Imperial College London
Department of Computing

Dynamic Analysis for Concurrent Modern C/C++ Applications

Christopher David Lidbury

May 2019

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

This thesis and the work it presents are my own except where otherwise acknowledged.

Christopher David Lidbury

Abstract

Concurrent programs are executed by multiple threads that run simultaneously. While this allows programs to run more efficiently by utilising multiple processors, it brings with it numerous complications. For example, a program may behave unpredictably or erroneously when multiple threads modify the same memory location in an uncoordinated manner. Issues such as this are difficult to avoid, and when introduced, can break the program in unpredictable ways. Programmers will therefore often turn towards automated tools to aide in the detection of concurrency bugs.

The work presented in this thesis aims to provide methods to aid in the creation of tools for the purpose of finding and explaining concurrency bugs. In particular, the following studies have been conducted:

Dynamic Race Detection for C/C++11 With the introduction of a weak memory model in C++, many tools that provide dynamic race detection have become outdated, and are unable to adequately identify data races. This work updates an existing data race detection algorithm such that it can identify data races according to this new definition. A method for allowing programs to explore many of the weak behaviours that this new memory model permits is also provided.

Record and Replay Much work has gone into record and replay, however, most of this work is focussed on *whole system* replay, whereby a tool will aim to record as much of the program execution as possible. Contrasting this, the work presented here aims to record *as little as* possible. This *sparse* approach has many interesting implications: some programs that were previously out of reach for record and reply become tractable, and vice versa. To back this up, controlled scheduling is introduced that is capable of applying different scheduling strategies, which combined with the record and replay is beneficial for helping to root out bugs.

Tool Support Both of the above techniques have been implemented in a tool, tsan11rec, that builds on the tsan dynamic race detection tool. A large experimental evaluation is presented investigating the effectiveness of the enhanced data race detection algorithm when applied to the Firefox and Chromium web browsers, and of the novel approach to record and replay when applied to a diverse set of concurrent applications.

Contents

List of Tables

List of Figures

1 Introduction

Linux introduced the notion of threads in C with *pthreads* [NBF96] 1996. But it wasn't until fifteen years later in 2011 that C formally adopted threads with C11 [ISO11b]. Most modern languages provide some form of support for multi-threading, including Java [GPB+06] and Go [go]. Hardware has also become very accommodating of threads, with many multi-core processors providing low level caches shared between threads, through which data can be shared, and hardware barriers to communicate. Multi-threading has therefore become commonplace, with many programs utilising it in some form, even indirectly through the use of a library.

The introduction of multi-threading has provided many benefits to programmers with regards to program speed, structure and many others aspects; however, it also introduces a number of pitfalls, which ultimately makes multi-threaded programming very difficult to do correctly. For a single-threaded program, program bugs will depend on the program code and how the program interacts with the environment, such as the file system or network. With multiple threads, the order in which they execute becomes a major source of nondeterminism, and so one must also consider how they communicate and modify parts of the program state. If two threads modify, for example, the same data structure, it may end up in an undefined state. Attempting to debug such programs is also more difficult. Running the same program twice and ensuring the relevant program environment and inputs are the same on both runs will not ensure that the program will follow the same sequence of states. In these cases, traditional debugging methods such as pausing a program with a debugger will not be enough, as there will be no clear indication as to how the program became in such a state.

Perhaps the most defining type of bug that arises with multi-threading is the *data race*, typically defined as follows: A data race occurs between two threads when both access the same location, at least one is a write, at least one is *non-atomic*, and neither happens before the other [Lam78]. But without any formalisation, what are "atomic accesses" and "happens-before"? Before C11, happens-before was assumed to occur primarily between the lock and unlock operations on mutexes, between a parent and child thread upon thread creation and joining, and between operations in the same thread according to program

order. It was also assumed that lock operations were atomic. From this grew the lock-based approach to multi-threading, where everything that could be accessed by multiple threads was guarded by some form of lock. Atomics were introduced with C11 that can be accessed by multiple threads without the need for locks, but are also more difficult to use, as there is no longer a region of code that is mutually exclusive.

To help fix these issues, programmers may use a *program analyser*. An analyser will capture and infer properties of the program in an attempt to inform the user of particular deficiencies in the program. There are many different kinds of analysers, that focus on different deficiencies and use different methods of approach. For example, the Address-Sanitizer [HH12] tool will analyse a program, while it is running, and inform the user of any address violations detected. This of course means that it will only detect address violations, and only if they occur during that particular execution.

1.1 Current Issues

The current state of concurrent programming has become unclear. While program analysers, libraries, research and understanding on the pitfalls of concurrent programming have come a long way, it has become muddied by the introduction of complicated memory models and the plethora of different models that different hardware and languages specify.

In the particular case of program analysers, while research continues to be performed on improving the efficiency and their ability to detect data races according to some arbitrary and simple definition of a data race, none of this research addresses the issue of the increased complexity of the memory models embedded in the languages, or helping the user to understand what is wrong with their program.

1.2 Contributions

In light of the issues raised above, this work focuses on three important issues:

Dynamic race detection for C11/C++11 (§3) focuses on the detection of data races in the presence of a weak memory model, specifically, C11/C++11. It does so by updating the traditional vector clock algorithm [FF09] to be aware of the nuances of the C11/C++11 memory model. It also allows the exploration of some of the weak behaviours permitted by C11/C++11, something that was previously restricted to static analysers due to the perceived difficulties.

Controlled scheduling with sparse record and replay (§4) attempts to solve the issue of both finding and reproducing difficult to find data races. It does so by introducing a method of controlled scheduling that works in tandem with a record and replay system. This takes a sparse approach to record and replay, where instead of recording as much as possible, it tries to record *as little as* possible.

ThreadSanitizer §5 details the implementation of the two previous contributions as an extension of ThreadSanitizer (tsan), a state of the art dynamic race detection tool. An extensive evaluation of the tool is provided, showing that it can scale to large applications and can both find and reproduce bugs. These applications range from small litmus tests, to large programs such as popular web browsers, and real-time applications such as videogames.

1.3 Publications

The material presented in this thesis has been published as follows:

Dynamic race detection for C++11 The dynamic race detection work of §3, and its implementation and evaluation in §5.2, have been published in the 2017 ACM SIGPLAN conference on Principles of Programming Language (POPL 2017) [LD17b]. This work has an approved artifact and a companion website which can be found online [LD17a].

Sparse Record and Replay With Controlled Scheduling The controlled scheduling and record and replay work of §4, and its implementation and evaluation in §5.3, have been accepted and is to appear in the 2019 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2019). A reference to a draft is provided [LD].

1.4 Acknowledgements

The author acknowledges and is grateful to the collaborators throughout this PhD. In particular Alastair Donaldson, who co-authored and contributed to the prose in all of the work presented in this thesis. Additional acknowledgements are given to John Wickerson and Mark Batty for help on understanding the technical aspects of the C++11 memory model, and on the presentation of the paper "Dynamic race detection for C++11" [LD17b].

2 Background

2.1 Multi-Threaded Programs

Multi-threaded programming allows programs to be executed with multiple threads of execution. In C++11, threads are created explicitly with the `std::thread` class, by passing in a function pointer. This function becomes the `main` function for the thread. An example program is shown in Figure 2.1, whereby the main thread constructs two threads `t1` and `t2`, passing in the `Summation` function as the entry point and a pointer to the data that needs summing. The `join` operation will cause the thread invoking it to block until the specified child thread has finished. Each thread may run on two separate processors concurrently, in theory doubling the speed over using a single thread. In practice, the overhead of thread creation and completion will result in diminishing returns for fixed-size data with the increase in thread count [Rod85].

Another example program is shown in Figure 2.2. The audio subsystem has been given its own dedicated thread, tasked with consolidating all sound samples being played, mixing them together and then sending the mixed sample to the audio device. Other threads do not need to be concerned with playing back the audio with the correct timing, simplifying them. The responsiveness of the audio will be improved, as the audio thread cannot be busy working on other parts of the program.

In both examples, certain steps are taken to ensure correctness in how the threads interact. In the audio example, access to `samples_pending` is guarded by a mutex. Mutexes ensure mutually exclusive access to parts of the process memory space. In this case, if a thread is in `PlaySample` and has acquired the mutex, the audio thread must wait until the mutex is freed before it can acquire it and move past the mutex acquire. Mutual exclusion is a common tool used in concurrent programs, but has the drawback that only one thread may be running inside a mutex-protected region at a time.

The summation example uses an atomic variable instead. Under most circumstances, variables are *not* safe to access concurrently. Atomics, however, are safe to use without the guarantee of mutual exclusion, so in this case, there is no issue with `total`. The benefit of this is that there is no need for mutual exclusion, and therefore threads are left to run

```
const size_t kDataCount = 1024;
int data[kDataCount];
std::atomic<int> total(0);

void Summation(const int *data, size_t count) {
  int total = 0;
  for (size_t idx = 0; idx < count; ++idx) {
    total += data[idx];
  }
  ::total += total;
}

int main(int argc, char **arv) {
  FillData();
  std::thread t1(Summation, &data[0 * kDataCount / 2], kDataCount / 2);
  std::thread t2(Summation, &data[1 * kDataCount / 2], kDataCount / 2);
  t1.join();
  t2.join();
  std::cout << "Total: " << total << std::endl;
  return 0;
}
```

Figure 2.1: Multiple threads perform a summation over disjoint sets of data concurrently, providing a linear speed-up over a single-threaded approach.

```
std::list<const SoundSample&> samples_pending;
std::list<const SoundSample&> samples_active;
std::mutex audio_mtx;

void RunAudio() {
  InitAudio();
  while (!shutdown) {
    SoundSample mixed_sample = MixSamples(samples_active);
    SendAudio(mixed_sample);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    std::unique_lock<std::mutex> lck(audio_mtx);
    samples_active.splice(samples_active.end(), samples_pending);
  }
}

void PlaySample(const SoundSample& sample) {
  std::unique_lock<std::mutex> lck(audio_mtx);
  samples_pending.push_back(sample);
}

int main(int argc, char **arv) {
  std::thread audio_thread(RunAudio);
  ...
  if (weapon_fired) {
    PlaySample(weapon_sample);
  }
  ...
  audio_thread.join();
  return 0;
}
```

Figure 2.2: The audio playback has a dedicated thread. The main thread does not need to think about it, and the audio will be responsive.

in parallel.

In these examples, the programmer has taken the necessary precautions to ensure that there are no adverse effects caused by the multi-threaded approach. Sometimes these precautions are not enough, and the program can fail in strange ways. Consider the `Summation` function in Figure 2.1, specifically the line `::total += total;`. For atomic locations, the `+=` is an atomic *read-modify-write*, it performs the load, addition and store as a single, irreducible operation. If this line is instead `::total = ::total + total;`, the load, addition and store will be three separate operations. Two threads could then load the original value of `total` in parallel, without seeing the updated value from the other thread. The resulting value would then be incorrect.

Now consider the audio example of Figure 2.2, and imagine someone has added functionality for stopping samples that are currently being played, that looks as follows:

```
void StopSample(const SoundSample& sample) {
  samples_active.remove(sample);
}
```

Said person has clearly forgotten to acquire the mutex first. The effects of this error is not clear, for example, the list may become corrupt and the program may crash as a result, the function may fail to stop the sample from playing, or, most likely, the program will appear to continue as expected. One of the major problems with concurrency bugs is their tendency to appear rarely and unpredictably, and so they are often colloquially known as *heisenbugs* [MQB+08].

Concurrent programming has undergone much work, with many programming constructs being created specifically for use in multi-threaded programs, such as condition variables, monitors, and even data structures specifically designed to be safely accessed by multiple threads. But even with these constructs, the extra complications that come with inter-thread interactions will still result in subtle bugs [ND13, BAM07].

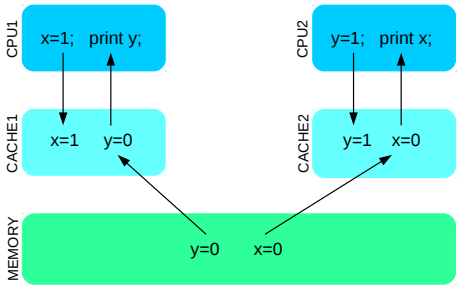
2.2 Memory Models

The examples shown in Figures 2.1 and 2.2 provide two distinct methods of multi-threaded interaction: the well established method of using mutual exclusion and the more modern approach of using atomics. While the atomic approach avoids the need to block threads, it has many nuances that must be formalised through the use of a *memory model*. To see why this is necessary, a closer look at the hardware is required.

Consider the simple abstract program fragment shown in Figure 2.3a. Each thread writes to one of `x` and `y` before reading the other in parallel. Assume that all accesses

```
x = 1;          ||    y = 1;
print y;        ||    print x;
```

(a) Two threads write to and read from `x` and `y`. The vertical bars represent separate threads running in parallel.



(b) The CPUs do not communicate directly with memory, and may read older values from the cache or leave stores in the cache.

Figure 2.3: Simple parallel program, showing one of the possible ways in which it can run on hardware.

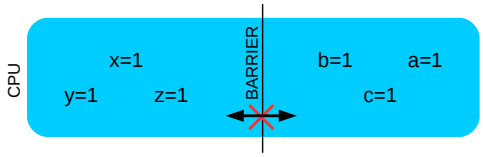
are atomic, and therefore safe. A common assumption will be that there is a sequentially consistent ordering over all of the operations. For example, one execution may be `x=1; y=1; print x; print y;`, and so the program will output `1 1`. Under sequential consistency, the program should never output `0 0`. These assumptions do not hold in reality though, and it is possible for the program to output `0 0`, depending on the hardware and compiler. One such way is shown in Figure 2.3b, whereby each store is kept in the CPU's cache causing each load to see the initial value in memory. Given the typical hardware setup of Figure 2.3b, enforcing sequential consistency over all threads will in fact be more tricky and expensive than going without. Other ways in which this can happen include the compiler reordering operations, or the CPU executing operations out-of-order, a common feature in modern CPUs.

Under a single processor, how memory operations interact is mostly unambiguous. With multiple processors, the state of the system can become unspecified, particularly if the memory operations are non-atomic. The role of a hardware memory model is to specify the behaviour of memory operations, such as how they interact and how they are ordered.

From Figure 2.3, it is clear that without some way to order memory operations, the values a load can read will be mostly unconstrained, which is particularly problematic when considering the locking example of Figure 2.2. One common solution is to use *memory barriers* (also known as memory fences), to prevent the memory operations from being reordered beyond certain points. An example of how memory barriers prevent reorderings is shown in Figure 2.4. This barrier will not prevent the operations on either side of the barrier from being reordered amongst themselves however. Different types of hardware

```
x=1; y=1; z=1;
__sync_synchronize();
a=1; b=1; c=1;
```

(a) Memory stores separated by a GCC builtin barrier, which will compile into a hardware barrier.

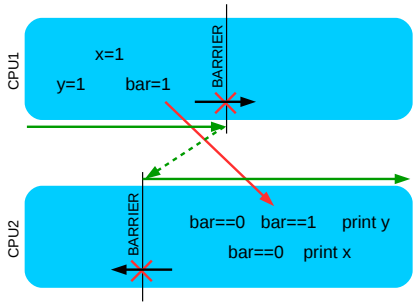


(b) Barrier prevents memory operations from shifting across it, but does not prevent other reorderings.

Figure 2.4: The diagram on the right shows how a barrier prevents memory operations from shifting over it.

```
x = 1;          ||  acquire();
y = 1;          ||  while(bar != 1);
                ||
bar = 1;        ||  print x;
release();      ||  print y;
```

(a) Synchronisation between release and acquire barriers.



(b) The green arrow represents *visibility*, or *happens before*. It crosses the thread boundary thanks to barrier synchronisation.

Figure 2.5: Barrier synchronisation. The operations before the barrier on `CPU1` become *visible* to those after the barrier on `CPU2`.

will have their own variations of barriers, which will be defined by the underlying memory model.

