Dynamic Race Detection For C++11

March 27, 2025

1 Introduction

- The definition of a data race in C++11 is far from trivial, due to the complex rules for when synchronisation occurs between various atomic operations provided by the language
- Another subtlety of this new memory model is the reads-from relation, which specifies the values that can be observed by an atomic load.

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. Maybe this is true in 2017, but nowadays it understands, maybe not fully.

```
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)

2 Background

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 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.

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, 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.

2.1.1 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, $\mathbf{a} : \mathbf{W_{rel}}\ \mathbf{x} = 1$ is a memory event that corresponds to a relaxed write of 1 to memory location **x**; **a** is a unique identifier for the event. The event types **W**, **R**, **RMW** and **F** represent read, write, RMW and fence events, respectively. Memory orderings are shortened to **rlx**, **rel**, **acq**, **ra**, **sc** and **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, $\mathbf{b} : \mathbf{RMW_{ra}}\ \mathbf{x} = 1/2$ shows event **b** reading value 1 from and writing value 2 to **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.

2.1.2 Presentations of Execution Graphs

Throughout the paper we present a number of execution graphs, such as those depicted in Figures 2.1.4 and 2.1.6. 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 2.1.4 we label events starting with c, not showing events a and b that give initial values to locations **x** and **nax**.

2.1.3 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.

2.1.4 Derived Relations

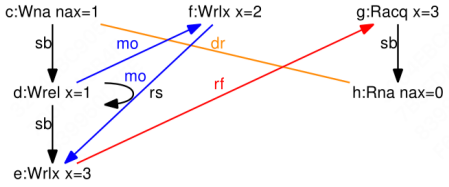


Figure 2: The release sequence headed by d is blocked by event f, causing a data race between c, the non-atomic write to **nax**, and h, the non-atomic read from **nax**; if the blocking event f is removed, there is no race

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 2.1.4 shows a release sequence that is immediately blocked by a relaxed write from another thread. I DON'T UNDERSTAND

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

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. An asw edge is also sw an edge.

Happens-before (**hb**) is simply $(sb \cup sw)^+$ (where + denotes transitive closure), representing Lamport's partial ordering over the events in a system. 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.

2.1.5 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 2.1.4 shows an execution with a data race, as there is no

sw edge between the release store d and acquire load \9g, 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

2.1.6 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**. 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. 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. equivalent to partial 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 $consistent_{ithb}$ axiom, without consume, simply requires hb to be irreflexive.

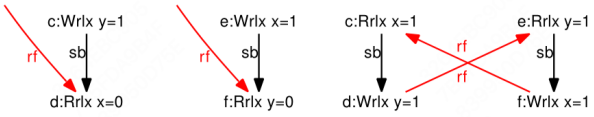
The last three axioms, $consistent_{scorder}$, $consistent_{mo}$ and $consistent_{rf}$ *_mapping*, correspond with the formation of the sc, mo and rf relations. We cover these in detail when presenting our instrumentation library.

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 2.1.6 shows two such executions that arise from well-known litmus tests.

- 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 load buffering example, one of the reads will read from a write that has not even been performed yet.



(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

2.2 Dynamic Race Detection

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, as defined in:

