trivially.

4.4 Consistent SC Order (§PR6.16)

Consistency of sc requires that sc be a strict total order over all events with sc ordering, and that hb and mo conform with sc. While tsan does not explicitly track the sc relation, our instrumentation uses global state to track properties of threads as they execute SC operations, which we introduce in §4.5. Access to this global state is mutex-protected, which implicitly induces a total order on SC operations. Conformance with hb and mo follows the same reasoning as that given in §4.3, so we omit it here.

4.5 Consistent Reads From Mapping (§PR6.19)

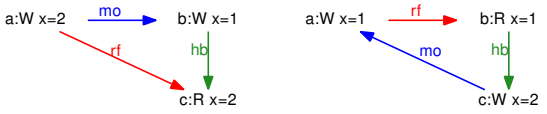
The rf requirements are the most complex out of the consistency rules. We have broken them down into three groups. The methods described in this section collectively give rise to an algorithm for determining the set of possible stores that a load can read from; this algorithm is presented formally in Figure 12 and discussed in §5.

Coherence Rules There are four coherence rules. We have already covered CoWW in §4.3, so we only discuss the other three.

(1) Coherence of Write-Reads (CoWR) states that a load cannot read from a store if there is another store later in mo such that said store happens before the current load. This essentially cuts off all of the mo before such stores.

(2) Coherence of Read-Writes (CoRW) states that a load cannot read from a store if there is another store earlier in mo that happens after the current load. This will cut off all of the mo after such stores. More formally, this states that $rf \cup hb \cup mo$ is acyclic.

The following illustrates the behaviours these rules forbid:



CoWR CoRW

These two rules leaves us with a range of stores in the mo that can potentially be read from.

Our instrumentation library automatically conforms to CoRW. This is because violating CoRW would require a thread to read from a store that has not yet been added to the store buffer for a location, something our instrumentation does not allow. This is illustrated by the execution fragment shown for CoRW above. This reasoning also assumes that we follow the hb relation.

For CoWR, each store element must record sufficient information to allow a thread issuing a load to determine whether the store happened before the load. To enable this, the id of the storing thread must be recorded when a store element is created, together with the epoch associated with the thread when the store was issued. When a load is issued, our instrumentation library can then search the store buffer to find the latest store in mo that happened before the cur-

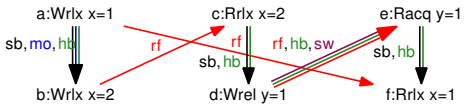


Figure 7: Inconsistent execution fragment caused by lack of CoRR

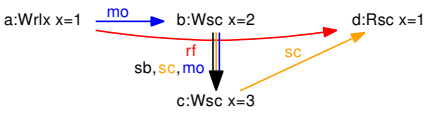


Figure 8: Consistency of sc-reads only forbids d reading from b

rent load; all stores prior to the identified store are cut off from the perspective of the loading thread. This is achieved by searching the buffer backwards, from the most recent store. For a given store element, let $c@t$ be the epoch of the thread that performed the store. With $\mathbb{C}$ denoting the VC of the loading thread, if $c \leq \mathbb{C}(t)$, then the store will happen before the load, so we halt the search.

We also have (3) Coherence of Read-Reads (CoRR). This states that if two reads from the same location are ordered by hb, the reads cannot observe writes ordered differently in mo. As a consequence, if a thread performs a load from a location and reads from a particular store element, all of the mo before said store is cut off for future loads. Loads from other threads will also be affected when synchronisation occurs. Consider the execution fragment shown in Figure 7. The two loads c and f are ordered by hb due to synchronisation between d and e. This means they must observe the two stores a and b in the same order, else read from the same stores. In this particular example, they do not, meaning the fragment will lead to an inconsistent execution.

To ensure CoRR, it is thus necessary for a thread to be aware of loads performed by other threads. To handle this, we equip our instrumentation library with software load buffers as follows. We augment every store element with a list of *load elements*. When a thread reads from a store element, a new load element is created and added to the list of load elements associated with said store element. Each load element records the id of the thread that issued the load, and the epoch associated with the thread when the load was issued. Whenever our instrumentation library is searching through the store buffer for the earliest store that a load is allowed to read from, it must also search through all the load elements associated with each store element. For a load element under consideration, let $c@t$ be the epoch of the thread that carried out the load, and $\mathbb{C}$ the VC of the thread that is currently performing a load. If $c \leq \mathbb{C}(t)$, then the load associated with the load element happened before the current load, and we must halt the search.

Not every load that has been issued needs to have an associated load element. For example, if a thread loads twice from a location without issuing an intervening release operation, the first load will not affect any other thread and thus can be pruned. Our implementation (§7.1) incorporates several such optimisations.

Finally, we have (4) consistent RMW reads. If an RMW event b reads from write event a, then b must follow a in mo. With our instrumentation library, an RMW will read from the back of the store buffer before adding a store element to the back. As the ordering of the store elements follows mo, (4) is satisfied.

Sequentially Consistent Fences SC fences add a layer of complexity to what the memory model allows. An SC fence will interact with other SC fences and reads in a number of ways. These are outlined as follows, where $\overset{sc}{\dashrightarrow}$ denotes an inter-thread sc edge:

(5) $\mathbf{W}_{\textbf{non-}SC} \overset{sb}{\longrightarrow} \mathbf{F}_{SC} \overset{sc}{\dashrightarrow} \mathbf{R}_{SC}$: The SC read must read from the last write sequenced before the SC fence, or any write later in modification order. Non-SC reads are unaffected.

(6) $\mathbf{W}_{SC} \overset{sc}{\dashrightarrow} \mathbf{F}_{SC} \overset{sb}{\longrightarrow} \mathbf{R}_{\textbf{non-}SC}$: The non-SC read must read from the SC write, or a write later in modification order. If there is no SC write, then the read is unaffected.

(7) $\mathbf{W}_{\textbf{non-}SC} \overset{sb}{\longrightarrow} \mathbf{F}_{SC} \overset{sc}{\dashrightarrow} \mathbf{F}_{SC} \overset{sb}{\longrightarrow} \mathbf{R}_{\textbf{non-}SC}$: Any read sequenced after the SC fence must read from the last write sequenced before the SC fence, or a write later in modification order.

Accommodating SC fences in our instrumentation library is not trivial, requiring additional VCs and VC manipulation on every SC operation. We begin by defining two global VCs: $\mathbb{S}_F$, representing the epoch of the last SC fence performed by each thread, and $\mathbb{S}_W$, the epoch of the last SC write performed by each thread. Each thread will update its position in these VCs whenever they perform an SC fence or SC write.