For interthread ordering, which is crucial for the locking example in Figure 2.2 to work, barriers must connect across threads. Most hardware will provide two kinds of barriers: a release barrier, which will prevent reordering past it, and an acquire barrier, which will prevent reordering before it. A barrier can also be both, such as the one shown in Figure 2.4b. Release barriers *synchronise with* acquire barriers, causing everything ordered before the release barrier to become visible to everything ordered after the acquire barrier. For this to work, the barrier is usually coupled with a memory access, and requires one memory access to read from another. An example is shown in Figure 2.5. When the load on `bar` reads from the store by `CPU1`, represented by the red arrow, the barriers synchronise, and so `CPU2` can safely access `x` and `y`.

While most hardware memory models share the same basic principles, trying to cater for all of them in a high level programming language such as C++ or Java would be difficult. These languages that provide support for concurrency will therefore also provide their own software memory model, to bridge the gap between the program and the hardware, and to provide a single unified memory model which the programmer can program against. It is the responsibility of the compiler, that implements a language specification, to ensure that whatever code gets generated to run on hardware respects the underlying software memory model of the language.

2.3 Program Analysis

With the complexity of programs increasing over time, so too does the complexity of the bugs they exhibit. Many programs will have bugs that are too contrived to detect manually and too difficult to fix by hand. The process of finding and explaining bugs autonomously through the use of program analysis can therefore be beneficial.

Broadly speaking, program analysers come in two forms, static and dynamic, as depicted in Table 2.1. A static analyser will work on the source program, while a dynamic analyser runs alongside the program, collecting information while it is running. Both kinds of analysis have a precision-scalability trade-off. Some cheap static analysers will scale really well, but only check simple properties (e.g. C Lint [Dar96]) or have a high false positive rate (e.g. FindBugs [HP07]), while others are much more detailed but do not scale (e.g. CDSChecker [ND13]). Dynamic analysers typically incur a linear increase in execution time, but the slowdown varies depending on the nature of the analysis. Dynamic analysis can almost never be used to prove absence of bugs, as they will typically only explore single executions, but bug reports will usually be accurate. Static analysers can be complete, but can also produce bug reports that are imprecise or are not even an issue, or if they are, it might not be clear how to reproduce them.

To demonstrate how these analyses work in practise, and the practical differences between the two, consider a static and dynamic analyser that can detect invalid memory accesses, that will be applied to the program of Figure 2.6. This program reads in a vector of numbers, sorts them, and then prints the nth number, specified by the first user argument. Assume that the `parse_numbers` function will correctly parse each number and exit the program if there is an invalid number provided.

An example run of a static analyser is shown in Figure 2.7. This particular analyser aims to construct a knowledge base of the program in question, using it to identify semantic errors such as memory access violations. The knowledge base of the program starts off

Static analysis	Dynamic analysis
Analyses the codebase, without execution.	Collects and analyses as the program is executing.
Collects information on all possible executions.	Only collects information on a single execution.
Generally works on small programs (<200 LOC), but can scale very well (e.g. for type checkers, linters).	Scales up to very large programs.
Runs as its own executable, applied to the source or executable for the codebase of interest	Requires instrumentation of the original program, with a static library bundled with the executable.

Table 2.1: Differences between static and dynamic analysis.

```c
int main(int argc, char **argv) {
  if (argc < 3) {
    return 1;
  }
  size_t count = argc - 2;
  long pos = strtol(argv[1], NULL, 10);
  long *nums = malloc(sizeof(long) * count);

  parse_numbers(&argv[2], count, nums);
  qsort(nums, count, sizeof(long), compare);
  if (pos < 0 || pos > count) {
    return 1;
  }
  printf("%ld\n", nums[pos]);

  return 0;
}
```

Figure 2.6: Finding the nth number in an unordered sequence of numbers.

empty, and is gradually added to upon processing each line. When the print statement is reached, it deduces that the range of potential indexes into `nums` can be invalid, as the program permits the index to equal `count`. This knowledge base can become complex when loops and functions are also included.

An example of a dynamic analysis is shown in Figure 2.8. Here the program is instrumented by calling into a library at certain points in the program execution, which keeps track of certain parts of the program's state. These library calls can be added to the relevant lines in the program by the compiler automatically, removing the need for the user to modify the program manually. In this case, only events that occur during a particular execution can be analysed.

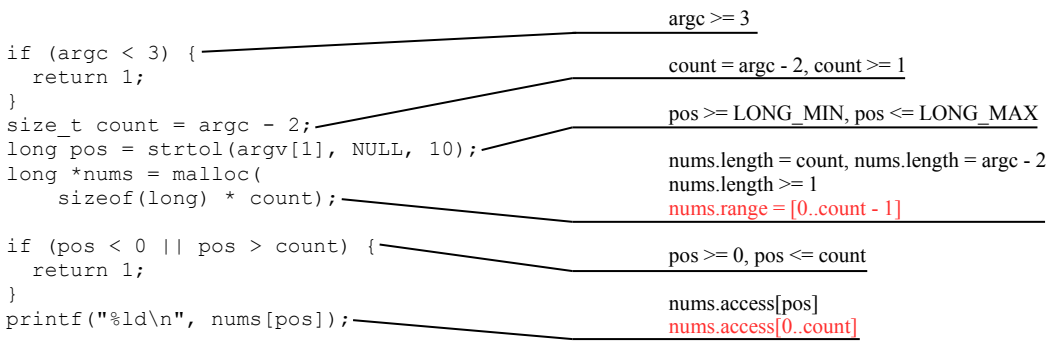


Figure 2.7: Knowledge about the program is gradually built up while processing the lines of the program. When `nums` is accessed, the possible range of indexes is out of the range of the array.

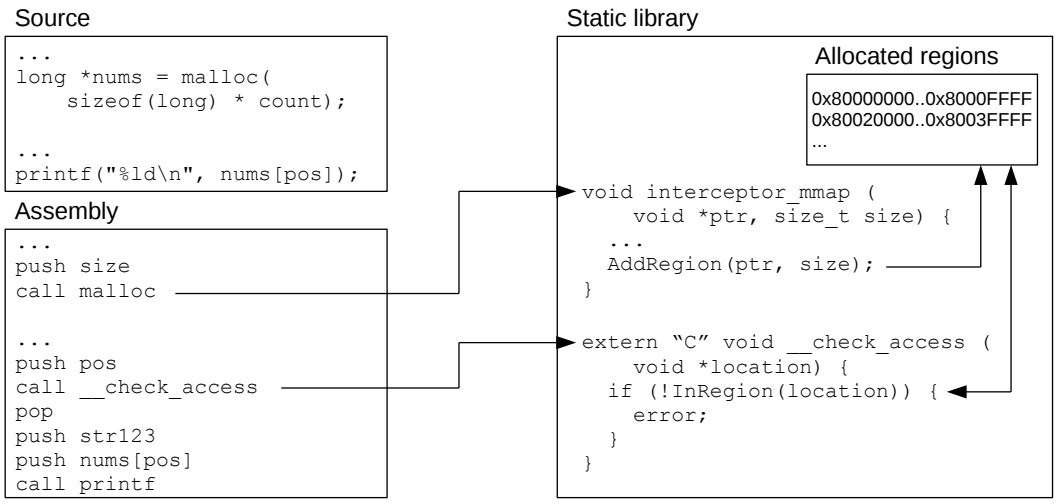


Figure 2.8: A static library is bundled with the executable, which is called into at certain points in the program. This library keeps track of all valid memory regions and ensures all memory accesses are valid.

Many tools exist that can perform program analysis to detect memory violations. A couple of the most widely used analysers are AddressSanitizer [SBPV12] and MemorySanitizer [SS15], two dynamic analysers built into LLVM.

3 Dynamic Race Detection for C/C++11

With the introduction of threads of execution as a first-class language construct, the C/C++11 standards give a detailed memory model for concurrent programs [ISO11b, ISO11a]. 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 to avoid data races.

Prior to the introduction of this memory model, the provision of threads was system and compiler dependent, and the definition of a data race was informal but commonly agreed upon. This lead to much work being created on the detection of data races in C/C++ programs, and other languages such as Java.

Despite the introduction of a formal definition of a data race, much of the work on data race detection for C/C++11 programs is still created with the old informal definition in mind. The most significant reason for this is that 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.

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) behaviours; such weak behaviours 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.

Because of these factors, working out by hand whether a program is race-free, even for small litmus tests, is difficult.

The aim of this chapter 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 [SI09] (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 miss both data races and errors, and report false alarms.

The main research questions considered 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 a memory model-aware dynamic race analysis tool scale 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; by modifying tsan to be fully aware of the memory model, can applications such as these be explored?

Sections 3.1 and 3.2 detail existing work. The rest of the section is original work that builds on these two sections. The original work presented in this chapter is structured as follows:

Extending the vector clock algorithm for C++11 §3.3 The vector clock-based dynamic race detection algorithm [FF09] is extended to handle C++11 synchronisation accurately, requiring awareness of release sequences and fence semantics.

Exploring weak behaviours §3.4 Many C++11 weak behaviours are due to the reads-from relation, which allows a load to read from one of several stores. This section presents the design of an instrumentation library that enables dynamic exploration of this relation, capturing a large fragment of the C++11 memory model.

Operational model §3.5 This section formalises the instrumentation of §3.4 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, the semantics presented here is intended as a basis for dynamic analysis of real-world applications, thus trades coverage for feasibility of implementation.

Characterising the operational model axiomatically §3.6 The practically-focussed design of the operational model in §3.6 means that not all memory model behaviours can be observed. To make this precise, the behaviours that are unobservable have been characterised via a single additional axiom to those of an existing axiomatic formalisation of C++11, and an argument that this strengthened memory model is in correspondence with the operational model is provided.

A detailed implementation and evaluation of the instrumentation described in this chapter is provided in §5.2. The implementation is provided as an extension to ThreadSanitizer.

3.1 C/C++11 Memory Model

The C/C++11 specification provides a unified memory model that abstracts away the memory model of the underlying hardware, allowing programmers to target a single platform [ISO11b, ISO11a]. The C/C++11 memory model has been designed such that it avoids excluding behaviours that a potential underlying hardware memory model could exhibit, as doing so would reduce the usability of C/C++11 for programmers who wish to make full use of such hardware. As a result, the C/C++11 memory model is very permissive in the behaviours that it will allow. In some case, this can lead to some confusing and unintuitive executions.

Four basic low level atomic operations are provided: stores, loads, read-modify-writes (RMWs) and fences. Stores and loads will write to and read from an atomic location respectively. RMWs will modify (e.g. increment) the existing value of an atomic location, storing the new value and returning the previous value atomically. Fences apply memory order constraints on the program. Atomic operations on the same location are safe to use across multiple threads in a racy manner, and are expected to race, as this allows communication between threads without the use of locks. These races are not regarded as *data races* according to the C++11 memory model however.

While atomics allow safe access to atomic locations, on their own they do not help memory accesses that are non-atomic. As shown in §2.2, the general hardware-based solution is to use memory barriers. To briefly recap, the release barrier prevents memory operations ordered before the barrier from being shifted after it, while also releasing, or *publishing*, the side effects for other threads to see. The acquire barrier prevents memory operations ordered after it from being shifted before it. A barrier is implicitly attached to certain atomic operations. When an atomic load with an acquire barrier reads the value stored by an atomic store, the side effects published by any release barrier on that store, and only that store, are guaranteed to become visible to any memory operation ordered after the acquire barrier. This is the most common method of inter-thread communication, called *synchronisation*.

In C/C++11, atomic operations are annotated with a *memory ordering*. There are six types of ordering: relaxed, consume, acquire, release, acquire-release and sequentially consistent. Release and acquire resemble the release and acquire barriers discussed in §2.2. Relaxed applies no barrier semantics, but does have other ordering implications. Sequentially consistent applies release and acquire semantics, and also enforces a strict total order over all operation marked as such, provided the program is race-free. Consume has a special meaning, and in line with many previous works, is not considered in this

work [ND13, BDW16, VBC+15]. The consume ordering is rarely considered due to its unusual semantics and lack of implementation in most compilers.

While these orderings order non-atomic operations, the question now is: what value should an atomic load read? Or rather, what *orders the orderings*? Because it is not feasible to specify precisely what value should be read, or the order in which atomic operations are executed, the C/C++11 memory model instead specifies what *can* and *cannot* happen. This means the model is *axiomatic*, and as such is defined as a set of *axioms*. An execution of a C/C++11 program is only considered valid if it abides by the axioms of the C/C++11 memory model.

The set of executions that a program can exhibit is often referred to as the *behaviours*. A behaviour that cannot be exhibited under strict sequential consistency is called a *weak* behaviour. The release orderings can be ordered by strength as relaxed > release > release-acquire > sequentially-consistent, with a similar ordering for the acquire orders. A weaker ordering here means that it will allow more program behaviours than a stronger ordering. If a particular hardware setup cannot handle a given ordering, it can be strengthened by going up the chain, with the guarantee that a stronger ordering will not introduce new behaviours, but may restrict them; this guarantee has been shown to be false in some circumstances however [BDW16]. By providing this guarantee, a program written to be able to exploit weak behaviours can still be run on hardware that cannot exhibit them, by strengthening the orderings, as long as the hardware is at least capable of strict sequential consistency. Any hardware that can enforce strict sequential consistency can satisfy the C/C++11 memory model.

The rest of this section is dedicated to formalising the C/C++11 memory model. This follows the *Post-Rapperswil* formalisation of Batty et al. [BOS+10]. Although recent works have condensed the formalisation [BDW16, VBC+15], the descriptive presentation of [BOS+10] provides a greater degree of intuition, especially for designing the instrumentation framework in §3.4.

Pre-executions A program execution represents the behaviour of a single run of the program. These are shown as execution graphs (e.g. Figure 3.2c), where nodes represent memory events. For example, $\mathbf{a{:}W_{rel}x{=}1}$ is a memory event that corresponds to a release 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 the value written. For example,

b:$\mathbf{RMW_{ra}}$**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 **c**:$\mathbf{F_{rel}}$.

Execution graphs are used throughout this work to provide a clear representation of specific executions. These graphs are best viewed in colour. In each graph, events in the same column are issued by the same thread.

Sequenced-before (*sb*) is an intra-thread relation that orders events by the order they appear in the program. The *sb* relation is not total within a thread, as operations within an expression are not ordered. For example, `++i + ++i` will result in two unsequenced loads of `i`.

Additional-synchronises-with (*asw*) causes synchronisation on thread launch, between the parent thread and the newly created child 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 child-parent join operation in the parent thread.

The events, *sb* edges and *asw* edges form a *pre-execution*. This represents one possible flow of the program according to the control flow of the program, with arbitrary values for the memory loads. Due to the presence of branches, there can be many, possibly infinitely many, pre-executions. A pre-execution represents a possibility, and may or may not be able to be extended to an actual execution.

Figure 3.2a shows an execution graph for a pre-execution of the program in Listing 3.1. Because there is no information on what store the loads have read from, the values of the loads are ambiguous. Unnecessary information is often omitted from these graphs to prevent them from becoming cluttered. Most notably, the initialisation of variables and the additional-synchronises-with edges will usually be omitted.

Candidate Executions Each pre-execution can be extended with a set of *witness relations*, to give a *candidate execution*. These relations represent the runtime observations of the execution. Not all pre-executions can be extended to a candidate execution, e.g. when a load cannot be matched with any store. Consider the pre-execution of Figure 3.2a in which event g reads the value 4, there will be no store that can be matched to g, and thus no candidate executions.

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. Each load must have exactly one *rf* edge incident on it.

```
int nax = 0;
std::atomic<int> x(0);
void T1() {
  nax = 1;
  x.store(1, std::memory_order_release);
  x.store(3, std::memory_order_relaxed);
}
void T2() {
  x.store(2, std::memory_order_relaxed);
}
void T3() {
  x.load(std::memory_order_acquire);
  nax; // read from 'nax'
}
```

Figure 3.1: Simple racy C++11 program.