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

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

Example 2.1. Consider example 1a.

Initially, the thread VCs are $\mathbb{C}_{T1} = (1,0,0)$, $\mathbb{C}_{T2} = (0,1,0)$, $\mathbb{C}_{T3} = (0,0,1)$, and we have $\mathbb{R}_{nax} = \mathbb{L}_x = \bot_V$.

Statement `A` writes to `nax`, which has not been accessed previously, no race check is required. After `A`, $W_{nax} = 1@T1$, bacause T1's epoch is 1. After T1's release store at `B`, $\mathbb{L}_x := \mathbb{L}_x \cup \mathbb{C}_{T1} = (1,0,0)$ and $\mathbb{C}_{T_1} = inc_{T1}(\mathbb{C}_{T1}) = (2,0,0)$. After `T2`'s acquire load `C`, $\mathbb{C}_{T2} = \mathbb{C}_{T2} \cup \mathbb{L}_x = (1,1,0)$. 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`, $\mathbb{C}_{T3} := \mathbb{C}_{T3} \cup \mathbb{L}_x = (1,0,1)$. 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 $c \leq \mathbb{C}_{T3}(t)$, where $W_{nax} = c@t$. Because $W_{nax} = 1@T1$, this simplifies to $1 \leq \mathbb{C}_{T3}(T1)$, which can be seen to hold. The execution is thus deemed race-free.

Later, 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

Limitations of tsan. 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`.

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.

3.1 Release Sequences

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.

3.1.1 Blocking Release Sequences

Recall the execution of Figure 2.1.4. 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 3.1. Recall our 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 $\mathbb{L}_x$ to be set to $\bot_V$. As a result, the acquire load by T3 at `E` yields $\mathbb{C}_{T3} = \mathbb{C}_{T3} \cup \mathbb{L}_x = (0, 0, 1)$. This causes the write-read race check on nax to fail at `F`.. Thus a race is detected, as required by the C++11 memory model.

3.1.2 Read-Modify-Writes

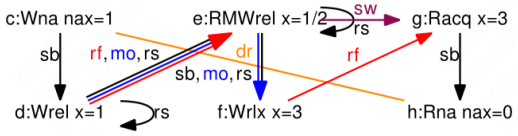


Figure 4: The release sequence started by d and continued by e is blocked by f; thus d does not synchronise with g, so c races with h

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 3.1.2, release sequences from the left and middle threads 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.

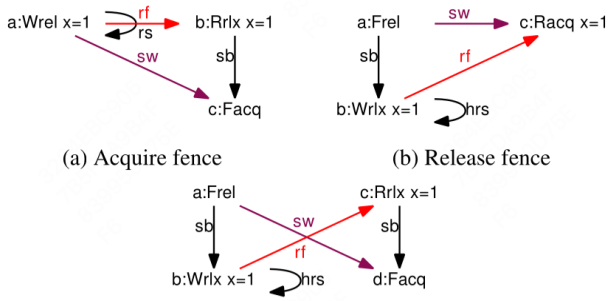
How $\mathbb{V}_m$ is updated depends on the type of operation being performed. If thread t performs a non-RMW store to m, $\mathbb{V}_m(u)$ is set to $\bot_V$ for each thread $u \neq t$. If the store has release ordering, $\mathbb{V}_m(t)$ and $\mathbb{L}_m$ are set to $\mathbb{C}_t$;

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 Lm nor Vm and all release sequences continue as before. If the RMW has release ordering, Vm (t) is updated to Ct , and the VC for t is joined on to the VC for m, i.e. Lm := Lm ∪Ct . By updating Lm 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

Fences are not handled in tsan: programs such as that of Figure 1c will not be properly instrumented, leading to false positives.



(a) Acquire fence

(b) Release fence

(c) Acquire and release fences

Figure 5: Synchronisation caused by fences

- 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 3.2a.
- Release fences use the hypothetical release sequence. 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 3.2b.

- Release fences and acquire fences can also synchronise with each other, shown in Figure 3.2c.

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 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, $\mathbb{L}_m$ is joined on to $\mathbb{F}_t^{acq}$

To illustrate fence synchronisation, consider the four operations shown in the execution fragment in Figure 3.2c. 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, $\mathbb{F}_u^{acq} = \mathbb{F}_u^{acq} \cup \mathbb{L}_x$. Finally, after d, we have $\mathbb{C}'_u = \mathbb{C}_u \cup \mathbb{F}_u^{acq} \geq \mathbb{C}_u \cup \mathbb{F}_t^{rel} = \mathbb{C}_u \cup \mathbb{C}_t$. Thus we have synchronisation between a and d.

4 Problems

5 References