Each thread t now has an extra three VCs: $\$_{F,t}$, $\$_{W,t}$ and $\$_{R,t}$. Each VC will control each of the three cases outlined above. These are updated when the thread performs an SC operation. When a thread performs an SC fence, it will acquire the two global SC VCs: $\$_{F,t} := \$_{F,t} \cup \mathbb{S}_F$ and $\$_{W,t} := \$_{W,t} \cup \mathbb{S}_W$. When a thread performs an SC read, it will acquire the global SC fence VC in the following way: $\$_{R,t} := \$_{R,t} \cup \mathbb{S}_F$. To see how this enforces the rules outlined above, consider a thread t that is performing an atomic load on location x. While searching back through the buffer, we have reached a store performed by thread u at epoch $c@u$. If the load is an SC load, and $\$_{R,t}(u) \geq c$, then we halt the search according to (5). If the store is an SC store, and $\$_{W,t}(u) \geq c$, then we halt the search according to (6). Regardless of whether the load or the store is SC, if $\$_{F,t}(u) \geq c$ then (7) applies.

We now cover the obvious missing case. (8) $\mathbf{W}_{SC}$**;** $\mathbf{R}_{SC}$: The SC read must read from the last SC write, a write later in mo than the last SC write, or a non-SC write that does not happen before some SC write to the same location. Figure 8 shows an execution fragment where the SC write of c blocks the SC read of d from reading from b, but not a.

This case is not covered by the machinery discussed earlier, as an SC write will update $\mathbb{S}_W$, but an SC read will acquire $\mathbb{S}_F$. To handle this, each store element must be marked with a flag indicating whether it was an SC store. Additionally, every store element that happens before the current store must also be marked as a SC store. When an SC load has searched back through the buffer and found the earliest feasible store to read from, it may read from any store element that is unmarked, or the last marked element.

Note that how we handle (8) does not affect (6), as if a later SC write has marked an earlier non-SC write as being SC, then that later write will block any thread from reaching the earlier write.

Visible Side Effects We do not cover these rules in detail, as they do not impact instrumentation much. In brief, a load must read from a store, or a store later in mo, where said store happens before the load. There can be at most one visible side effect for any load, which is already captured by (1). This can lead to cases where there are no visible side effects for a given load, due to locations being initialised from another thread which has yet to synchronise with. Locations that are initialised by the global thread will therefore overcome this issue.

5. Operational Model

We now formalise the instrumentation of §4 as operational semantics for a core language. As well as making our approach precise,

```
Prog   ::= Stmt ; ϵ
Stmt   ::= Stmt ; Stmt
         | if (LocNA) {Stmt} else {Stmt}
         | LocNA := Expr
         | LocNA = Fork(Prog)
         | Join(LocNA)
         | StmtA
         | ϵ
StmtA  ::= LocNA = Load(LocA, MO)
         | Store(LocNA, LocA, MO)
         | RMW(LocA, MO, F)
         | Fence(MO)
MO     ::= relaxed | release | acquire
         | rel_acq | seq_cst
Expr   ::= <literal> | LocNA | Expr op Expr
```

Figure 9: Syntax for our core language

this allows us to argue in §6 that our instrumentation matches an axiomatically-defined fragment of the C++11 memory model.

5.1 Programming Language Syntax

We present our formal operational model with respect to a core language that captures the atomic instructions defined by C++11, the syntax for which is described by the grammar of Figure 9.

A program is a sequence of statements that are executed by an initial thread. We use `LocA` and `LocNA` to denote disjoint sets of atomic and non-atomic locations, respectively. The forms of simple statement are: assigning the result of an expression over non-atomic locations to a non-atomic location (we leave the set of operators that may appear in expressions unspecified); forking a new thread, capturing a handle for the thread in a non-atomic location (similar to C++'s `std::thread`); joining a thread via its handle; and performing an atomic operation. Atomic operations, described by the `StmtA` production rule, consist of loads, stores, RMWs and fences. An `RMW` takes a functor, `F`, to apply to an atomic location, for example, the increment function.

The language supports compound `if` statements; loops are omitted for simplicity. An empty statement is represented by ϵ.

5.2 Operational Model Formalised

The structure of the state of a program is shown in Figure 10. It describes the set of possible states a program can be in, and includes the machinery described in §5 that allows us to explore weak behaviours. Figure 10b gives us a pictorial representation of the state, giving us an intuitive view of how the state described formally in Figure 10a is laid out.

The state of the system comprises of the set of threads, global vector clocks for handling SC fences, and mappings from memory locations to either the value stored in the location, or the atomic information associated with the location, depending on whether the location is atomic or not. The set of atomic and non-atomic locations are disjoint ($LocA \cap LocNA = \emptyset$). $ALocInfo$ holds the information for store buffering and race detection. $Prog$ is a program expressed using the syntax of Figure 9.

The initial state of the program will have empty mappings for atomic and non-atomic locations, and the VCs for the SC fences will be $\bot_V$. There will just be a single thread representing the program's main function. Formally, let the main thread be denoted M, the initial state will be $\Sigma = ([M], \emptyset, \emptyset, \bot_V, \bot_V)$. The initial state of M will have $\mathbb{C}$ initialised to $inc_t(\bot_V)$ and its three SC fence VCs initialised to $\bot_V$, t will be a random identifier and P will be the entire program.

The race detection machinery has been left out for clarity, but note that the race analysis and store buffering both use the threads VC ($\mathbb{C}$) and the VC for the atomic location ($\mathbb{L}$).

5.3 Operational Semantics

Figures 11 to 13 show the state transitions for our operational model. They are defined for each atomic instruction in our simple language, as well as for a few internal instructions that do not appear in source programs. Details of the non-atomic instructions appear in Appendix B [27].

A system under evaluation is a triple of the form (Σ, ss, T). The state of the system is represented by Σ, as shown in Figure 10. The program being executed is ss, with the $ThrState$ of the thread running the program being T. A thread will only update its own state when executing a program, so T will change as ss is executed. This will cause the $ThrState$ for the current thread in Σ to become stale, but will refresh upon a context switch.

Figure 11 gives the semantics for atomic statements. Each atomic function will call into the appropriate sequentially consistent helper function of Figure 13, and the appropriate buffer implementation functions. These SC helpers perform the updates described in the SC fence section of §4.3, or nothing, if the memory ordering is not `seq_cst`.

Each atomic function will first call into the VC algorithm described in §3, as shown by calls to functions of the form $[X]$ that correspond with the inference rules in Figure 6. The state used by the VC algorithm has a different representation, that makes it easier to compare with other VC algorithms; Appendix B details how to convert between the two representations [27].