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

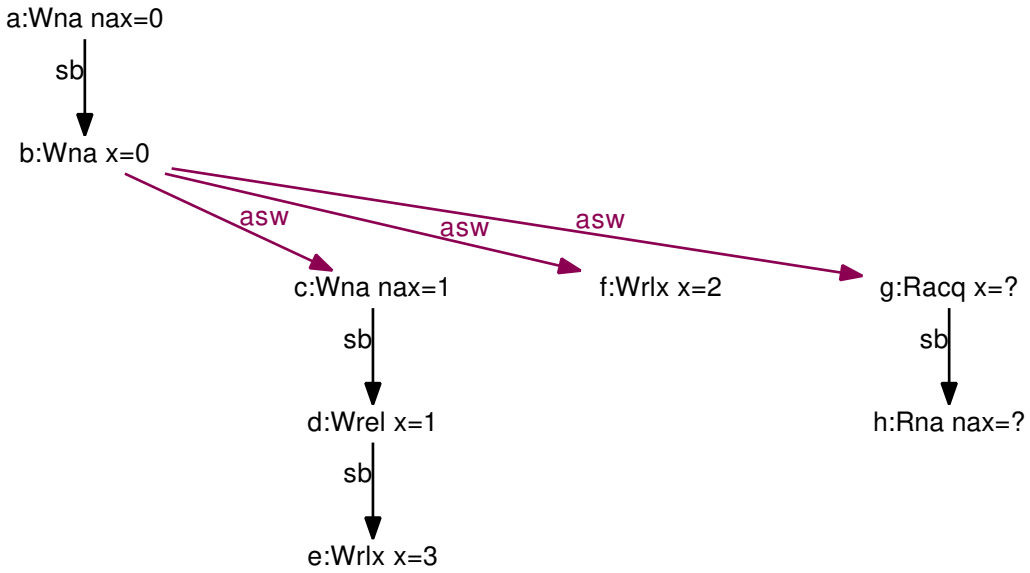
Sequentially-consistent (*sc*) order is a total order over all atomic operations in the execution marked with sequentially-consistent ordering. It is expected that the other relations, *sb*, *rf* and *mo*, do not conflict with *sc*, thus restricting many of the behaviours that would otherwise be allowed.

The set of all candidate executions for a given program is called the *candidate set*. Figure 3.2b shows one possible candidate execution for the pre-execution of Figure 3.2a. The number of ways in which a pre-execution can be extended grows very quickly with the number of memory events, due to the many ways in which the relations may be arranged. For example, the modification order over the 4 writes to x in Figure 3.2a can be arranged in 24 different ways. When combined with the 4 possible stores the load in g can read from and the 2 stores for h, gives 192 candidate executions.

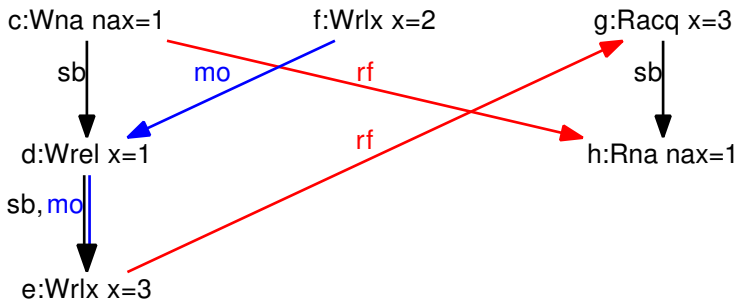
A candidate execution does not have to be viable according to the C/C++11 memory model, as checking for consistency comes at a later step. Many of these candidates will therefore have seemingly bizarre behaviours. For example, Figure 3.3 shows another candidate execution for the pre-execution of Figure 3.2a, with *mo* going *backwards* in program order.

Derived Relations A candidate execution must still be checked against the C/C++11 memory model for consistency. To simplify the consistency rules, the candidate execution can be extended with a set of *derived relations*.

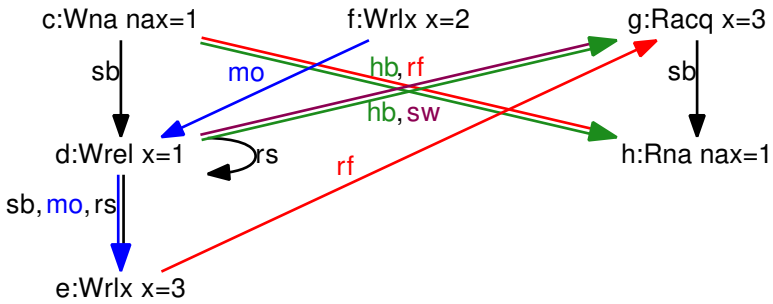
A *release-sequence* (*rs*) represents a continuous subset of the modification order. An *rs* is headed by a release store, and continues along each store to the same location. The *rs*



(a) A pre-execution of the program in Listing 3.1, the only possible pre-execution for this program.



(b) The pre-execution in Figure 3.2a has been extended with a set of possible witness relations to give a candidate execution. The initialisations of the main thread have been excluded.



(c) Given the candidate execution of Figure 3.2b, you can derive the derived relations. The hypothetical release sequence and some happens-before relations have been excluded for clarity.

Figure 3.2: Pre-execution, witness and derived relations for the program in Listing 2.1.

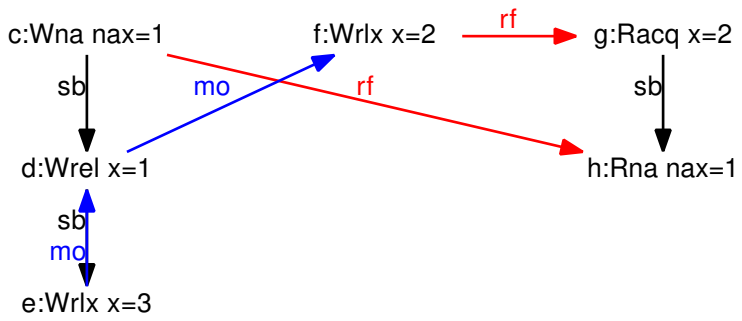


Figure 3.3: A candidate execution for the pre-execution of Figure 3.2a. The modification order will prevent this from being allowed by the C/C++11 memory model.

is *blocked* when another thread performs a store to the location. An RMW from another thread will however continue the *rs*. An example of this is shown in Figure 3.2c, whereby an *rs* headed by d extends down to e. Another example is shown in Figure 3.4, where instead the *rs* headed by d is blocked by the store in another thread labelled f.

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. This means that release sequences are necessary for synchronisation, and so the aforementioned release sequence blocking behaviour can be detrimental if synchronisation is required. An *asw* edge is also an *sw* edge.

Happens-before (*hb*) is simply $(sb \cup sw)^+$ (where $+$ denotes transitive closure), representing Lamport's partial ordering over the events in a system [Lam78]. Because an *sw* edge is also an *hb* edge, when event a in thread A synchronises with event b in thread B, every side effect sequenced before a in A will become visible to every event sequenced after b in B.

Figure 3.2c shows the candidate execution of Figure 3.2b that has been extended with the derived relations. The release store of d heads a release sequence, that then extends down to e. When the acquire load of g reads from e, the event that headed the release sequence synchronises with g. Due to synchronisation, the non-atomic write of c happens before the non-atomic read of h.

Figure 3.4 shows another candidate execution. The release sequence is blocked by the

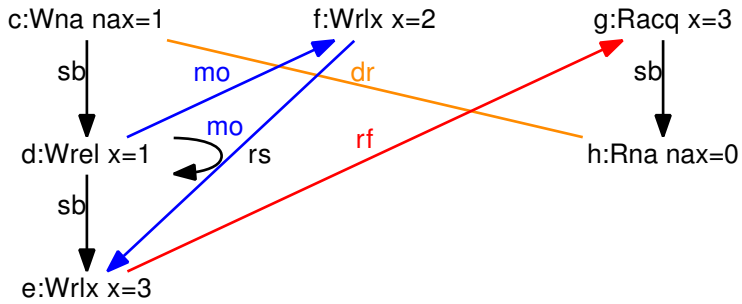


Figure 3.4: The release sequence headed by d is blocked by event f, causing a data race between c, the non-atomic write to **nax**, and h, the non-atomic read from **nax**; if the blocking event f is removed, there is no race.

store of f. As e is not a part of the release sequence, there is no synchronisation, and so the non-atomic accesses are not ordered.

Data Races A further relation can now be defined that will identify data races. To reiterate, a data race occurs between two memory accesses when both operate on the same location, at least one is non-atomic, at least one is a store, and neither happens before the other. The *hb* relation defined above suitably fits this notion of happens before. Figure 3.4 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 C/C++11 standards state that data races are undefined behaviour. It does not matter if a particular execution does not have a data race; if any of the consistent executions have a data race the behaviour of the program is undefined.

The negative consequences of data races are well known from a research perspective [Adv10a, Adv10b], but to a regular programmer, they can seem inconsequential. The obvious assumption is that the physical representation of the bits in memory may become mangled after two concurrent stores. In practise, this is not an issue for most systems, especially with the introduction of transactional memory [CSW18]. The bigger issue lies with the compiler. Consider a C++11 compiler that takes as input a well formed C++11 program. The compiler will assume that any program given to it is well formed, and thus race free. A programmer can still give it a racy program, but the compiler will transform it under the assumption that it is race free. For example, consider a thread that performs a series of non-atomic stores, but does not follow up with a release atomic operation. From the compiler's perspective, no other thread is going to try to access the locations of these

stores, as doing so would constitute a data race. The compiler may therefore simply omit the stores for efficiency, leaving them invisible to other threads.

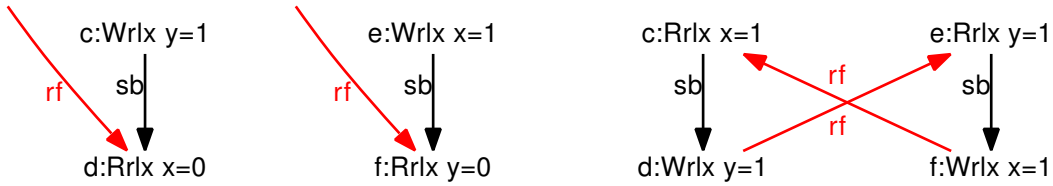
Consistent Executions A *consistent execution* is a candidate execution that abides by the C/C++11 memory model. The *consistent set* is a candidate set filtered by consistency. The rules for consistency are presented as a set of *axioms*. For an execution to be consistent it must satisfy all of the axioms. The set of executions that are allowable is the consistent set, with the caveat that if any consistent execution contains a data race, the set of allowed executions is empty and the program is undefined.

There are seven axioms that determine consistency [BOS+10]. Some of these axioms will be simplified, as consume ordering and locks are not being considered. A brief overview of each axiom is given in this section, but will be covered in more detail where appropriate when describing the exploration of weak behaviours in §3.4.

The *well-formed-threads* axiom restricts the formation of memory events, *sb*, and *asw*. This will prevent considering pre-executions that will only lead to inconsistent executions. The *well-formed-rf-mapping* axiom similarly restricts *rf*, such as not allowing a load specified at one location reading from a store to another location, or a load reading from multiple stores. This will prevent considering candidate executions that will trivially be inconsistent. The *consistent-locks* axiom is not considered, as locks have not been affected by this work.

The three axioms, *consistent-sc-order*, *consistent-mo* and *consistent-rf-mapping*, correspond with the formation of the *sc*, *mo* and *rf* relations. These are non-trivial, and are covered in more detail in §3.4. The last axiom is *consistent-ithb* axiom which, without consume, simply requires *hb* to be irreflexive.

As long as an execution follows these axioms, it is deemed a valid execution of a well formed C/C++11 program. This leads to some interesting behaviours. A *weak* behaviour is one that would not appear under any interleaving of the threads using sequentially consistent semantics. To illustrate this, Figure 3.5 shows two such executions that arise from well-known litmus tests [AMT14, BOS+11, BWB+11, ND13]. 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 could not have occurred yet at that point in the execution. While these behaviours are allowed by the memory model, whether they are observed in practice will depend on practical issues such as the effect of compiler reorderings and properties of the hardware on which a program is executed.



(a) Store buffering. **(b)** Load Buffering.

Figure 3.5: Example executions showing some of the common weak behaviours allowed by the C/C++11 memory model.

3.2 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 [FF09, Mat88, PS03, PS07, ISZ99]. It aims to 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. The epoch for thread t' in $\mathbb{C}_t$ represents the logical time of the last instruction executed by t' that happens before any instruction thread t will execute in the future. The local epoch for thread t, $\mathbb{C}_t(t)$, is denoted $c@t$.

VCs have an *initial value*, $\bot_V$, a *join* operator, $\cup$, 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, $\mathbb{L}_m$, which 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

```
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
}
```

Figure 3.6: 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.

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, specific checks are performed to ensure 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, denoted W_x, needs to be known at any point. As data races do not occur between reads, the reads 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 data 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 data race:

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

Example An illustration of the VC-based race detection algorithm is provided using the example of Figure 3.6, 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 $\mathbb{R}_{\texttt{nax}} = \mathbb{L}_x = \bot_V$. Because `nax` has not been written to, $W_{\texttt{nax}}$ has initial value 0@`T1`, where the choice of `T1` is arbitrary: epoch 0 for any thread would suffice [FF09].

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`,

Statement	$\mathbb{C}_{\texttt{T1}}$	$\mathbb{C}_{\texttt{T2}}$	$\mathbb{C}_{\texttt{T3}}$	$\mathbb{L}_{\texttt{x}}$	$\mathbb{R}_{\texttt{nax}}$	$\mathbb{W}_{\texttt{nax}}$
-	$(1,0,0)$	$(0,1,0)$	$(0,0,1)$	$\bot_V$	$\bot_V$	-
A						1@T1
B	$(2,0,0)$			$(1,0,0)$		
C		$(1,1,0)$				
D						
E			$(1,0,1)$			
F			$(1,0,1)$			1@T1

Figure 3.7: Trace of the program in Figure 3.6, showing the value of the VCs after each statement. Only updated values are shown, and those where race detection checks are performed.

$\mathbb{C}_{\texttt{T2}} := \mathbb{C}_{\texttt{T2}} \cup \mathbb{L}_{\texttt{x}} = (1,1,0)$. 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.

A sketch of this execution is shown in Figure 3.7. Only updates are shown, except where a race condition check is performed, which is coloured red. A check is performed during statement F, which shows that $1 \leq \mathbb{C}_{\texttt{T3}}(\texttt{T1})$ and so there is no race.

3.3 Data Race Detection for C++11

The rest of this section describes original work, that build upon the pre-existing work outline in sections 3.1 an 3.2.

The traditional VC algorithm outlined in §3.2, and implemented in tsan, is defined over simple release and acquire operations. It is unaware of the more complicated synchronisation patterns of C/C++11. This section describes an updated VC algorithm that properly handles C/C++11 synchronisation, by showing where the original VC algorithm falls short and how the updated algorithm fixes these shortcomings. The algorithm is summarised as a set of inference rules in §3.3.3.

3.3.1 Release Sequences

As described in §3.1, release sequences are key to synchronisation in C++11. Event a will synchronise with event b if a is a release store, b is an acquire load, and b reads from a

Statement	$\mathbb{C}_{\texttt{T1}}$	$\mathbb{C}_{\texttt{T2}}$	$\mathbb{C}_{\texttt{T3}}$	$\mathbb{L}_{\texttt{x}}$	$\mathbb{R}_{\texttt{nax}}$	$\mathbb{W}_{\texttt{nax}}$
-	$(1,0,0)$	$(0,1,0)$	$(0,0,1)$	$\bot_V$	$\bot_V$	-
A						1@T1
B	$(2,0,0)$			$(1,0,0)$		
C		$(1,1,0)$				
D				$\bot_V$		
E			$(0,0,1)$			
F			$(0,0,1)$			1@T1

Figure 3.8: Trace of the program in Figure 3.6, showing the value of the VCs after each statement. Only updated values are shown, and those where race detection checks are performed. Blue updates show those that differ from the sketch of Fig.3.7.

store in the release sequence headed by a. This subsection explains why the existing VC algorithm does not accurately capture release sequence semantics, and how the new VC algorithm will fix these shortcomings.

Blocking Release Sequences Recall the execution of Figure 3.4. 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. The effect is that release sequences will continue unhindered until some thread performs a release store to the same location.