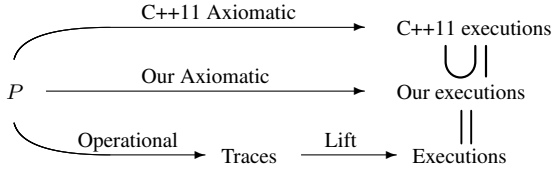
The buffer implementation functions `Store` and `Load` carry out the store buffering and load buffering. These are not directly used by the programmer, rather, they are used by the other atomic functions to carry out shared functionality. The load implementation takes a store buffer element to load from. If an RMW is being evaluated, then this element is simply the last in the buffer. For atomic loads, an element is non-deterministically chosen from a reads-from set, computed using the ReadsFromSet helper function (Figure 12), which uses the consistent reads-from of §4.3. The ++ operator represents list concatenation.

6. Characterising Our Model Axiomatically

We designed the instrumentation strategy of Section 4, formalised by the operational model of Section 5, by considering the sorts of non-SC behaviour that would be feasible to explore in an efficient dynamic analysis tool. However, the intricacy of the operational rules make it difficult to see, at a high level, which behaviours are allowed vs. forbidden by our model. We provide a clearer high-level picture of this by devising an axiomatic memory model that precisely describes the behaviours that our operational semantics allows, and show that the axioms strengthen those of C++11.

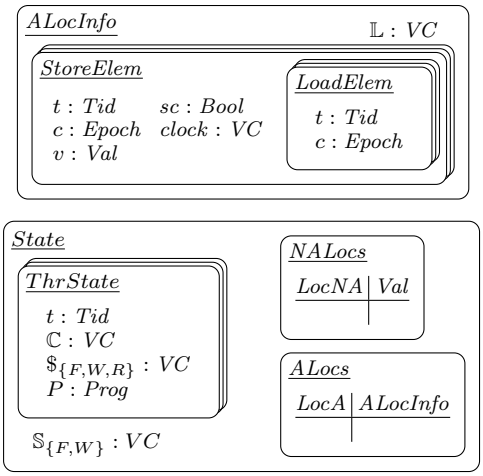
We first show how to lift a trace given by our operational model to an execution. This lifting procedure intuitively gives rise to additional axioms to those of C++11, which form our axiomatic memory model. Because our axiomatic model consists of the C++11 axioms plus an additional axiom, our axiomatic model is strictly stronger than that of C++11. We then argue that the executions given by lifting the set of traces produced by our operational model exactly match the executions captured by our axiomatic model.

The following diagram summarises what we wish to show:



$$
\begin{aligned}
Tid &\triangleq \mathbb{Z} \quad Epoch \triangleq \mathbb{Z} \quad Val \triangleq \mathbb{Z} \\
VC &\triangleq Tid \to Epoch \\
ThrState &\triangleq (t : Tid) \times (\mathbb{C} : VC) \times \\
&\quad (\$_{\{F,W,R\}} : VC) \times (P : Prog) \\
LoadElem &\triangleq (t : Tid) \times (c : Epoch) \\
StoreElem &\triangleq (t : Tid) \times (c : Epoch) \times (v : Val) \times \\
&\quad (sc : Bool) \times (clock : VC) \times (loads : LoadElem \text{ set}) \\
StoreBuffer &\triangleq StoreElem \text{ list} \\
ALocInfo &\triangleq (\mathbb{L} : VC) \times StoreBuffer \\
ALocs &\triangleq LocA \to ALocInfo \\
NALocs &\triangleq LocNA \to Val \\
State &\triangleq ThrState \text{ list} \times ALocs \times NALocs \times (\mathbb{S}_{\{F,W\}} : VC)
\end{aligned}
$$

(a) Formal definition



(b) Pictorial definition

Figure 10: Operational State

Notation Let P denote a program written in our language. The set of executions allowable for P according C++11's axiomatic memory model is denoted $consistent(P)$. Our operational model takes program P and produces a set of traces, denoted $traces(P)$. We use σ to denote an individual trace, which is a finite sequence of state transitions of the form $s_1 \to s_2 \to ... \to s_k$. For a given trace σ, let $lift(\sigma)$ denote the lifting of σ to an axiomatic style execution. For a set of traces S, we define $lift(S) = \{lift(\sigma) \mid \sigma \in S\}$, which is the application of $lift$ to each trace in S. Therefore, $lift(traces(P))$ gives the set of executions that can be obtained by running P on our operational model.

6.1 Lifting Traces

Before we can define our axiomatic model, it must be clear how a trace is lifted to an axiomatic execution. We must first extend our operational state with auxiliary labels to track events. We define a label as: $Label \triangleq \{a, b, c, \dots\} \cup \{\bot\}$. Each load and store element will have a label representing the event id. Each $ThrState$ will have a *last sequenced before* (lsb) label and the $State$ a *last sequentially consistent* (lsc) label that enables tracking of the sb and sc relations, as explained below. The $ThrState$ will additionally have an *last additional synchronises with* ($lasw$) label, that enables us to see the last event a forking thread performed before the thread, as lsb may have updated before the new thread has begun. Including this information allows us to create an execution by inspection of the trace and resulting state. We present this in detail below.

To begin with, consider the four event types used in executions: **R**, **W**, **RMW** and **F**. These correspond with the `Load`, `Store`, `RMW` and `Fence` instructions shown in Figure 9. Reads and writes with non-atomic orderings correspond with `Read` and `Write`. The labels inside the $LoadElem$ and $StoreElem$s created by the load and store instructions will match the event ids of their corresponding events in the execution. The RMW instruction will create both a $LoadElem$ and a $StoreElem$, both of which will have the same label. Fences do not create any state, but will be assigned an event and label upon inspection of the trace.

We give a short description on how to lift event relations. Instruction here refers to just those that create events.

An sb edge is created when a thread T performs an instruction and $T.lsb \neq \bot$. The rf edges can be created by inspection of the trace, by seeing which $StoreElem$ a load reads from. The mo can be easily seen from the order of the $StoreElem$s in the store buffer. For sc, an edge will be drawn from $\Sigma.lsc$ to the next instruction with sequentially consistent ordering, as long as $\Sigma.lsc \neq \bot$.

The asw edges are created in a couple of ways: when a thread T performs a `Fork`, creating thread T', T' stores $T.lsb$ in $T'.lasw$. When T' performs an instruction, $T'.lasw \neq \bot$ and $T.lsb = \bot$, an asw edge is created. Alternatively, when thread T' has finished, thread T created thread T' and performs a `Join` with $T'.tid$, $T'.lsb \neq \bot$ and T performs and instruction.

All other relations are derived from the events and these five relations, thus, do not need to be explicitly tracked with any auxiliary state or the lifting function.

6.2 Restricted Axiomatic Model

Now that we can see how our operational model relates to executions, we can reason about the behaviours our model can exhibit.

We notice that the direction of all the relations is in the order they are created:

$$\xrightarrow[\mathbf{co},\, sb,\, asw,\, rf,\, mo,\, sc]{s_1 \to s_2 \to ... \to s_k}$$

co represents the *commitment order*, it is the order in which events are added to an execution as a program is running [30]. Assume that we have a partial trace, σ_i and a corresponding partial execution, E_i. When we advance σ_i to produce σ_{i+1}, possibly adding event e_{i+1} to E_i to produce E_{i+1}, we can see from the lift function that there can be no edges of the form $(e_{i+1}, e_{j \leq i})$ in any of our relations, but there can be $(e_{j \leq i}, e_{i+1})$, hence all the relations must conform.

Let $rConsistent(P)$ be the set of executions allowable for P according to our axiomatic model. This is defined as follows:

$$
\begin{aligned}
rConsistent(P) = consistent(P) \wedge \\
acyclic(sb \cup asw \cup rf \cup mo \cup sc)
\end{aligned}
$$

Acyclicity is due to all the relations conforming. For there to be a cycle, one of the edges must go back in the commitment order. This extra axiom prohibits behaviours that require a load to read from a store that has yet to be committed, such as load buffering.

6.3 Equivalence of Operational and Axiomatic Models

We argue that the set of executions a program P can exhibit under our restricted axiomatic model is equal to the set of executions we get by lifting the set of traces that our operational model can

ATOMIC STATEMENTS:

[ATOMIC LOAD]

$$\frac{\begin{array}{c}(\Sigma, T, mo) \rightarrow_{load} (\Sigma, T') \\ S \in \mathsf{ReadsFromSet}(\Sigma.ALocs(a), mo, T') \\ T'' = [\text{LOAD}](S, mo, T')\end{array}}{(\Sigma, l = \texttt{Load}(a, mo); ss, T) \rightarrow (\Sigma, l = \underline{\texttt{Load}}(a, mo, S); \delta; ss, T'')}$$

[ATOMIC STORE]

$$\frac{\begin{array}{c}(\Sigma, T, mo) \rightarrow_{store} (\Sigma', T) \\ (A', T') = [\text{STORE}](\Sigma'.ALocs(a), mo, T) \\ \Sigma'' = \Sigma'[ALocs := \Sigma'.ALocs[a := A']]\end{array}}{(\Sigma, \texttt{Store}(l, a, mo); ss, T) \rightarrow (\Sigma'', \underline{\texttt{Store}}(l, a, mo); \delta; ss, T')}$$

[ATOMIC RMW]

$$\frac{\begin{array}{c}(\Sigma, T, mo) \rightarrow_{load} (\Sigma, T') \qquad (\Sigma, T, mo) \rightarrow_{store} (\Sigma', T) \\ l \text{ is fresh} \qquad S = \Sigma.ALocs(a).SE.back \\ (A', T'') = [\text{RMW}](\Sigma.ALocs(a), mo, T') \\ \Sigma'' = \Sigma'[ALocs := \Sigma'.ALocs[a := A']]\end{array}}{\begin{array}{l}(\Sigma, \texttt{RMW}(a, mo, \texttt{F}); ss, T) \rightarrow \\ (\Sigma'', l = \underline{\texttt{Load}}(a, mo, S); l = \texttt{F}(l); \underline{\texttt{Store}}(l, a, mo); \delta; ss, T'')\end{array}}$$

[ATOMIC FENCE]

$$\frac{(\Sigma, T, mo) \rightarrow_{fence} (\Sigma', T') \qquad T'' = [\text{FENCE}](mo, T')}{(\Sigma, \texttt{Fence}(mo); ss, T) \rightarrow (\Sigma', \delta; ss, T'')}$$

[ATOMIC LOAD IMPL]

$$\frac{\begin{array}{c}ld.t = T.t \qquad ld.c = T.\mathbb{C}(T.t) \\ S' = S[LD := S.LD \cup \{ld\}] \qquad \Sigma.ALocs(a).SE = L{+}{+}[S]{+}{+}R \\ \Sigma' = \Sigma[ALocs := \Sigma.ALocs[a := \Sigma.ALocs(a)[SE := \\ L{+}{+}[S']{+}{+}R]]] \\ \Sigma'' = \Sigma'[NALocs := \Sigma'.NALocs[l := S.v]]\end{array}}{(\Sigma, l = \underline{\texttt{Load}}(a, mo, S); ss, T) \rightarrow (\Sigma'', ss, T)}$$

[ATOMIC STORE IMPL]

$$\frac{\begin{array}{c}S.t = T.t \qquad S.c = T.\mathbb{C}(T.t) \qquad S.v = \Sigma.NALocs(l) \\ S.sc = mo{=}{=}\texttt{seq\_cst} \qquad S.clock = A.\mathbb{L} \\ A = \Sigma.ALocs(a) \qquad A' = A[SE := A.SE.pushback(S)] \\ \Sigma' = \Sigma[ALocs := \Sigma.ALocs[a := A']]\end{array}}{(\Sigma, \underline{\texttt{Store}}(l, a, mo); ss, T) \rightarrow (\Sigma', ss, T)}$$

Figure 11: Semantics for atomic statements

```
ReadsFromSet(A, mo, T) {
    if A.SE = ∅ then error
    SS := {A.SE.back}
    S := A.SE.back
    FoundSC := S.sc
    do {
        if S.c ≤ T.ℂ(S.t) then return SS
        if ∃ld ∈ S.LD : ld.c ≤ T.ℂ(ld.t) then return SS
        if S.c ≤ T.$_F(S.t) then return SS
        if S.c ≤ T.$_W(S.t) ∧ S.sc then return SS
        if S.c ≤ T.$_R(S.t) ∧ mo = seq_cst then return SS
        if S = A.SE.front then error
        S := S.prev
        if ¬S.sc ∨ ¬FoundSC then SS := SS ∪ {S}
        FoundSC := FoundSC ∨ S.sc
    }
}
```

Figure 12: Construction of the reads-from set

SC FENCE HELPERS:

[SC ATOMIC LOAD]

$$\frac{mo = \texttt{seq\_cst} \qquad T' = T[\$_R := T.\$_R \cup \Sigma.\mathbb{S}_F]}{(\Sigma, T, mo) \rightarrow_{load} (\Sigma, T')}$$