To adapt the VC algorithm to correctly handle the blocking of release sequences, each location m will additionally store the id of the thread that performed the last release store to m. Let $\mathbb{T}_m$ denote the mapping from location m to the 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 denotes 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: $\mathbb{L}_m := \bot_V$. This has the effect of blocking the release sequence started by $\mathbb{T}_m$.

A similar sketch to that of Figure 3.7 is provided in Figure 3.8, showing that $\mathbb{L}_{\texttt{x}}$ is cleared out on the store in statement D. This results in the non-atomic load in statement D becoming racy with respect to the store in statement A.

This aspect of the memory model is often seen as a "bug", and not intended, mostly due to its unintuitiveness and lack of representation in real hardware memory models [VBC+15]. As such, there has been work on trying to improve the C/C++11 mem-

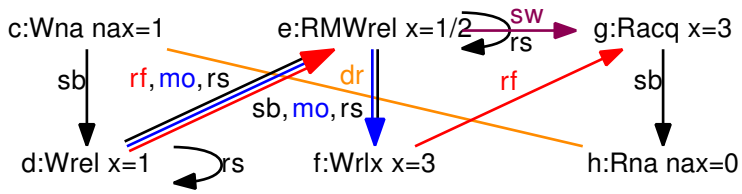


Figure 3.9: The release sequence started by d and continued by e is blocked by f; thus d does not synchronise with g, so c races with h.

ory model by removing these aspects [VBC+15]. Nevertheless, this work formalises the C/C++11 memory model as it stands.

Example revisited Recall from Section 3.2 the worked example of the VC algorithm applied to schedule `A`–`F` of Figure 3.6, in which a data race on `nax` is missed by the original VC algorithm. Revising this example taking release sequence blocking into account, 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 $\mathbb{C}_{\mathtt{T3}}(\mathtt{T1}) = 0$. The data 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 started by multiple threads. 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 3.9, 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, $\mathbb{L}_m$ will now 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 $\mathbb{L}_m$. Now consider the case of collapsing release sequences when a thread t that started a release sequence on m performs a relaxed non-RMW store. $\mathbb{L}_m$ must be replaced 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 t performed the original release store. To preserve this information, each location m will now have a *vector of vector clocks* (VVC), denoted $\mathbb{V}_m$, that stores the VC for each thread that has started a release sequence on m.

Statement	$\mathbb{C}_{\texttt{T1}}$	$\mathbb{C}_{\texttt{T2}}$	$\mathbb{C}_{\texttt{T3}}$	$\mathbb{L}_{\texttt{x}}$	$\mathbb{V}_{\texttt{x}}$	$\mathbb{R}_{\texttt{nax}}$	$\mathbb{W}_{\texttt{nax}}$
-	$(1,0,0)$	$(0,1,0)$	$(0,0,1)$	$\bot_V$	$\emptyset$	$\bot_V$	-
c							1@T1
d	$(2,0,0)$			$(1,0,0)$	$((1,0,0),\bot_V,\bot_V)$		
e		$(0,2,0)$		$(1,1,0)$	$((1,0,0),(0,1,0),\bot_V)$		
f				$(0,1,0)$	$(\bot_V,(0,1,0),\bot_V)$		
g			$(0,1,1)$				
h			$(0,1,1)$				1@T1

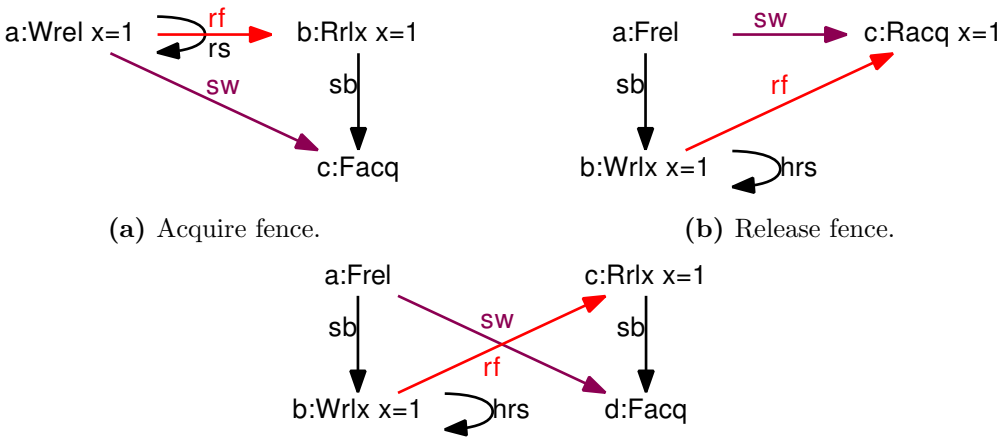
Figure 3.10: Trace of the program in Figure 3.9, showing the value of the VCs after each statement. Only updated values are shown, and those where race detection checks are performed.

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 the store heads a new release sequence, and $\mathbb{C}_t \geq \mathbb{V}_m(t)$, the previous contents of $\mathbb{V}_m(t)$ are redundant, and so are discarded. Thread t will now be 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 $\mathbb{V}_m(t)$, which is 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, when a thread acquires from m, it will synchronise with all threads that head a release sequence on m.

In practice, the full generality of a VVC for each location is not needed, and would be wasteful. An implementation would instead use a mapping from thread ids to VCs that grows on demand when threads perform RMWs. For example, if a location only has two active release sequences on it, the VVC for the location will only have two thread id to VC pairs.

To give an example of how these new VVCs work, another sketch is provided in Figure 3.10. This time the sketch is on the program fragment of Figure 3.9, where the statements are carried out in the order *cdefgh*. After statement *f*, the VC for thread `T1` has been cleared out of both the VC and the VVC for location x.



(a) Acquire fence.

(b) Release fence.

(c) Acquire and release fences.

Figure 3.11: Synchronisation caused by fences.

3.3.2 Fences

Fences provide many of the same guarantees as the other atomic operations, with the difference that they do not operate on any specific memory location. A fence 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 §3.4.4. As described in §3.2, the common VC algorithm uses VCs indexed by locations for the purpose of synchronisation. By not considering fence synchronisation, a tool that implements this algorithm is going to report false positives.

The three cases of synchronisation with fences are shown in Figure 3.11. 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 3.11a. Release fences use the hypothetical release sequence, described in §3.1. 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 3.11b. Release fences and acquire fences can also synchronise with each other, shown in Figure 3.11c.

Handling synchronisation from release sequences uses the VC associated with an atomic location to convey VC information. With fences, as there is no atomic location, another intermediate VC must exist. Furthermore, the fence itself does not start a hypothetical release sequence, atomic stores do, and they can be blocked and restarted any number of times. Using $\mathbb{C}_t$ in these cases is not correct, as $\mathbb{C}_t$ will have been updated since the release fence. A similar issue exists with acquire fences, as synchronisation does not occur on the

relaxed load, but the acquire fence that follows. To handle fence synchronisation, the VC 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 that VC. To handle this, two new per-thread VCs will be introduced to track this information: the *fence release clock*, $\mathbb{F}_t^{rel}$, and the *fence acquire clock*, $\mathbb{F}_t^{acq}$. The VC algorithm is then extended 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 3.11c. 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, $\mathbb{C}'_u = \mathbb{C}_u \cup \mathbb{F}_u^{acq\prime} \geq \mathbb{C}_u \cup \mathbb{F}_t^{rel} = \mathbb{C}_u \cup \mathbb{C}_t$. Thus there is synchronisation between a and d.

3.3.3 Algorithm

The extended VC algorithm, combining the original VC algorithm of [FF09] with the techniques described in §3.3.1 and §3.3.2 for handling release sequences and fences, is summarised by the inference rules of Figure 3.12. The rules for non-atomic reads and writes have been omitted, as these are unchanged.

For each thread t, a vector clock $\mathbb{C}_t$, and fence release and acquire clocks, $\mathbb{F}_t^{rel}$ and $\mathbb{F}_t^{acq}$ (see §3.3.2) are recorded. For each variable m, both a vector clock $\mathbb{L}_m$ and vector of vector clocks $\mathbb{V}_m$ (see §3.3.1) are recorded. The symbols $\mathbb{C}$, $\mathbb{F}^{rel}$ and $\mathbb{F}^{acq}$, and $\mathbb{L}$ and $\mathbb{V}$ are used to denote these clocks across all threads and locations respectively.

The algorithm shown in Figure 3.12 is just one of multiple possible variants, the choice of which will be a trade-off between efficiency and convolution. The one chosen here is simple, but always uses the VVC, $\mathbb{V}_m$, even when there is only one active release sequence or hypothetical release sequence active on m. Because the VVC is always in use, the mapping from location to thread id shown in §3.3.1, $\mathbb{T}_m$, is not needed.

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 never stored in the VVC, because if a thread performs a relaxed store requiring the VVC to collapse, $\mathbb{F}^{rel}$ will be joined onto the VC for the location regardless.

STATE:

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

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

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]

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

[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]

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

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]

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

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]

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

Figure 3.12: Semantics for tracking the happens-before relation with loads, stores, RMWs and fences.

3.4 Exploring Weak Behaviours

The C/C++11 memory model allows programs to exhibit various weak, non-SC behaviours, as described in §3.1. For a dynamic analysis tool it can be difficult to analyse properties of programs with such behaviours. Typically a dynamic analysis tool resides in a part of the program's memory, and acts as a kernel through which the rest of the program will interact at certain points. The tool becomes aware of events as and when they happen, implying that there is some strict total order over the events of the program, or strict partial order if the kernel allows concurrency between certain events. Any data race that relies on non-SC behaviours to appear will be impossible for a race detector such as tsan to detect. For example, the behaviour of Figure 3.5a cannot be exhibited under sequential consistency, and so any race that depends on $x = 0$ and $y = 0$ will not be detected.

The design of a novel library to address this issue is presented in this subsection. The library 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 core principle 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. The information for each store will be sufficient that it can be determined whether the store can be read by a load when the load would occur.

By controlling the order in which threads are scheduled and the stores from which atomic load operations are read, the instrumentation enables exploration of a large set of non-SC behaviours. The buffering-based approach has some limitations, for example it does not facilitate a load reading from a store that has not yet occurred. The fragment of the memory model covered by this technique is formalised in §3.6.2. This instrumentation forms the basis for an extension of the tsan tool for the detection of data races arising from non-SC program executions by randomising the stores that are read from by atomic loads. This extension is discussed in §5.2.

As stated in §3.1, the Post-Rapperswil memory model presentation of Batty et al. [BOS+10] is followed in the design of the instrumentation library presented here. The notation "§PRX" is used to refer to section X of the Post-Rapperswil formalisation.

Going back to the witness relations described in §3.1, it is these relations that differentiate one run of a program from another. An ideal tool would be able to explore all the possible arrangements of these relations, while ignoring 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. The different arrangements of *mo* and *sc* can be handled by exploring different thread schedules, whereas the *rf* relation cannot. Exploring the *rf* relation requires the tool to know all the stores that each load could read from. The *rf* relation can be thought of as the source of weak behaviours.

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. This is an axiom, and is referred to as **AxSB**. Also assume that the order in which sequentially consistent operations are carried out conforms with the *sc* relation, and is referred to as **AxSC**. Axioms that require showing conformance between certain relations are described in brief, but nonetheless, these will be useful in showing that our instrumentation follows the C++11 memory model.

3.4.1 Post-Store Buffering

Consider a thread that performs an atomic store to some atomic location, after which some thread, possibly the same one, performs another atomic store to the same location. Now consider a different thread performing an atomic load to the same location. Depending on the state of the loading thread, and the memory ordering used, the atomic load should be able to read from the first store, even though there are intervening stores to the location. To facilitate this, each atomic store to each atomic location will be recorded in a buffer, allowing an instrumentation library to search through and pick a valid store to read from.

The core idea is as follows. On intercepting a store to location m, the VC updates described in §3.3 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, upon each load, the instrumentation library can be certain that reading from the store will lead to a consistent execution. The meta-data required to ensure this is covered throughout the rest of §3.4, and is guided by the C/C++11 consistency axioms. The store buffer is formally defined in §3.5.

This approach is referred to as *post-store* buffering, as each store is buffered after the real store has been performed. The alternative is *pre-store* buffering, which would speculatively place stores in the buffer, on the assumption that threads would later perform them. Atomic loads could then read from future stores, as in the example of Figure 3.5b. This

approach is very difficult to perform in a dynamic environment however. For example, what would happen if a load reads from a store that does not end up happening? Rewinding large applications is not trivial, and so is not considered in this work.

3.4.2 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) *hb* restricted to the writes at a location is a subset of *mo*, in other words, *mo* conforms with *hb*. (3) Restricting the composition of (sb ; $\mathbf{F}_{sc} \overset{sc}{\nrightarrow} \mathbf{F}_{sc}$; sb) to the writes at a location is a subset of *mo*.

The store elements in each store buffer form an ordered list. When an atomic store to m occurs, the store element for the store is pushed to the back of the list. This ordered list represents the modification order for the location, and must be a strict total order, satisfying (1).

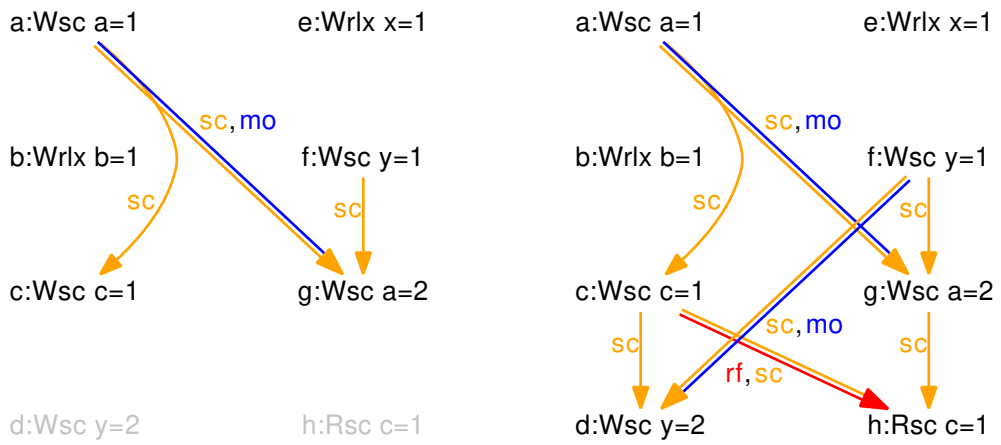
To satisfy (2), it must be shown that *mo* conforms with *hb*, which is the transitive closure of *sb* and *sw*, thus it must be shown that *mo* conforms with each of *sb* and *sw*. The **AxSB** axiom already shows that *mo* conforms with *sb*. Synchronisation follows the *rf* relation, and *sb* when fences are involved. As a load can only read from a store already in the store buffer, due to the **AxSB** axiom, *mo* must conform with the composition ($sb \circ rf \circ sb$), and so (2) is satisfied. The agreement between *mo* and *hb* shown here is also referred to as coherence of write-writes (CoWW).

Due to **AxSB** and **AxSC**, (3) holds trivially.

3.4.3 Consistent SC Order (§PR6.16)

The *consistent*$-sc$ axiom requires the following: (1) *sc* is a strict total order over all events with *sc* ordering. (2) *sc* conforms with *hb*. (3) *sc* conforms with *mo*.

Rule (1) sounds trivial, and generally is, as the instrumentation will not weaken operations that have SC ordering. To preserve parallelism, only certain operations will compete over locks, such as stores and loads that access the same location, and so the instrumentation will only enforce the *sc* ordering between certain pairs of operations. However, it can be shown that for certain pairs, the ordering is irrelevant. One way to show this is through the construction of a program execution. Figure 3.13 shows an execution gradually being formed. Due to **AxSC**, SC events ordered by *sb* are also ordered by *sc*. Events a and e remain unordered. Figure 3.13a shows events a and g both performing an SC store to the same location which, due to the reasoning given in §3.4.2, will cause an *sc* edge to form between them. Similar reasoning can be given for the two new events in Figure 3.13b.



(a) Before d and h, only sc edges that the instrumentation would enforce have been shown.

(b) After d and h, the new sc edges conform with mo and rf.

Figure 3.13: Construction of a program execution.

The result is that whenever two events would create an sb, mo or rf edge, and both have SC ordering, an sc edge will also form. Because these edges form the candidate execution, how other SC events are ordered cannot affect the execution. SC events such as a and f will still be ordered by sc, as the instrumentation does not weaken their ordering, but the direction of the ordering is immaterial to the instrumentation. Rules (2) and (3) are now trivially satisfied.

3.4.4 Consistent Reads From Mapping (§PR6.19)

The rf requirements are the most complex among the consistency rules. These are broken 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 3.22 and discussed in §3.5.

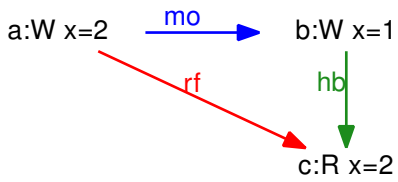
Coherence Rules There are three coherence rules with regards to reads, and one consistency rule for RMWs.

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

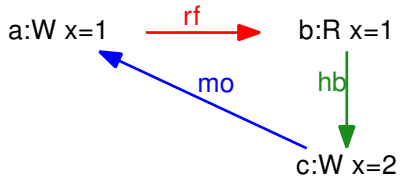
(2) Coherence of Read-Writes (CoRW) states that a load cannot read from a store if there is another store earlier in mo that happens after the current load. This will cut off

all of the modification order 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 leave a range of stores in mo that can potentially be read from.

The instrumentation automatically conforms to CoRW, as 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 the instrumentation does not allow. This is illustrated by the execution fragment shown for CoRW above. This reasoning also assumes that the instrumentation follows 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, the instrumentation library can then search the store buffer to find the latest store in mo that happened before the current 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 the search is halted.

(3) Coherence of Read-Reads (CoRR) 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 stores ordered before the store in mo will be cut off for future loads. Loads from other threads will also be affected when synchronisation occurs. Consider the execution fragment shown in Figure 3.14. 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, or 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 necessary for a thread to be aware of loads performed by other threads. To handle this, the instrumentation library will make use of software load buffers

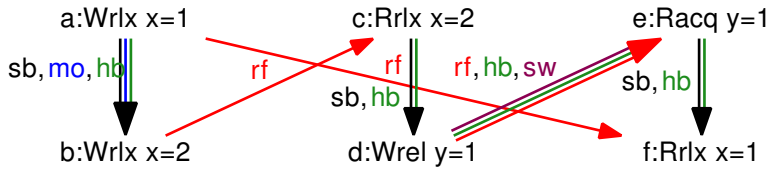


Figure 3.14: Inconsistent execution fragment caused by lack of CoRR.

as follows. Every store element is augmented 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 that 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 the 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 the load. If $c \leq \mathbb{C}(t)$, then the load associated with the load element happened before the current load, and the search is halted. 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.

(4) Consistent RMW reads states that if an RMW event b reads from write event a, then b must follow a in mo. With the 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 SC operations in a number of ways. These are outlined as follows, where $\overset{sc}{\longrightarrow}$ denotes an inter-thread sc edge:

(5) $\mathbf{W}_{\textbf{non-}SC} \overset{sb}{\longrightarrow} \mathbf{F}_{SC} \overset{sc}{\longrightarrow} \mathbf{R}_{SC}$: 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) $\mathbf{W}_{SC} \overset{sc}{\longrightarrow} \mathbf{F}_{SC} \overset{sb}{\longrightarrow} \mathbf{R}_{\textbf{non-}SC}$: 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) $\mathbf{W}_{\textbf{non-}SC} \overset{sb}{\longrightarrow} \mathbf{F}_{SC} \overset{sc}{\longrightarrow} \mathbf{F}_{SC} \overset{sb}{\longrightarrow} \mathbf{R}_{\textbf{non-}SC}$: 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.

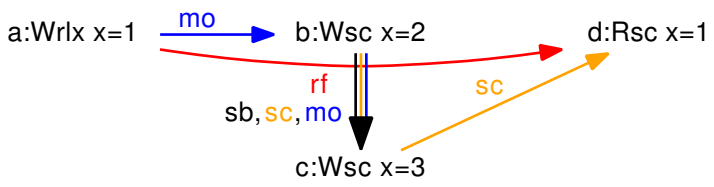


Figure 3.15: Consistency of *sc*-reads only forbids d reading from b.

Accommodating SC fences cannot be trivially solved using the existing machinery, and requires additional VCs and VC manipulation on every SC operation. To start off, two new global VCs will be defined: $\mathbb{S}_F$, 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, assume it has reached a store to x performed by thread u at epoch $c@u$. If the load is an SC load, and $\$_{R,t}(u) \geq c$, then the search is halted according to (5). If the store is an SC store, and $\$_{W,t}(u) \geq c$, then the search is halted according to (6). Regardless of whether the load or the store is SC, if $\$_{F,t}(u) \geq c$ then (7) applies.

(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 3.15 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 does not involve SC fences, and will not be 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 (8), each store element will additionally have an *sc* flag. The flag is initialised to *true* if the store is SC, or *false* if not. When a thread performs an SC store, every store element in the store buffer that happens before the current store has their *sc* flag set to *true*. When a thread performs an SC load, it searches through the store buffer for a set of possible stores to read from as usual, but out of all of those stores that are marked

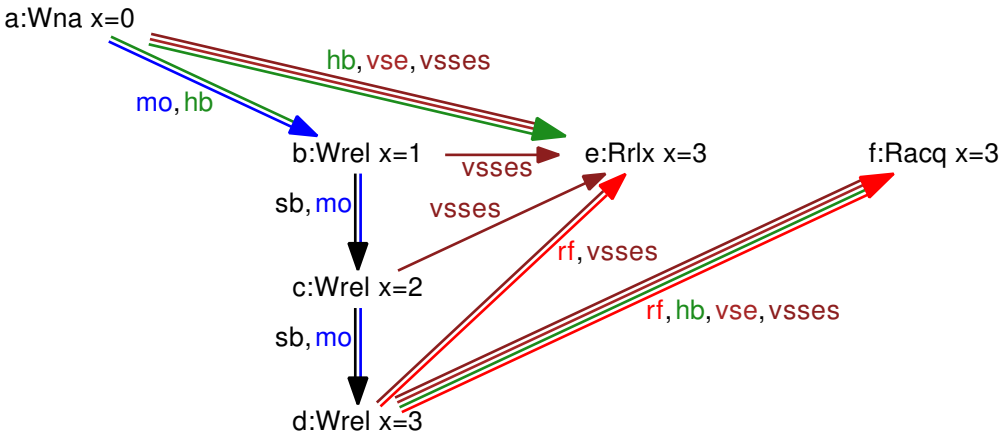


Figure 3.16: Visible side effects and visible sequence of side effects.

sc, only the latest one in the modification order can be read from. The *sc* flag acts as a marker indicating which stores have been hidden due to the effects of other SC stores.

Visible Side Effects One of the parts of the C/C++11 memory model that has not been covered is the notion of visible side effects. This is fairly complex to describe, but is mostly already covered by the instrumentation above, and so no special treatment is provided for it in the instrumentation. A brief explanation is given here for the sake of completeness.

Visible-side-effect (*vse*) relates stores and loads based on happens-before visibility. A store a is a visible side effect for a load b if a happens before b and there is no intervening store c such that a happens before c and c happens before b. This applies to both atomic and non-atomic operations.

visible-sequence-of-side-effects (*vsses*) is a similar relation between stores and loads that only applies to atomic operations. The *vsses* represents a contiguous part of the modification order starting from the visible side effect and ending before the first store that the load happens before.

Figure 3.16 shows an example of these relations. The *vsses* for the relaxed load of e includes all of a, b, c and d. For the acquire load of f, as the load causes synchronisation, d retroactively becomes the visible side effect for f, and thus the only store in the *vsses* for f.

The remaining two consistency rules are as follows: **(9)** A non-atomic load must read from a visible side effect. **(10)** An atomic load must read from a store that is a part of the *vsses*. What isn't apparent from these descriptions is that this addresses the *initialisation*

problem, as opposed to there being too many things to read from. For the non-atomic case, having two or more visible side effects would imply a data race between them, as they would not be ordered by happens before. For the atomic case, the CoWR rule will prevent the load from reading from a store that is *mo* ordered before the visible side effect, and thus not a part of the *vsses*.

The problem these rules address is there being *no* visible side effects, and so no *vsses*. For a non-atomic load either the location has not been initialised or the load and the initialisation form a data race. In the atomic case, the load could potentially read from a store that causes synchronisation, but there may be no stores to read from or all of the stores have relaxed ordering with no preceding fences. A load that cannot read from a visible side effect or *vsses* is called an *uninitialised load*.

In the context of C++, an uninitialised load on an atomic location is not usually an issue. If an atomic has global scope, it will be constructed statically, and therefore initialised before the program has begun. If it is a class member, it will be constructed as the object is constructed. Only indirection could cause an issue, at which point there will likely be several other memory issues. For these reasons, (9) and (10) have not been given much consideration.

3.5 Operational Model

In order to make the instrumentation of §3.4 sufficiently precise such that it can be implemented as a tool and reasoned about, a formalisation of the instrumentation is provided. This formalisation is presented as an operational semantics for a core language, and then used in §3.6 to argue that the instrumentation matches an axiomatically-defined fragment of the C++11 memory model.

3.5.1 Programming Language Syntax

The formal operational model is presented 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 3.17. A program is a sequence of statements that are executed by an initial thread. Identifiers `LocA` and `LocNA` 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 (the set of operators that may appear in expressions is left 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

```
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 3.17: Syntax for the core language.

performing an atomic operation. Atomic operations, described by the `StmtA` production rule, consist of loads, stores, RMWs and fences. An `RMW` takes a function, `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.

3.5.2 Operational Model Formalised

The structure of the state of a program is shown in Figure 3.18, which takes inspiration from prior work [WBBD15]. It describes the set of possible states a program can be in, and includes the machinery described in §3.5 that allows for the exploration of weak behaviours. Figure 3.18b gives a pictorial representation of the state, giving an intuitive view of how the state described formally in Figure 3.18a 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 3.17.

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 ($\mathbb{L}$).

3.5.3 Operational Semantics

Figures 3.19 to 3.20 show the state transitions for the operational model. These transitions are defined for each atomic instruction in the simple language, and for a few internal instructions that do not appear in source programs.

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 3.18. 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. Note that carrying T around is redundant, but simplifies the rules by not having to repeatedly index into Σ.

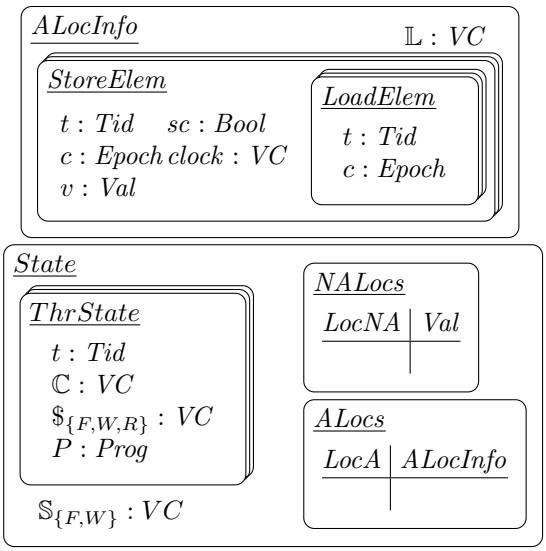
Figure 3.19 gives the semantics for atomic statements. Each atomic function will call into the appropriate sequentially consistent helper function of Figure 3.20, and the appropriate buffer implementation functions. These SC helpers perform the updates described in the SC fence section of §3.4.2, or nothing, if the memory ordering is not `seq_cst`. This information is then used by the ReadsFromSet helper function shown in Figure 3.22 to determine what stores can be read from. In particular, the thread local VCs $\$_F$, $\$_W$ and $\$_R$ are checked against the epoch for each load element in the store element. The global VCs $\mathbb{S}_F$ and $\mathbb{S}_W$ are used to convey SC information across threads. A simple way to view this is that information is conveyed from fence to fence, from fence to load and from store to fence.

Each atomic function will first call into the VC algorithm described in §3.3, as shown by calls to functions of the form $[X]$ that correspond with the inference rules in Figure 3.12. The state used by the VC algorithm has a different representation that makes it easier to compare with other VC algorithms, and is converted to this representation using the interface in Figure 3.21.

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 shown

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

(a) Formal definition.



(b) Pictorial definition.

Figure 3.18: Operational State.

in Figure 3.22, which formalises the consistent reads-from of §3.4.4. The `++` operator represents list concatenation.

Each of the high level operations is followed by δ. This is the context switch operator, whereby T and ss are switched out for one of the active threads in Σ. This shows that atomic operations from multiple threads may be interleaved in a fine grained manner, but two atomic operations may not update the state simultaneously.

To help with understanding the details of these rules, consider an example case, where thread t performs an SC store to location x at epoch e. This is represented as $\texttt{Store}(l, a, mo)$, and matches with the [ATOMIC STORE] rule. The $\rightarrow_{store}$ sub-rule updates Σ according to the [SC ATOMIC STORE] rule, which updates the SC store VC with the current epoch, $\Sigma.\mathbb{S}_W(t) = e$. The [RELEASE STORE] rule of Figure 3.3.3 is then performed. The `Store` is then replaced by a <u>`Store`</u> followed by a context switch. This carries out the [ATOMIC STORE IMPL], which simply creates a new store element and appends it to the back of the store buffer for x. As this is an SC store, all of the store elements in the store buffer that happen before this store have their sc flag set to true.

3.6 Characterising The Model Axiomatically

The instrumentation strategy shown in §3.4, and formalised by the operational model of Section 3.5, is designed with consideration of the sorts of non-SC behaviours that would be feasible to explore with an efficient dynamic analysis tool. However, the intricacy of the operational rules makes it difficult to see, at a high level, which behaviours are allowed versus forbidden by this operational model. To clarify this, the model can be characterised as an axiomatic memory model, showing precisely the behaviours that the operational model allows and how it relates to the C/C++11 memory model.

To compare the behaviours allowable by the operational model and the axiomatic model, it is necessary to *lift* the traces given by the operational model to program executions. This lifting procedure intuitively gives rise to an additional axiom to those of C++11. Because this axiomatic model consists of the C++11 axioms plus an additional axiom, it is strictly stronger than that of C++11. It can be shown that the executions given by lifting the set of traces produced by the operational model exactly match the executions captured by this more restrictive axiomatic model. The following diagram illustrates this:

ATOMIC STATEMENTS:

[ATOMIC LOAD]

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

[ATOMIC STORE]

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

[ATOMIC RMW]

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

[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]

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

[ATOMIC STORE IMPL]

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

[CONTEXT SWITCH]

$$\frac{\begin{array}{c}T' = T[T.P := ss] \\ \Sigma' = \Sigma[ThrState := \Sigma.ThrState[T.t := T']] \qquad T'' \in \Sigma'.ThrState \qquad ss' = T''.P\end{array}}{(\Sigma, \delta; ss, T) \rightarrow (\Sigma', ss', T'')}$$

Figure 3.19: Semantics for atomic statements.

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]

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

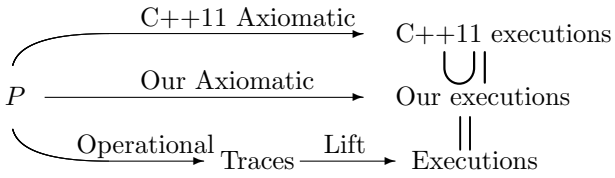
[SC ATOMIC FENCE]

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

[NON-SC ATOMIC]

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

Figure 3.20: Semantics for sequentially consistent fence functions.



Notation Let P denote a program written in the language of Figure 3.17. The set of executions allowable for P according C++11's axiomatic memory model is denoted *consistent*(P). The operational model takes program P and produces a set of traces, denoted *traces*(P). An individual trace is denoted σ, which is a finite sequence of state transitions of the form $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_k$. For a given trace σ, let *lift*(σ) denote the lifting of σ to an axiomatic style execution. For a set of traces S, 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 the operational model.

$$
\begin{aligned}
\textbf{Define:}\ & (A', T') = [X](\Sigma, a, mo, T) \\
\textbf{As:}\ & (\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{X_{mo}(a,T.t)} \\
& (\mathbb{C}', \mathbb{L}', \mathbb{V}', \mathbb{F}^{rel\prime}, \mathbb{F}^{acq\prime}) \\
\textbf{Where:}\ & \\
& \mathbb{C} = \{(T.t, T.\mathbb{C})\} \qquad \mathbb{L} = \{(a, \Sigma.ALocs(a).\mathbb{L})\} \\
& \mathbb{V} = \{(a, \Sigma.ALocs(a).\mathbb{V})\} \\
& \mathbb{F}^{rel} = \{(T.t, T.\mathbb{F}^{rel})\} \qquad \mathbb{F}^{acq} = \{(T.t, T.\mathbb{F}^{acq})\} \\
& T' = T[\mathbb{C} := \mathbb{C}', \mathbb{F}^{rel} := \mathbb{F}^{rel\prime}, \mathbb{F}^{acq} := \mathbb{F}^{acq\prime}] \\
& A' = \Sigma.ALocs(a)[\mathbb{L} := \mathbb{L}', \mathbb{V} := \mathbb{V}'] \\
& X \in \{\text{LOAD}, \text{STORE}, \text{RMW}, \text{FENCE}\}
\end{aligned}
$$

Figure 3.21: Interface from operational model to VC algorithm of Figure 3.12. The [LOAD] operation is slightly different, as a is a store element. For the [FENCE] operation, the Σ and a parameters are omitted, and $\mathbb{C}$, $\mathbb{L}$ and $\mathbb{V}$ will be empty.