[SC ATOMIC STORE]

$$\frac{mo = \texttt{seq\_cst} \qquad \Sigma' = \Sigma[\mathbb{S}_W := \Sigma.\mathbb{S}_W[T.t := T.\mathbb{C}(T.t)]]}{(\Sigma, T, mo) \rightarrow_{store} (\Sigma', T)}$$

[SC ATOMIC FENCE]

$$\frac{\begin{array}{c}mo = \texttt{seq\_cst} \qquad \Sigma' = \Sigma[\mathbb{S}_F := \Sigma.\mathbb{S}_F[T.t := T.\mathbb{C}(T.t)]] \\ T' = T[\$_F := T.\$_F \cup \Sigma'.\mathbb{S}_F] \\ T'' = T'[\$_W := T'.\$_W \cup \Sigma'.\mathbb{S}_W]\end{array}}{(\Sigma, T, mo) \rightarrow_{fence} (\Sigma', T'')}$$

[NON-SC ATOMIC]

$$\frac{mo \neq \texttt{seq\_cst} \qquad x \in \{load, store, fence\}}{(\Sigma, T, mo) \rightarrow_x (\Sigma, T)}$$

Figure 13: Semantics for sequentially consistent fence functions

produce for P. Formally, we wish to show the following:

$$\forall P \forall E(E \in rConsistent(P) \leftrightarrow \exists \sigma \in traces(P) \,.\, lift(\sigma) = E)$$

We sketch the argument here; for a more detailed argument, refer to Appendix C [27].

The forward case is shown by induction on construction of an execution E. Given a *partial execution* graph E_i, that is composed of events e_j for all $0 < j \leq i$, and trace σ_i where $lift(\sigma_i) = E_i$, when E_i is extended to E_{i+1} by adding event e_{i+1}, we can extend the trace σ_i to σ_{i+1} such that $lift(\sigma_{i+1}) = E_{i+1}$. The backward case is similar, we show that extending a partial trace for P that lifts to a partial execution of E, we will always end up with either the same partial execution or a new partial execution.

The order in which we add events to the partial execution must follow the commitment order described in §6.3. Therefore, we must first topologically sort the events of E.

7. Implementation and Experiments

We describe the implementation of our new techniques as tsan11, an extension to tsan (§7.1). We evaluate the effectiveness of tsan11 in practice, guided by the following research questions: **RQ1:** To what extent is tsan11 capable of finding known relaxed memory defects in moderate-sized benchmarks, and how does the tool compare with existing state-of-the-art in this regard? **RQ2:** What is the runtime and memory overhead associated with applying tsan11 to large applications, compared with native execution and application of the original tsan tool? **RQ3:** To what extent does tsan11 enable the detection of new, previously unknown errors in large applications, that could not be detected using tsan prior to our work?

In §7.2, we address RQ1 by applying tsan11, the original tsan tool and CDSChecker to a set of benchmarks that were used in a previous evaluation of CDSChecker [31]. In §7.3 we consider RQs 2 and 3 via analysis of the Firefox and Chromium web browsers.

Reproducibility To aid in reproducing our results, our tools, benchmarks and result log files are available online [26].

7.1 The tsan11 Tool

The goal of our work is to apply efficient, C++11-aware race detection to large programs. Therefore, we have implemented the enhanced VC algorithm of §3 and the instrumentation library described in §4 and formalised in §5 as an extension to the ThreadSanitizer (tsan) tool. The original tsan tool supports concurrent C++

programs and provides instrumentation for C++11 atomic operations, but, as illustrated in §2.3, does not handle these atomic operations properly. We refer to the original version of tsan as **tsan03** (because it does not fully cater for C++11 concurrency, and C++**03** is the version of C++ prior to C++11), and to our extension, that captures a large part of the C++**11** memory model, as **tsan11**.

The tsan tool is part of the `compiler-rt` LLVM project,[2] and our tsan11 extension is a patch to SVN revision 272792.

Bounding of store and load buffers To prevent unbounded memory overhead, we must bound the size of store buffers so that the oldest element of a full buffer is evicted when a new store element is pushed. This restricts the stores that loads can read from, so the buffer size trades memory overhead for observable behaviours. For our evaluation we used a buffer size of 128 to allow a relatively wide range of stores to be available to load operations. Load buffers need not be bounded. This is because at most one load element per thread is required for any store element: the oldest load has the smallest epoch, so if a later load blocks a thread, so will the oldest.

Resolving load operations at runtime Our instrumentation lets us control the reads-from relation via the the algorithm of Figure 12, allowing for variety of randomised and systematic strategies for weak behaviour exploration. Our implementation favours reading from older stores, choosing the oldest feasible store with 50% probability, the second-oldest with 25% probability, and so on.

7.2 Evaluating Using Benchmark Programs

Benchmark programs To compare tsan11 with tsan03 and CDSChecker at a fine-grained level, we applied the tools to the benchmarks used to evaluate CDSChecker previously [31]. These are small C11 programs ranging from 70 LOC to over 150 LOC. We had to convert the benchmarks to C++11 for use with tsan, due to the lack of a C11 threading library. Example benchmarks include data types and high level concurrency concepts, such as Linux read-write locks. There are 13 benchmarks, however some of these rely on causality cycles or load buffering to expose bugs and, as discussed in §6, tsan11 does not facilitate exploration of these sorts of weak behaviour. Of the 7 benchmarks whose behaviours tsan11 can handle, only 2 have data races. We therefore induced data races into the other 5 by making small mutations such as relaxing memory order parameters, reordering instructions and inserting additional non-atomic operations. The benchmarks, both before and after our race-inducing changes, are provided online at the URL associated with our experiments.

Notes on comparing tsan with CDSChecker Comparing tsan11 and CDSChecker is difficult as the tools differ in aim and approach. CDSChecker explores *all* behaviours of a program, guaranteeing to report all races; tsan11 explores only a single execution, determined by the OS scheduler and randomisation of the reads-from relation, reporting only those data races that the execution exposes. The goal of CDSChecker is exhaustive exploration of small-but-critical program fragments, while tsan11 is intended for analysis of large applications. CDSChecker requires manual annotation of the operations to be instrumented, and can only reason about C11 (not C++11) concurrency. This is a practical limitation because, at time of writing, C11 threads are not supported by mainstream compilers such as GCC and Clang.[3] In contrast, tsan11 automatically instruments all memory operations, and supports C++11 concurrency primitives. Nevertheless, we present a best effort comparison as CDSChecker is the most mature tool for analysis of C11 programs that we are aware of.