```
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 mo ≠ seq_cst ∨ ¬S.sc ∨ ¬FoundSC then SS := SS ∪ {S}
        FoundSC := FoundSC ∨ S.sc
    }
}
```

Figure 3.22: Construction of the reads-from set

3.6.1 Lifting Traces

Before defining the new axiomatic model, it must be made clear how a trace is lifted to an axiomatic program execution. The operational state is first extended with auxiliary labels to track events. A label is defined 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 a *last additional synchronises with* (*lasw*) label, that denotes the last event a forking thread performed before creating the child thread, as *lsb* may have updated before the new thread has begun. Including this information allows for the creation of an execution by inspection of the trace and resulting state. This is presented 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 3.17. 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.

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.

3.6.2 Restricted Axiomatic Model

Now that it is clear how the operational model relates to executions, the behaviours that the operational model exhibits can be easily reasoned about.

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

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

co represents the *commitment order*, it is the order in which events are added to an

execution as a program is running [NMS16]. Assume that there is a partial trace, σ_i and a corresponding partial execution, E_i. When σ_i is advanced to produce σ_{i+1}, possibly adding event e_{i+1} to E_i to produce E_{i+1}, it can be seen from the lift function that there can be no edges of the form $(e_{i+1}, e_{j \leq i})$ in any of the 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 the new 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.

3.6.3 Equivalence of Operational and Axiomatic Models

A sketch of the argument that the set of executions a program P can exhibit under the restricted axiomatic model is equal to the set of executions gained by lifting the set of traces that the operational model can produce for P is now provided. The formal statement of this is:

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

The forward case is shown by induction on the construction of an execution E, similar to the reasoning given for the satisfaction of the consistent SC order in §3.4.3. 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}, the trace σ_i can be extended to σ_{i+1} such that $lift(\sigma_{i+1}) = E_{i+1}$. The backward case is similar, and shows that by extending a partial trace for P that lifts to a partial execution of E, it will always end up as either the same partial execution or a new partial execution.

As an example, consider a trace that is extended with an atomic store to location x. The store buffer for x will have a store element appended to the back corresponding to this new store. The partial execution that corresponds with the initial trace can now have a store event for x added. By the inductive property that the lifted initial trace and the initial partial execution are equivalent, the order of the store buffer for x and the modification order for x match. An mo edge can be inserted between the last store event in mo for x and the new store event. The lifted new trace and the new partial execution will therefore

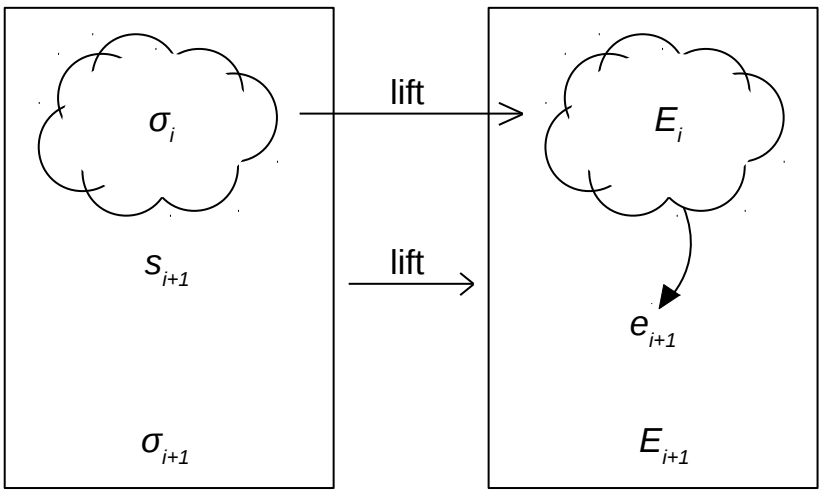


Figure 3.23: Visual representation of the proof of equivalence between program traces and executions.

be equivalent.

The reverse case is also true, so long as the order in which events are added to the partial execution follows the commitment order described in §3.6.3. Therefore, the events of E must first be topologically sorted, which is always possible due to the union of the events of E being acyclic.

A visual representation of this proof is shown in Figure 3.23. The partial execution E_i is extended such that edges do *not* go from the new event back into an event in E_i; any new edges involving the new event are incident on the new event. This is satisfiable because, as mentioned before, the union of all of the events of E is acyclic. New statements are added to the trace in program order, thus following sequenced-before. This is a must, as going against sequenced before would result in a new event in the partial execution with an *sb* edge going backward. Note that s_{i+1} may contain more than one statement, as not all statement create memory events.

3.7 Related Work

There is a large body of work on data race analysis, largely split into dynamic analysis techniques (e.g. [SBN+97, PS03, PS07, EQT10, FF09, ISZ99]) and static approaches (e.g. [EA03, PFH06, VJL07, Ora10, Ste93]). 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 [ND13, ND16], which we study in §5.2.2, uses dynamic partial order reduction [FG05] to reduce state explosion. Cppmem [BOS+11], and an extended version of the Herd memory model simulator [AMT14, BDW16], explore litmus tests written in restricted subsets of C11. Similarly, the Relacey tool supports thorough reasoning about the behaviours of concurrency unit tests, accounting for C++11 memory model semantics [Vyu16]. 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 [KW15, PSN16, LGV16, NMS16, PS16, DV16]. A key work here presents an executable operational semantics for the memory model [NMS16], and we based our notion of *commitment order* on this work. The main difference between our contribution and that of [NMS16] is that the approach of [NMS16] provides complete coverage of the memory model: the operational semantics is provably equivalent to the axiomatic model of [BOS+11]. 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 [AKNT13]. 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 [AKNT13] 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 [VBC+15, MPN13, CV16, PK16], as has the correctness of compilation schemes from C11/C++11 to various architectures [BMO+12, vVZN+13, SMO+12, BOS+11]. 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 [FF10a, CRSB16]. An alternative would be to explore this relation systematically, similar to a recent approach for testing concurrent programs under the TSO memory model [ZKW15], and a method for memory model-aware model checking of concurrent Java programs [JYS12].

The KernelThreadSanitizer (ktsan) tool provides support for fence operations, which are prevalent in the Linux kernel [Goo16], and source code comments indicate that an older version of tsan provided some support for non-SC executions.[1]

3.8 Summary

With the introduction of a formal memory model and weak behaviours in C++11, a clear gap has appeared with regards to dynamic analysis techniques. The work outlined in this chapter introduces techniques that not only cover these issues, but will also scale to large applications as required of dynamic analysis tools.

The traditional vector clock algorithm used for detecting data races has been fixed in §3.3, such that it is aware of the C++11 definition of happens-before. Previously undetected data races involving the misuse of relaxed atomics and RMWs will be detected. Likewise, false positives that would occur due to fence operations will no longer occur.

The machinery for the exploration of weak behaviours covered in §3.4 allows for the exploration of program executions that could not be found under sequential consistency, which is typical of dynamic analysis tools. This machinery is not perfect however, as certain types of behaviours are still not possible. Specifically, those involving load buffering. To back up the accuracy of this machinery, an operational model, and its equivalence to the axiomatic model provided by the C++11 standard is covered on §3.5 and §3.6.

An implementation of the techniques outlined in this chapter is discussed in §5.2, in which the ThreadSanitizer tool has been extended.

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

4 Sparse Record and Replay with Controlled Scheduling

Modern applications include many sources of nondeterminism, due to, for example, concurrency, signals, and system calls that interact with the external environment. Finding and reproducing bugs in the presence of this nondeterminism has been the subject of much prior work in three main areas: (1) controlled concurrency-testing, where a custom scheduler replaces the OS scheduler to find subtle bugs; (2) record and replay, where sources of nondeterminism are captured and logged so that a failing execution can be replayed for debugging purposes; and (3) dynamic analysis for the detection of data races, as described in §3.

Controlled concurrency testing has proven successful in finding subtle bugs in concurrent programs, by exploring a diverse set of schedules (see e.g. [God05, MQB+08, ND13, YNPP12, TDB14]). However, such techniques are known to be limited by the assumption that the thread scheduler is the only source of nondeterminism. For example, in an empirical study of systematic scheduling algorithms, many benchmark programs, such as Apache's httpd, had to be excluded due to their reliance on external factors such as the network [78].

In contrast, record and replay tools aim to capture the external factors that affect the behaviour of a system as the system runs, so that an execution can be faithfully replayed (see e.g. [47, 53, 57, 66]). The degree to which replay is faithful varies, but many systems aim to be extremely thorough by monitoring, intercepting and facilitating replay of virtually all sources of nondeterminism. Unlike controlled concurrency testing, these tools typically leave threads to be scheduled by the regular OS scheduler, recording whatever schedule results. This is fine if a bug happens to be triggered, but does not support systematic or controlled-randomized exploration of thread schedules to find subtle bugs. Faithful record and replay is also difficult from an engineering perspective, often requiring surgical changes to the OS or underlying hardware, and demanding high resource usage due to the many details that must be kept track of.

The aim of this work is to lift controlled concurrency testing so that it can be applied to

larger and more realistic settings, by drawing on ideas from record and replay, sacrificing faithfulness in order to keep overhead low. To do this, a *sparse* approach is taken: relevant sources of nondeterminism affecting an application are assumed to come from (a) the thread scheduler (including the handling of signals), and (b) input from the network, peripherals such as keyboard and mouse, and well-understood system calls that can be configured for particular applications of interest. This chapter presents a record and replay mechanism that captures minimal information about these sources of nondeterminism that suffices to enable efficient controlled concurrency testing for a range of applications. The advantage of this is that execution is efficient and recording overhead is low. The price is that the tool cannot handle systems whose behaviour is influenced significantly by other sources of nondeterminism (e.g. memory layout) without programmer intervention. While this work is distinct from the race detection work of §3, they can mutually benefit each other: the control over the reads-from relation removes nondeterminism due to atomic reads, and the controlled scheduling with record and replay helps to root out bugs and deterministically replay them for the race detection to flag up.

The main research questions considered are: (1) Can an efficient and adaptive scheduler be created, that maximises the expressiveness of the program while still allowing for parallelism? Expressiveness here means the ability for the program to explore any schedule that the semantics of the underlying program permits. (2) Given a set of nondeterministic aspects of a program, can a system be devised that efficiently stores these aspects during program execution and then enforces these stored aspects, and only those that are stored, on the program as a set of constraints during a later execution? (3) To what degree can the stored constraints of (2) be minimised? This work takes a sparse approach to record and replay, thus, is there a minimal set of behaviours that suffices to replay a wide set of applications?

This chapter is structured as follows:

Background on controlled scheduling and record and replay §4.1 §4.2 These first two sections give a short introduction on what is meant by controlled scheduling and record and replay, along with why they are useful techniques to pursue.

Scheduler Protocol §4.3 This section outlines the scheduler, showing how it can be easily adapted to new scheduling strategies, and how it maximises the expressiveness of the underlying program. A description of the protocol through which threads interact with the scheduler is provided.

Sparse record and replay §4.4 Building on the scheduler detailed in §4.3, the record and replay system is outlined. Unlike previous works on record and replay, the system outlined here aims to record as little as possible, thus, an explanation of what needs to be recorded, what *must* be ignored, and the trade-offs of recording some of the more indecisive parts of the program is provided.

A detailed implementation and evaluation of the instrumentation described in this chapter is provided in §5.3. The implementation is provided as an extension to ThreadSanitizer.

4.1 Controlled Scheduling

At the core of every modern operating system is a scheduler tasked with deciding the order in which to run threads on available processors. This decision is usually based on thread priorities and contention for resources, but there can be many additional factors depending on the specific OS. The resulting choice of schedules is often very difficult to make, and will have a large impact on performance; schedulers are thus very complex, heuristically-driven components.

In the context of a program analyser, controlling the scheduling of threads, referred to as *controlled scheduling*, allows for the exploration of interesting and unusual schedules. Such schedules can reveal subtle bugs that the system scheduler would trigger with low probability, and having control over which schedules are explored is important for replay of bug-inducing schedules. This is typically handled by applying a *scheduling strategy*. For example, a *depth-first* strategy would have a single thread run for as long as possible, before switching to another thread.

Scheduling decisions are made at scheduling points, which correspond to visible operations: a visible operation is an operation performed by a thread that may influence the behaviour of other threads.

As an example, consider the program fragment shown in Listing 4.1. A scheduler applying a depth-first strategy could produce the interleavings `ABCDEF`, `CDEABF`, or even `ABFCDEF`, while a random strategy could produce `CAFDEBF`. According to the C11 memory model explained in §3.1, all of these schedules can produce a data race, because if the load of `F` reads from the store of `E` there will be no synchronisation between thread `T1` and thread `T3`. But for the machinery discussed throughout §3.3 to detect the race on *nax*, an ordering in which `E` is the most recent write to x before `F` is required. The number of interleavings grows exponentially with the size of the program, therefore an exhaustive search is not possible, and scheduling strategies must be used instead. Different strategies

```
void T1() {
  nax = 1;
  x.store(1, std::memory_order_release);  // A
  y.store(1, std::memory_order_release);  // B
}
void T2() {
  if (y.load(std::memory_order_relaxed) == 1 &&  // C
      x.load(std::memory_order_relaxed) == 0)    // D
    x.store(2, std::memory_order_relaxed);       // E
}
void T3() {
  if (x.load(std::memory_order_acquire) > 0)  // F
    print(nax);
}
```

Figure 4.1: A racy C++11 program using atomic operations.

are effective at finding different sets of bugs [TDB14].

This work does not delve into the details of various scheduling strategies, instead, scheduling is used to both facilitate record and replay, and demonstrate the variable nature by which different strategies have on program overhead and bug finding ability. Usually the interleaving of operations imposed by the default OS scheduler will be unpredictable, and therefore introduce nondeterminism in the program. By enforcing a particular interleaving on the program, that can be duplicated on separate executions, nondeterminism caused by the OS scheduler will be removed.

4.2 Record and Replay

The ability to record and replay has many useful applications, notably allowing consist reproduction of bugs in nondeterministic programs. In general, recording and replaying involves identifying relevant sources of nondeterminism, and enforcing the same resolution of this nondeterminism during replay as was observed while recording. The granularity at which nondeterminism is controlled varies between approaches. To see how useful record and replay is, consider the example program shown in Figure 4.2. The program receives buffers from a server, processes them, and then sends them back. But what happens if the connection fails or we get a "poll error"? Replaying an execution that shows an error, without having to actually connect to a real server, allows us to reliably explore the cause of the error. This is particularly useful for larger programs that utilise complicated communication protocols, and have time consuming setups or hard to find bugs.

In general, a program can have many such sources of nondeterminism. Aside from the thread interleaving and value of atomic reads, other sources include interaction with the file system, system calls, certain libc functions (e.g. the conditions under which malloc