Experimental setup These experiments were run on an Intel i7-4770 8x3.40GHz with 16GB memory running Ubuntu 14.04 LTS. We added a sleep statement to the start of each thread in each benchmark in order to induce some variability in the schedules explored by the tsan tools. We used the Linux `time` command to record timings, taking the sum of user and system time. This does not incorporate the time associated with the added sleep statements, thus the wall-clock time associated with running the tsan tools is longer than what we report. We omit this time because, with further engineering, we could implement a strategy for inducing variability in the thread schedule with low overhead; the use of sleep is simply a proxy for this missing feature. The tsan-instrumented benchmarks were compiled using Clang v3.9. We used the revision of CDSChecker with hash 88fb552.[4]

The results of our experiment are summarised in Table 1, where all times are in ms, and discussed below. For each benchmark, we report the time taken for exploration using CDSChecker (deterministic tool), averaged over 10 runs, and the average time over 1000 runs for analysis using tsan11 (which is nondeterministic). For tsan11 we report the rate at which data races are detected, i.e. the percentage of runs that exposed races (**Race rate**), the number of runs required for a data race to be detected with at least 99.9% probability based on the race rate (**No. 99.9%**), and the associated time to conduct this number of runs, based on the average time per run (**Time 99.9%**). The **Runs to match** column shows the number of runs of tsan11 that could be performed in the same time as CDSChecker takes to execute (rounded up), and **Race chance** uses this number and the race rate to estimate the chances that tsan11 would find a race if executed for the same time that CDSChecker takes for exhaustive exploration. The table also shows the average time taken, over 1000 runs, to apply tsan03 on each benchmark and the associated race rate. We use the configuration of CDSChecker flags recommended in the CDSChecker documentation for all benchmarks. For tsan11, we use the default system scheduler and the store buffer bound and reads-from strategy discussed in §7.1.

Results The results show that tsan11 was able to find races in all but one of the benchmarks (barrier), but that the rate at which races are detected varies greatly, being particularly low for mpmc-queue. This is due to the dynamic nature of the tool: the thread schedule that is followed is dictated by the OS scheduler. For the remaining seven benchmarks, comparing the time taken to run CDSChecker with the "Time 99.9%" column for tsan11 shows that for 2 benchmarks, exhaustive exploration with CDSChecker is faster than reliable race analysis using tsan11, while for the other 5 benchmarks it is likely to be faster to use tsan11 to detect a race. Recall, though, that these times exclude the time associated with the sleep statements added to the benchmarks that tsan11 analyses, as discussed above. The "Race chance" column indicates that overall, with the exception of barrier, repeated application of tsan11 for the length of time that CDSChecker takes for exploration has a high probability of detecting a race. Note however that we measure the time for *full* exploration using CDSChecker; if CDSChecker were modified so as to exit on the first race encountered, the time it takes to find a race would likely be lower.

The race rate results for tsan03 show that in some cases the tool did not detect a race, either because the race depends on weak behaviour (meaning that tsan03 would be incapable of finding it) or is more likely to occur if non-SC executions are considered (for example, tsan03 does find a race in mcs-lock, but with a very low race rate). The timing results for tsan03 show that it is usually faster per

[2] `http://llvm.org/svn/llvm-project/compiler-rt/trunk`

[3] A recent Stack Overflow thread provides an overview of C11 threading support: `http://stackoverflow.com/questions/24557728/does-any-c-library-implement-c11-threads-for-gnu-linux`.

[4] `git://demsky.eecs.uci.edu/model-checker.git`

execution compared with tsan11. In general this is to be expected since tsan11 performs a heavier-weight analysis. However, these benchmarks are so short-running that small differences, such as the fact that tsan11 is slightly faster for analysis of chase-lev-deque, may be due to experimental error.

7.3 Evaluation Using Large Applications

Applications The programs we have focused on are Firefox and Chromium, two web browsers with very large code bases. Both browsers make heavy use of threads and atomics: Firefox can have upwards of 100 threads running concurrently, while Chromium starts multiple processes, each of which will will run many threads. As tsan03 had already been applied to both Firefox and Chromium, there were clear instructions on how to run both with tsan.

Experimental setup These experiments were run on an Intel Xeon E5-2640 v3 8x2.60GHz CPU with 32GB memory running Ubuntu 14.04 LTS, revision r298600 of Firefox[5] and the Chromium version tagged "tags/54.0.2840.71".[6] The browsers were compiled using Clang v3.9, following instructions for instrumenting each browser with tsan as provided by the developers of Firefox[7] and Chromium.[8] We run the browsers in a Docker container (using Docker v1.12.3, build 6b644ec) via ssh with X-forwarding.

We tested both browsers with tsan03 and tsan11, and without instrumentation. We use FF, FF03 and FF11 to refer to Firefox without instrumentation, and instrumented using tsan03 and tsan11, respectively; CR, CR03 and CR11 refer similarly to Chromium.

To make our evaluation as reproducible as possible, we tested the browsers using JSBench v2013.1 [40].[9] JSBench runs a series of JavaScript benchmarks, sampled from real-world applications, presenting runtime data averaged over 23 runs. We recorded peak memory usage via the Linux `time` command, reporting the "Maximum resident set size" data that this command records. For the browser versions instrumented with race analysis, we record all details of reported data races to a file. In the case of tsan11, we record, during analysis, data on the number and kinds of atomic operations, including their memory orders, that are issued during execution. The full JSBench reports for all browser configurations, together with memory usage information, data race reports and statistics on atomic operations, are available on our companion web page [26].

Results Table 2 shows results on memory usage, execution time and races reported running our browser configurations on JSBench. Recall that JSBench runs a series of benchmarks 23 times. The "Peak mem" column shows the maximum amount of memory (in MB) used throughout this process, as reported by the `time` tool. The "Mean time" column shows the mean time, averaged over the 23 runs, for running the benchmarks (data on standard deviation, and per-benchmark statistics reported by JSBench, are available from our web page). The "Races" column shows, for all configurations except FF and CR, the number of races reported during the entire JSBench run. The results for Firefox show that the increase in memory usage associated with FF03 vs. FF is 2.7×, compared with 9.6× for FF11 vs. FF. Thus, as expected, our instrumentation leads to significantly higher memory consumption. Performance-wise, our instrumentation leads to a more modest overhead: average JSBench runtime increases by 11.2× when using FF03 vs. FF, and by 14.2× when using FF11 vs. FF. Interestingly, the memory overhead associated with tsan03-based race instrumentation for Chromium is higher—a 10.6× increase with CR03 vs. CR—but grows less significantly when tsan11 is used—a 13.6× increase with CR11 vs. CR. The growth in runtime for Chromium follows a similar pattern to that for Firefox, with an increase in average runtime of 11.1× for CR03 vs. CR, and 17.1× for CR11 vs. CR.