```
void sig_handler() {
  quit.store(1);
}

void Listener() {
  while (!quit.load()) {
    int res = poll(&server_fds, 1, 100);
    if (res == 0) continue;
    CHECK(res > 0 && server_fds.revents == 1 && "poll error");
    char *buf = new char[100];
    recv(server_fd.fd, buf, 100, 0);
    std::unique_lock<std::mutex> lck (mtx);
    requests.push(buf);
  }
}

void Responder() {
  while (!quit.load()) {
    std::unique_lock<std::mutex> lck (mtx);
    if (requests.size() == 0) continue;
    char *buf = requests.front();
    requests.pop();
    lck.unlock();
    Process(buf);
    send(server_fd.fd, buf, 100, 0);
    delete[] buf;
  }
}
```

Figure 4.2: Generic client for processing and returning requests sent from some server.

can fail are not deterministic), instructions that query the state of the CPU (such as the x86 RDTSC for reading the processor's time-stamp counter), or in some cases even the value of pointers (e.g. iterating through an ordered container of pointers)

The choice of what nondeterminism to record, and the method of recording and replaying, is a substantial area of research. The current state-of-the-art tool is rr [OJF+16], which achieves performance overheads as low as 1.5× native for some applications, as well as low storage overheads. The recording strategy of rr is to record all sources of nondeterminism, including that of the filesystem, scheduler, system calls and memory layout. The scheduling strategy of rr is priority-based first come first served, with each thread given a time slice before yielding; execution is sequentialised so that only one thread runs at a time. A thorough comparison between rr and the work presented in this chapter is provided in §5.3, which details the implementation of this work in tsan.

The majority of the work that covers record and replay, attempts to do so by recording as much of the nondeterminism of the program in question as possible. But this presents a few problems. For example, the entire record and replay system can be thrown off if a small and infrequent source of nondeterminism goes unnoticed, and the overheads involved with such an extensive recording system can become a problem. The work presented here

approaches the problem from the opposite end of the spectrum, by recording *as little as* possible. This will be explained in detail later in §4.4.

4.3 Scheduling Protocol

One major source of nondeterminism within a multi-threaded program is the interleaving of the operations in its threads. For a record and replay system to reliably replay an execution for such a program, it must be able to control the interleaving of the operations of its execution. A scheduler is therefore required, as a prerequisite to the record and replay system. Having control over the scheduler can also be beneficial in other ways. For example, a scheduling strategy can be enforced on the program to help root out bugs [MQB+08].

The scheduler and the corresponding record and replay system will sit in user space. This helps to ensure that the ensuing tool is easy to use, as modifications to the OS will require some effort on the part of the user to set up. It will also allow it to be integrated into the tsan tool outlined in §5.3, along with the race detection machinery outlined in §3. This scheduler is not intrinsically tied to tsan however, and a separate standalone tool could be created instead, but building on top of tsan helps with much of the code that would otherwise have to be rewritten.

Enforcing an interleaving for every operation in a program will result in a significant overhead. But not every operation must be ordered. A *visible* operation is one that can be seen by another thread without the help of another operation [God05]. For example, an atomic write to an atomic location can be read by another thread. An invisible operation, such as a non-atomic write, cannot be seen by another thread, without some form of inter-thread ordering—if it could, it would result in a data race. In this section, a visible operation is extended to include any operation that can introduce nondeterminism into the program, including I/O and certain system calls. This provides the first point: (1) Only the visible operations need to be ordered, invisible operations may be left as is.

Given two visible operations in two different threads, these operations are said to be *independent* if either ordering results in an identical execution. For a scheduler, this means that the two operations do not need to be scheduled in any particular order, and may run concurrently. Determining which sets of operations are independent and ignoring all pairs of operations within said sets is known as *dynamic partial order reduction* (DPOR) [ZKW15]. Determining sets of independent operations is non-trivial however, and could even increase the time overhead associated with a scheduler. This work forgoes DPOR for simplicity, providing the second point: (2) All visible operations will be

sequentialised over a single abstract processor.

For a tool based around finding and explaining bugs, the ability for the scheduler to exhibit bugs is of importance. While state space exploration is not the focus of this work, the tool should be wary of unintentionally hiding bugs. To prevent this, the scheduler should be capable of exploring, with non-zero probability, every possible interleaving of the visible operations. This provides the third point: (3) A context switch must be possible after every indivisible visible operation.

Rather than using an overarching scheduler thread, details of scheduling decisions are stored in a designated piece of shared state. The threads interact indirectly via this shared state using a protocol, to cooperatively determine when they should be scheduled. The protocol has been designed so that new scheduling strategies can be easily added. This section will focus on two strategies: *random*, which at each scheduling point chooses the next thread to schedule at random, using a fixed seed thus provides controlled random scheduling similar to that described in [TDB14]; and *queue*, which schedules threads in a first-come-first-served manner.

4.3.1 Protocol Details

Rather than using an overarching scheduler thread, details of scheduling decisions are stored in a designated piece of shared state. The threads interact indirectly via this shared state using a protocol, to cooperatively determine when they should be scheduled. This protocol has been designed so that new scheduling strategies can be easily added.

The scheduler acts as an instrumentation layer that is called into via a set of functions. As with similar libraries, such as tsan, calls to these functions may be easily added to the executable by the compiler on behalf of the user, removing the need for the user to modify the program.

Scheduling is handled by two core ordering functions, `Wait()` and `Tick()`:

`Wait()` – Block this thread until the scheduler activates it.

`Tick()` – Choose a thread to activate.

A thread enters `Wait()` right before it executes a visible operation. Depending on the scheduling strategy used, the state of the scheduler may need to be updated when `Wait` is called: the queue strategy requires the thread to enqueue itself; the random strategy requires no action. If the thread already happens to be the next thread due for scheduling, `Wait()` returns without blocking. A thread enters `Tick()` once it has completed a visible operation. By executing `Tick()`, the thread applies the scheduling strategy (random or queue) to choose the next thread to be scheduled and update the scheduler state to reflect

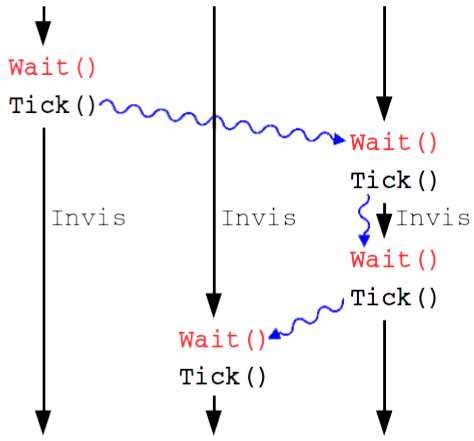


Figure 4.3: Sequentialised critical sections and parallel invisible operations. Blue wavy arrows represent scheduler-imposed ordering; black arrows represent program order.

this. In the case of the queue strategy this involves incrementing the current queue position; for the random strategy this involves choosing the next thread id at random. When the thread returns from the `Tick()` function it is free to continue performing invisible operations unhindered until it reaches the next visible operation, where it will enter `Wait()`. This allows for parallelism between threads, as sections of invisible code are unordered. multiple invisible operations that can execute in parallel are illustrated in Figure 4.3.

The combination of a visible operation and associated scheduling-related code, wrapped in a `Wait()` and `Tick()` pair, is called a *critical section*. The code inside a critical section should be indivisible, or else split into multiple `Wait()` and `Tick()` pairs such that each critical section is. By doing so, the scheduler can interleave visible operations in ways that the program could do uninstrumented, maximising the expressiveness of the scheduler. As an example, consider a mutex lock operation, where the mutex is already acquired. An attempt to acquire the mutex will fail, and so will attempt to acquire the lock again at a later point in time. Each acquire attempt can be considered its own critical section, instead of the mutex lock operation as a whole. This example, and other such cases, are described in more detail in §4.3.2.

4.3.2 Special Cases

The approach described in §4.3.1 of wrapping the visible operation in a critical section works directly for most visible operations. We now discuss a number of operations that require extra attention to detail, mainly because their semantics necessitate specific up-

dates to the scheduler state, or because they cannot be represented as a simple critical section.

Throughout this section, several functions will be introduced as part of the scheduler instrumentation. Functions that begin with `intercept_` will replace the intercepted functions in the compiled program, such that these intercepted functions will be called instead of the intercepted functions. For example, `intercept_mutex_lock` will replace all calls to `mutex_lock`. These intercept functions exist for functions that require more work than a simple scheduler function.

Thread management Thread creation, deletion and joining are all treated as visible operations. This is because they must update the state of the scheduler, and as such will affect the scheduling of threads going forward. To handle this, there will be three scheduler functions: `ThreadNew(tid)`, `ThreadJoin(tid)` and `ThreadDelete()`.

The `ThreadNew` and `ThreadJoin` functions are added during the thread creation and joining primitives. The `ThreadNew(tid)` function, called by the parent of the newly created thread, will take the thread `tid` of the new thread and *enable* it within the scheduler, allowing it to be scheduled. The `ThreadJoin(tid)` function will block itself until thread `tid` has finished, and as such must instead *disable* itself in the scheduler, and also mark itself as waiting on `tid`. Disabling itself within the scheduler is necessary as thread join is a blocking operation, so if it is selected by the scheduler, it would cause deadlock. On completion, a thread calls `ThreadDelete`, which involves (a) enabling the parent thread within the scheduler if it is disabled waiting for this thread to finish, and (b) disabling itself in the scheduler. All three operations are wrapped in a `Wait()` and `Tick()` pair.

Mutexes The mutex operations *trylock*, *lock* and *unlock* are all visible and require instrumentation. Trylock can simply be wrapped in a `Wait()` and `Tick()` pair as for regular visible operations. Unlock is similar to thread deletion in that it must also re-enable threads that were blocked waiting on the mutex, although in this case only one of the blocked threads needs to be re-enabled. The thread to be re-enabled can be chosen at random.

Mutex lock poses an interesting issue in that a thread attempting to acquire a mutex will block if the lock operation fails. To account for this, the mutex lock operation has been intercepted to be as shown in Figure 4.4. This changes it to a trylock loop, where each lock attempt is its own critical section. Note that this is the native trylock, there is no instrumented version. The `MutexLockFail(m)` function is similar to `ThreadJoin`: a thread calling this function disables itself in the scheduler and informs the scheduler that

```
int intercept_mutex_lock(void *m) {
  int res = EBUSY;
  while (res == EBUSY) {
    Wait();
    res = trylock(m); // native trylock
    if (res == EBUSY) {
      MutexLockFail(m);
    }
    Tick();
  }
  return res;
}
```

Figure 4.4: Instrumented mutex lock. The real lock function is called inside a `trylock` loop, where each lock attempt is a separate critical section.

it is waiting on `m`. The thread will then reenter `Wait` on line 4, but as it is disabled, it will block until it is re-enabled and then scheduled to run. It will be re-enabled and then scheduled later when some other thread calls `MutexUnlock(m)` on the same mutex. The `MutexUnlock(m)` function is called when a thread releases a mutex, and will re-enable one thread that is disabled due to waiting on m. The native version of mutex lock is never called, because if a thread blocks while inside a scheduler critical section, the scheduler will deadlock.

There is no `Wait` or `Tick` inside `MutexLockFail(m)` and `MutexUnlock(m)`. Note that it is possible for another thread to acquire the mutex between a thread being re-enabled and it attempting the trylock. This is OK: a thread being pre-empted in such a way is possible in an uninstrumented program, and so the thread will simply block itself again.

Condition variables Condition variables allow control over when certain threads will wake up and try to acquire a mutex. When a thread initially acquires a mutex, it may check a condition required for it to proceed, and if it fails, release the mutex and block itself via the condition variable associated with the failed condition. This thread will only wake up and try to reacquire the mutex when another thread *notifies* it via the same condition variable. Checking the conditional is performed by the conditional wait function; waking up one or all of the waiting threads is performed by the signal and broadcast functions respectively.

The conditional wait accepts a timer, determining the length of time after which the thread unblocks itself. This timer represents a *physical* time. This is in contrast to the scheduler's ticker, which represents a *logical* time. This difference between physical and logical timing means that from the perspective of the scheduler, the conditional's wakeup timer is nondeterministic. Semantically speaking, a thread can wake up from the timer, and acquire the conditional's mutex before another thread can, even if the timer is very

```
void intercept_cond_wait(void *m, bool timed) {
  Wait();
  CondWait(m, timed);
  mutex_unlock();
  MutexUnlock(m);
  Tick();
  intercept_mutex_lock();
}
```

Figure 4.5: Instrumented conditional wait. When the thread has released the mutex and entered the intercepted mutex lock function, it will block waiting to be signalled and reacquire the lock.

long. This is handled by *not* disabling the thread if it calls a conditional wait with a timer. Despite not being disabled when timed, a thread can still *eat* a conditional signal, and so should still mark itself as waiting on the conditional in the scheduler.

For the three conditional functions, there are three corresponding scheduler functions `CondWait(m, t)`, `CondSignal(m)` and `CondBroadcast(m)`. `CondSignal(m)` and `CondBroadcast(m)` simply wake up one thread and all threads waiting on `m` respectively, and are both simply called from inside of a `Wait()` and `Tick()` pair.

Conditional wait is a little more involved, and details are shown in Figure 4.5. Between the `Wait` and the `Tick`, a thread informs the scheduler that it is performing a conditional wait via `CondWait`. This informs the scheduler that the thread is either blocked waiting for a signal, or performing a timed conditional wait, so that while not blocked it can nevertheless eat a signal. The thread then releases the mutex, informing the scheduler via the `MutexUnlock` scheduler function described earlier that this has been done. Finally, the thread enters the intercepted version of `mutex_lock`, described above. Because this starts with a `Wait`, in the case of an untimed conditional wait, the thread will block until it is re-enabled by a conditional signal or broadcast. By using distinct critical sections to separate a thread marking itself as being blocked on a signal, and attempting to reacquire the mutex, it allows for the possibility for another thread to be scheduled in between, possibly acquiring the mutex.

Once a thread has reacquired the mutex, it will typically recheck the condition it was originally waiting on. If it is not satisfied, it will call `cond_wait` again. This is where the risk of deadlock comes in: if it was the only thread signalled and it re-enters `intercept_cond_wait`, it will not signal other threads first, and all threads that are blocked by the conditional will remain blocked. Preserving any potential deadlocks in the underlying program is useful, and so the scheduler does not limit this from happening; it is also careful not to introduce new deadlocks.

Signals A brief description of signals and signal handlers is provided. It should be noted that these are distinct from the signals associated with the conditional wait operations described above. Only *asynchronous* signals need to be considered, as these can be received by processes at any time, and thus contribute an additional source of nondeterminism. This is distinct from *synchronous* signals, e.g. SIGSEGV, which are raised by the thread as and when certain operations are performed. Unlike in the case of e.g. a memory load operation, which has a designated program point that can be intercepted to facilitate interaction with the scheduler, a signal can arrive at *any* time. The standard also specifies a `signal` function, that binds a handler function to a specific signal.

There are several implementation-defined aspects to signals, such as whether a signal handler is re-entrant or if a certain signal is ignored by default. Depending on how a tool handles these may result in introducing new behaviours or restricting some. The implementation of this work, as detailed in §5.3, piggybacks off of tsan's implementation.

Scheduling with signals is handled by simply marking the entrance to the signal handler, and the aforementioned `signal` function, as visible operations. From a scheduling perspective, besides the arrival of a signal, signals are not a problem. Recording and replaying signals is where things become difficult, which is discussed in §4.4.3.

4.3.3 Liveness

While the scheduler strives to ensure that all possible program behaviours can be explored in principle, in practice, depending on the strategy, this can lead to massive slowdowns in particular cases. For example, suppose a thread is scheduled and undertakes a vast number of invisible operations, or calls a sleep function for some duration, before finally issuing a visible operation. If all other threads end up blocked waiting to perform visible operations and the scheduler doesn't give them a chance to run, the performance of the program may be drastically impacted. This can become particularly problematic when dealing with programs that rely on responsiveness, such as real-time applications.

To cope with this, the scheduler will sacrifice expressiveness slightly by allowing it to force a reschedule in such cases. By expressiveness, this means the scheduler's ability to explore any possible schedule. By forcing a *reschedule* after n milliseconds, the probability of exploring a schedule whereby a thread performs two visible operations consecutively separated by more than n milliseconds is greatly reduced. To achieve this, a background thread must be introduced that can periodically query the state of the scheduler by calling `Reschedule()` every n milliseconds, for some given n.

To see the impact this may have on the expressiveness of a program, consider the