Examination of the tsan logs showed 39 race reports for FF03 vs. 52 for FF11, and 1 for CR03 vs. 6 for CR11. We do not yet know whether the higher rate of races detected using tsan11 for both browsers is due to the additional behaviours that our instrumentation exposes, or simply a result of our instrumentation and its overheads causing a more varied set of thread interleavings to be explored. A tsan race report shows the stacks of the two threads involved in the race. It is hard to determine the root cause of the race from this, and harder still to understand whether the race depends on weak memory semantics; we leave a deeper investigation of this (requiring significant novel research) to future work.

When running FF11 and CR11 on JSBench, we recorded the number of each type of atomic operation that tsan11 intercepted. The full data is provided online, but we summarise the results in Table 3. The **atomic operations** row shows the total number of atomic operations that were issued during the entire JSBench run, indicating that both browsers, and especially Firefox, make significant use of C++11 atomic operations. We then show the percentage of operations associated with each operation type—load, store, RMW and fence. This indicates that fence operations were so scarce they contribute negligible percentage (12,203 and 78 fence operations were intercepted for Firefox and Chromium, respectively, and in all cases these were SC fences), that loads significantly outnumber stores (expected if busy-waiting is used), that relaxed operations are common, and that the other memory orderings are all used to a varying degree. Our results also confirmed that the *consume* ordering is not used. The heavier use of atomic operations by Firefox perhaps explains the larger growth in memory overhead associated with dynamic race instrumentation for this browser.

We do not yet have data on the distribution of executed atomic operations throughout the browser source code, nor the typical use cases for these operations, and believe that a detailed empirical study of atomic operation usage in these browsers, and in other large applications, is an important avenue for future work.

In summary: our experiments with the web browsers shows that (a) tsan11 is able to run at scale, with significant but not prohibitive memory and time overheads compared with tsan03, (b) tsan11 reports a larger number of races compared with tsan03, and (c) both web browsers make significant use of C++11 atomic operations. What our evaluation does not settle is the question of which aspects of our extensions to tsan to support C++11 concurrency are important in practice, for identifying new data races and suppressing possible false alarms reported by tsan03.

8. Related Work

There is a large body of work on data race analysis, largely split into dynamic analysis techniques (e.g. [13, 15, 21, 37, 38, 42]) and static approaches (e.g. [14, 33, 39, 45, 47]). Unlike our approach, none of these works handles C/C++11 concurrency.

Several recent approaches enable *exhaustive* exploration and race analysis of small C11 programs. CDSChecker [31, 32], which we study in §7.2, uses dynamic partial order reduction [17] to reduce state explosion. Cppmem [6], and an extended version of the Herd memory model simulator [3, 8], explore litmus tests written in restricted subsets of C11. Similarly, the Relacey tool supports thorough reasoning about the behaviours of concurrency unit tests, ac-

[5] `https://hg.mozilla.org/mozilla-central/`

[6] We obtained Chromium according to the instructions at `https://www.chromium.org/developers/how-tos/get-the-code/working-with-release-branches`.

[7] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Thread_Sanitizer`

[8] `https://www.chromium.org/developers/testing/threadsanitizer-tsan-v2`

[9] `http://plg.uwaterloo.ca/~dynjs/jsbench/`

	CDSChecker	tsan11						tsan03	
Test	**Time**	**Time**	**Race rate**	**No. 99.9%**	**Time 99.9%**	**Runs to match**	**Race chance**	**Time**	**Race rate**
barrier	5	18	0.0%	∞	∞	1	0.0%	16	0.0%
chase-lev-deque	90	7	18.3%	35	245	13	92.8%	8	94.5%
dekker-fences	4341	10	48.9%	11	110	434	>99.9%	9	100.0%
linuxrwlocks	11700	12	3.9%	174	2088	975	>99.9%	9	0.0%
mcs-lock	1206	24	19.8%	32	768	50	>99.9%	10	0.3%
mpmc-queue	11606	11	0.8%	861	9471	1055	>99.9%	9	0.0%
ms-queue	50	88	100.0%	1	88	1	100.0%	84	100.0%

Table 1: Comparison of CDSChecker, tsan11 and tsan03; all times reported are in ms

Browser	Peak mem (MB)	Mean time (ms)	Races (#)
FF	1,159	128	N/A
FF03	3,092	1431	39
FF11	11,092	1819	52
CR	109	103	N/A
CR03	1,158	1148	1
CR11	1,481	1765	6

Table 2: Memory usage, runtime and number of races reported for our browser configurations running on JSBench

Browser	Firefox	Chromium
# atomic operations	437M	280M
loads	55.33%	74.73%
stores	9.39%	7.76%
RMWs	35.28%	17.51%
fences	0.00%	0.00%
relaxed	38.97%	77.59%
acquire	14.28%	13.46%
release	1.98%	0.68%
acq/rel	4.83%	1.64%
SC	39.94%	6.63%

Table 3: The number of atomic operations executed by the browsers during a complete JSBench run, with a breakdown according to operation type and memory order

counting for C++11 memory model semantics [49]. Our approach is different and complementary: we do not aim for full coverage, but instead for efficient race analysis scaling to large applications.

Formulating an operational semantics for C/C++11 has been the subject of recent work [12, 23, 24, 30, 34, 35]. A key work here presents an executable operational semantics for the memory model [30], and we based our notion of *commitment order* on this work. The main difference between our contribution and that of [30] is that the approach of [30] provides complete coverage of the memory model: the operational semantics is provably equivalent to the axiomatic model of [6]. This is achieved by having the operational semantics track a prefix of a consistent candidate execution throughout an execution trace. These prefixes can grow very large and become expensive to manipulate, and it seems unlikely that the approach would be feasible for instrumentation of large-scale applications such as the web browsers that we study. In contrast, our semantics covers only a subset of the memory model, but can be efficiently explored during scalable dynamic analysis.

A program transformation that simulates weak memory model behaviours is the basis of a technique for applying program analyses that assume SC to programs that are expected to exhibit relaxed behaviours [2]. Like our instrumentation, the method works by introducing buffers on per memory location basis in a manner that allows non-SC memory accesses to be simulated. The key distinction between this work and ours is that we account for C++11 atomic operations with a range of memory orderings, whereas the method of [2] only applies to racy programs without atomic operations, applying a single consistency model to all memory accesses.

A limitation of our approach is that our instrumentation does not take account of program transformations that might be applied due to compiler optimisations. The interaction between C/C++11 concurrency and compiler optimisations has been the subject of several recent works [11, 29, 36, 46], as has the correctness of compilation schemes from C11/C++11 to various architectures [6, 7, 41, 48]. Future work could consider exploring the effects of program-level transformations during dynamic analysis.

Randomising the reads-from relation during uncontrolled dynamic analysis has been applied in other works [10, 16]. An alternative would be to explore this relation systematically, similar to a recent approach for testing concurrent programs under the TSO memory model [50], and a method for memory model-aware model checking of concurrent Java programs [22].

The KernelThreadSanitizer (ktsan) tool provides support for fence operations, which are prevalent in the Linux kernel [18], and source code comments indicate that an older version of tsan provided some support for non-SC executions.[10]

9. Conclusion

We have presented a method for accurate dynamic race analysis for C++11 programs, and an instrumentation library that allows a large fragment of the C++11 relaxed memory model to be explored. Our experiments show that our implementation, an extension to tsan, can detect races that are beyond the scope of the original tool, and that our extended instrumentation still enables analysis of large applications—the Firefox and Chromium web browsers. Avenues for future work include: developing more advanced heuristics for exploring captured weak behaviours; devising further instrumentation techniques to capture a larger fragment of the memory model; conducting a larger-scale experimental study of data race defects in C++11 software, to understand the extent to which weak memory-related bugs, vs. bugs that can already manifest under SC semantics, are a problem in practice; and designing extensions our technique to cater for the OpenCL memory model [8], facilitating weak-memory aware data race detection for software running on GPU architectures, which are known to have weak memory models [4] that can lead to subtle defects in practical applications [44].

Acknowledgements

Special thanks to Paul Thomson and Hugues Evrard for assistance with our final experimental setup and evaluation. Thanks to Dmitry Vyukov, Anton Podkopaev, Tyler Sorensen, John Wickerson, and the anonymous reviewers and artifact evaluators, for their feedback on our work. This work was supported by a PhD studentship from GCHQ, and by EPSRC Early Career Fellowship EP/N026314/1.

[10] `https://github.com/Ramki-Ravindran/data-race-test/commit/d71e69e976fe754e40cac13145ab31e593a2edd1`

References

[1] S. Adve. Data races are evil with no exceptions: Technical perspective. *Commun. ACM*, 53:84–84, 2010.

[2] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, pages 512–532, 2013.

[3] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[4] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, pages 577–591, 2015.

[5] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency: The post-Rapperswil model. Technical Report N3132=10-0122, JTC1/SC22/WG21 – The C++ Standards Committee, 2010.

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.

[7] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++ 11 to POWER. In *POPL*, pages 509–520, 2012.

[8] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648, 2016.

[9] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *PPDP*, pages 113–124, 2011.

[10] M. Cao, J. Roemer, A. Sengupta, and M. D. Bond. Prescient memory: exposing weak memory model behavior by looking into the future. In *ISMM*, pages 99–110, 2016.

[11] S. Chakraborty and V. Vafeiadis. Validating optimizations of concurrent C/C++ programs. In *CGO*, pages 216–226, 2016.

[12] M. Doko and V. Vafeiadis. A program logic for C11 memory fences. In *VMCAI*, pages 413–430, 2016.

[13] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race-aware Java runtime. *Commun. ACM*, 53(11):85–92, 2010.

[14] D. R. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[15] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[16] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *PLDI*, pages 244–254, 2010.

[17] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[18] Google. KernelThreadSanitizer, a fast data race detector for the Linux kernel, visited November 2016. `https://github.com/google/ktsan`.

[19] ISO/IEC. Programming languages – C. International standard 9899:2011, 2011.

[20] ISO/IEC. Programming languages – C++. International standard 14882:2011, 2011.

[21] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward integration of data race detection in DSM systems. *J. Parallel Distrib. Comput.*, 59(2):180–203, 1999.

[22] H. Jin, T. Yavuz-Kahveci, and B. A. Sanders. Java memory model-aware model checking. In *TACAS*, pages 220–236, 2012.

[23] R. Krebbers and F. Wiedijk. A typed C11 semantics for interactive theorem proving. In *CPP*, pages 15–27, 2015.

[24] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *POPL*, pages 649–662, 2016.

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

[26] C. Lidbury and A. F. Donaldson. Companion webiste for reproducibility of experiments, 2017. `http://multicore.doc.ic.ac.uk/projects/tsan11/`.

[27] C. Lidbury and A. F. Donaldson. Dynamic race detection for C++11: Extended version, 2017. `https://www.doc.ic.ac.uk/~afd/homepages/papers/pdfs/2017/POPLExtended.pdf`.

[28] F. Mattern. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[29] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, pages 187–196, 2013.

[30] K. Nienhuis, K. Memarian, and P. Sewell. An operational semantics for C/C++11 concurrency. In *OOPSLA*, pages 111–128, 2016.

[31] B. Norris and B. Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA*, pages 131–150, 2013.

[32] B. Norris and B. Demsky. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.*, 38(3):10, 2016.

[33] Oracle Corporation. Analyzing program performance with Sun WorkShop, Chapter 5: Lock analysis tool. `http://docs.oracle.com/cd/E19059-01/wrkshp50/805-4947/6j4m8jrnd/index.html`, 2010.

[34] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL*, pages 622–633, 2016.

[35] A. Podkopaev, I. Sergey, and A. Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016.

[36] D. Poetzl and D. Kroening. Formalizing and checking thread refinement for data-race-free execution models. In *TACAS*, pages 515–530, 2016.

[37] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multihreaded C++ programs. In *PPoPP*, pages 179–190, 2003.

[38] E. Pozniansky and A. Schuster. Multirace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[39] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *PLDI*, pages 320–331, 2006.

[40] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *OOPSLA*, pages 677–694, 2011.

[41] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI*, pages 311–322, 2012.

[42] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[43] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *WBIA*, pages 62–71, 2009.

[44] T. Sorensen and A. F. Donaldson. Exposing errors related to weak memory in GPU applications. In *PLDI*, pages 100–113, 2016.

[45] N. Sterling. WARLOCK - A static data race analysis tool. In *USENIX Winter*, pages 97–106, 1993.

[46] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220, 2015.

[47] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *FSE*, pages 205–214, 2007.

[48] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[49] D. Vyukov. Relacy race detector, visited November 2016. `http://www.1024cores.net/home/relacy-race-detector`.

[50] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *PLDI*, pages 250–259, 2015.