```
std::atomic<int> gate(0);
int nax = 0;
void T1() {
  nax = 1;
  std::this_thread::sleep_for(std::chrono::milliseconds(10000));
  gate = 1;
}
void T2() {
  gate = 2;
}
void T3() {
  int gate;
  while ((gate = ::gate) == 0) ;
  if (gate == 1) {
    print nax;
  }
}
```

Figure 4.6: Printing `nax` is semantically possible, and therefore the program contains a data race. But by preserving the liveness of the program, the racy execution will be very unlikely to occur.

program fragment shown in Figure 4.6. It is semantically possible for thread `T1` to finish the sleep, set `gate` to 1, and have thread `T3` read 1 before threads `T2` and `T3` finish. By forcibly rescheduling away from thread `T1` while it is in the sleep, this behaviour is unlikely to occur, and the data race on `nax` will go unnoticed.

In §4.3.2, it is mentioned that the physical timer for conditional variables is nondeterministic according to the scheduler's logical time. The `Reschedule()` function relies on physical time also, and thus introduces nondeterminism into the scheduler.

4.4 Sparse Record and Replay

Record and replay is very broad concept, and can be implemented in a variety of ways. Several key questions arise, for example: what does it mean to "record" an execution and then "replay" its execution later? Ho do you formalise a recording? What makes replaying an execution *valid*? What should be recorded and what should *not* be recorded?

When a program executes, most of the instructions will be carried out in a predictable manner. However, certain visible operations will lead to nondeterminism. Recording an execution therefore means capturing information about these visible operations in a form that can be used to reproduce relevant aspects of the execution during replay. This captured information is called the *demo file*, or *demo* for short. An execution that is replaying a demo is a *replay*, and the replay is *synchronised*, unless something has gone wrong with the replay such that the execution has diverged, in which case it is *desynchronised*.

This leaves the question of what it means for an execution to desynchronise? A demo is

defined as a series of *constraints* arising from the recorded execution, which the replay is required to satisfy. The tool will enforce these constraints on the program during replay, and as long as it can, the replay is deemed to be synchronised. If at any point the tool is unable to enforce a constraint on the program, then the replay has *hard desynchronised*, in which case the tool will abort. An example of this could be that a certain thread is expected to run, but is disabled. In some cases, the replay may abide by the constraints, but appear to diverge from the recorded execution, for example, by producing console output in a different order. This is instead called *soft desynchronisation*. To illustrate an extreme case of this, the empty demo is trivially synchronised for any replay, as there are no constraints that need satisfying, but will lead to soft desynchronisation practically everywhere unless the system under test is highly deterministic.

The record and replay mechanism is built into the scheduler discussed in §4.3. In cases where a nondeterministic choice needs to be made that is unrelated to the scheduling of threads, a PRNG is used, seeded by two calls to `rdtsc()`. Given the same two seeds, the sequence of numbers produced by the PRNG will be the same, thus, the seeds will stored in the demo and used in place of `rdtsc()` upon replay.

It is desirable for a tool to be as easy to use as possible, and for the most part, the instrumentation outlined here is easy to use by avoiding the need for user annotation. However, there are some cases where they are either unavoidable, or it is not currently known how to avoid their use; this is shown in §5.3.6.

4.4.1 Motivating Example

To help lay out the reasoning and technical explanation given in the rest of this section, an example program is provided. The program fragment in Listing 4.2 shows a simple client that repeatedly receives `char` buffers from a server, applies a transformation to the buffers, then sends the buffers back. There are several behaviours present in this program that can affect how the rest of the program behaves, but which of those that needs to be recorded and which do not depends on the properties of the program that needs to be preserved during replay. A pragmatic approach is taken and discussed below.

What to record The obvious case here is the interleaving of threads. This will ensure that the order of operations to the atomic locations `quit` and `mtx`, and the order of the syscalls used throughout will be the same during replay. Other operations are invisible, and thus will not affect other threads or introduce nondeterminism.

System calls that interact with the environment can be seen as inputs to the program,

which in this case determines how many requests to handle and the contents of each request. For example, `poll` informs the program on whether there is data to be read from the server, and thus needs to be recorded, as do the system calls `recv` and `send`.

The signal handler in this example is used to trigger the end of the program. The arrival of the signal is asynchronous, and comes from outside the program. During replay the tool will need to ensure that the same signal arrives at the same point in logical time.

What to ignore The first, and likely most contentious element, is the layout of memory. This will of course depend on the program in question, but in this example, and for many programs in practice, the position of objects in memory will have no effect on the rest of the program. If the request queue was instead an ordered set of `char` pointers, then it would matter, as the pointer values would determine the order in which the requests are iterated through. This is particularly important for programs such as SQLite and SpiderMonkey, covered in §5.3.7

The control flow of the program is very much a defining factor of a program execution, however, said control flow is usually a consequence of other elements, which in this example will be the result of `poll` and the atomic operations.

4.4.2 Interleaving

As explained in §4.4.1, the ordering of visible operations must be preserved during replay. There is no generic way of storing this information however, as each scheduling strategy will require different demo data. To help illustrate this, a description of how the random and queue strategies store their orderings is provided.

To recap the strategies described in §4.3: the random scheduler chooses which thread to allow to run the next visible operation randomly after each visible operation has completed; the queue scheduler is first come first serve for whichever threads attempt to perform a visible operation.

For the random strategy, the entire thread interleaving is encapsulated in the PRNG. Therefore, no information besides the two seeds used for the PRNG is required.

For the queue strategy, the ordering during record will depend on the order in which threads happen to reach `Wait()`. The ordering of threads will therefore depend on physical timing. Because relying on physical timings is not feasible, the ordering will be converted into a logical form, and stored in a file called QUEUE. This file records (a) a map specifying, for each thread id, the first tick at which the thread should be scheduled, and (b) an ordered list of ticks to be consumed by threads each time they leave a critical section; the tick

that a thread consumes on leaving a critical section informs the thread as to the next tick at which it is to be scheduled. Run-length encoding is used to efficiently record the case where a thread is scheduled multiple times in succession.

There is clearly a tradeoff between these strategies. Where the random strategy stores no data, the queue strategy may need to store data on every visible operation. The queue strategy will be much faster however, as a thread is unlikely to be blocked in `Wait()` unless another thread is already critical.

4.4.3 Signals

Signals were mentioned briefly in §4.3.2, but deferred discussion to this section as most of the difficulties with signals are in attempting to replay them. Synchronous signals are ignored (e.g. SIGSEGV, SIGPIPE) as these should reoccur at the same point in the execution without the help of a tool. To clarify, it is *entering the signal handler* that is the visible operation. When inside the signal handler, the thread cannot interact with the rest of the process except through atomic operations, which are themselves visible operations. From this, it is clear that it does not matter at which point between a `Tick()` and following `Wait()` that the signal handler is entered.

Any asynchronous signal that arrived during recording will become a synchronous signal upon replay. When a thread receives a signal, it will record the value of the tick seen during the most recent call of `Tick()`, along with the signal value in a file called SIGNAL. For example, consider the case where the `Responder` thread in Fig. 4.2, T2, has just performed the atomic load on tick 5, but has not yet attempted to acquire the lock. The thread receives the signal and performs a `Wait()` and `Tick()` so that it can enter the signal handler. The SIGNAL file will therefore have the line "`2 5 15`", indicating that thread T2 receives signal 15 at tick 5. During replay, when the `Responder` thread calls `Tick()` during tick 5, it will raise signal 15 itself at the end of `Tick()`. It does not matter at which precise point between `Tick()` and the following `Wait()` that the signal arrived at during recording, it will float to the end of the most recent `Tick()` for that thread as shown in Fig.4.7. Signal handling will be disabled inside of `Tick()`, but the thread will attempt to enter the signal handler as soon as it returns from `Tick()`.

4.4.4 System Calls

As shown in §4.3.2, system calls are a significant source on nondeterminism in an application. To ensure that the relevant properties of an application are preserved during replay, the results of relevant system calls must be recorded. This is a fundamental challenge that

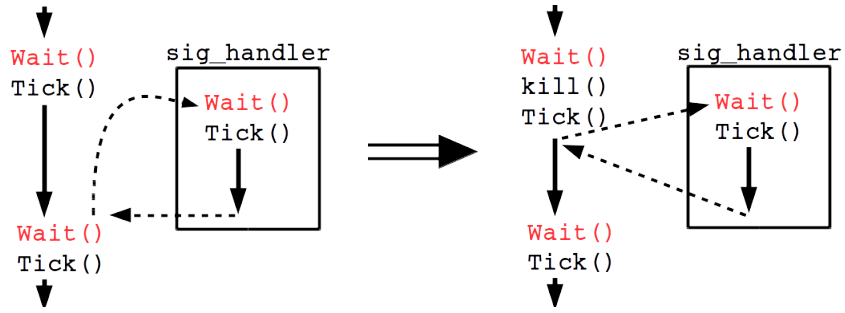


Figure 4.7: Signals are replayed immediately after the preceding tick.

any record-and-replay system must face, and state-of-the-art tools such as rr [OJF+17] aim to be as comprehensive as possible in the system calls they support, so that they can be applied directly to a wide range of applications. In contrast, the idea behind the sparse approach presented here is to identify a minimal subset of system calls such that recording these system calls suffices to enable faithful replay of particular applications of interest.

The high level approach to system call support is to incrementally add support for system calls based on a trial-and-error process. For example, one can first run the program using strace to understand the full set of system calls issued by an application, and then repeatedly attempt to record and replay the application. Determining which additional system calls to support is decided through analysis of the sources of replay failures. The system calls discussed here were identified via this trial-and-error investigative process using the tsan11rec tool and case studies described in §5.3. This process did require quite some manual effort, and would need to be iterated further to handle applications with significantly different system call requirements than those used to develop the current iteration of the tool.

The term "system call" is a bit of a misnomer, as the instrumentation detailed in this chapter and tools such as tsan will instead intercept the glibc wrappers around the system calls, instead of the system call directly. These glibc functions are much easier to use on behalf of the programmer, as they will take care of system specific details, pushing the arguments onto the stack and interpreting the results returned by the kernel. The term "system call" is still used throughout, as it is in the underlying system call where the nondeterministic behaviours occur.

How the system call is executed is beyond the control of the scheduler—while the system call is executed in kernel space, the scheduler sits in user space. Instead, the scheduler must manipulate the information that is passed to and from the system call.

Each syscall takes a variable number of user allocated buffers, the contents of which will be mutated by the system call, before setting `errno` and returning some value. The `errno`

location is provided by the environment and will be set by the system call to indicate any problems that occurred during execution. During recording, the return value, `errno` value, and the contents of any appropriate buffers will be compressed and stored in a file called SYSCALL. During replay, the actual data returned will be overwritten by the data in SYSCALL. Only the interaction with the SYSCALL file is part of the critical section, which reduces contention inside the scheduler.

As an example, consider the `Listener` thread in Figure 4.2 performing the `poll()` and `recv` syscalls in succession. The return value, error number and two elements in the `server_fds` structure must be stored for `poll`; the return value, error number and contents of the buffer must be stored for `recv`. Each of these elements will be treated as character buffers and have a simple run length encoding applied.

One of the difficulties that arises from adding a system call is the *knock-on* effect it can have with respect to other syscalls. Consider, for example, the system calls that interact with the filesystem, `create`, `open`, `read`, etc. If you intercept `open`, on the assumption that you may not have a valid file descriptor (fd) during replay where you did during record, you will then have to intercept every syscall that works with that fd. What starts out as a single interception becomes potentially hundreds of intercepted system calls. There is a delicate balance to be struck between those system calls that need to be recorded to make important execution features deterministic during replay, against those system calls that are better left un-recorded because (a) determinism of replay does not depend on them being recorded, and (b) recording them leads to a snowball effect where many other system calls must also be recorded to avoid desynchronisation.

As mentioned earlier, system call support is determined through a trial and error investigative approach. Based on this, the current set of system calls supported includes `read`, `write`, `recvmsg`, `recv`, `sendmsg`, `accept`, `accept4`, `clock_gettime`, `ioctl`, `select` and `bind`. These have allowed a significant set of applications to be successfully recorded and replayed (modulo a few workarounds detailed in §5.3.6). The applications supported will typically issue many additional system calls that have shown to be unnecessary to record, in the sense that simply re-issuing the system call during replay has no observable effect on the application's behaviour. Sometimes whether a call must be recorded depends on the fds that the call receives. For instance, for all of the applications covered, it never proves necessary to record read and write calls whose fds correspond to files in the file system, but it is necessary to record these calls if the associated fds are associated with pipes used for interprocess communication. Rather than this set of system calls being a starting point towards full system call coverage, the idea is that efficient record and replay that preserves parallelism can benefit from selective system call recording, based on application-specific

```
void Scheduler::SyscallBind(int *ret, int fd, void *addr, uptr addrlen) {
  if (!SyscallIsInputFd(addr, addr_len)) {
    input_fd_[sockfd] = false;
    return;
  }
  input_fd_[fd] = true;
  int replay_fd = fd;
  int errno_ = *__errno_location();
  void *params[2] = {ret, &replay_fd, &errno_};
  uptr param_size[2] = {sizeof(int), sizeof(int), sizeof(int)};
  Wait();
  DemoPlaySyscallNext("bind", 2, params, param_size);
  fd_map_[replay_fd] = fd;
  DemoRecordSyscallNext("bind", 2, params, param_size);
  *__errno_location() = errno_;
  Tick();
}
```

Figure 4.8: Record and replay setup for `bind`.

knowledge, and that a tool supporting a core set of essential system calls is configurable with support for further system calls to suit particular record and replay scenarios. For example, to handle a program such as `htop` would require instrumentation of the interaction with the `/proc` filesystem, but doing this in the general case would be wasteful, and maybe even harmful if future calls depended on this interaction.

Many of the system calls are simple to record and replay, requiring just the user buffers and nothing more. Some of them can become substantially more complicated, in particular, those that handle multiple fds. Because fds are assigned by the kernel, their values are unpredictable, and so any behaviour that depends on multiple fds may also be unpredictable. A detailed implementation and explanation of some of these has been provided. Some parts of the implementation have been simplified or omitted for clarity. In each example, the scheduler instrumentation function is called *after* the actual system call has already been executed.

`bind` (Figure 4.8) The `bind()` system call is used to set up a socket fd to listen for incoming request via some name. This is useful for servers that needs to listen for incoming connections, such as web servers.

The address being bound is first checked to see if it is an *input* address, and therefore is the associated fd is an *input fd*. For system call that handle fds, only those that act upon input fds need to be recorded, as indicated by `input_fd_`. Because the value of the fds will be different during record and replay, a mapping from recorded values to the current values is maintained in `fd_map_`.

```
// Identify input fds and add poll buffers to be recorded.
__sanitizer_pollfd *poll_fds = (__sanitizer_pollfd *)fds;
uptr icount = 0;
void *params[64] = {ret};
uptr param_size[64] = {sizeof(int)};
for (uptr p = 0; p < nfds; ++p) {
  if (!input_fd_[poll_fds[p].fd]) {
    continue;
  }
  params[2 * icount + 1] = &poll_fds[p].events;
  param_size[2 * icount + 1] = sizeof(poll_fds[p].events);
  params[2 * icount + 2] = &poll_fds[p].revents;
  param_size[2 * icount + 2] = sizeof(poll_fds[p].revents);
  ++icount;
}
// Attach error number to end of buffers. Quit if there were no input fds.
int errno_ = *__errno_location();
params[2 * icount + 1] = &errno_;
param_size[2 * icount + 1] = sizeof(int);
if (icount == 0) {
  *__errno_location() = errno_;
  return;
}
// Critical section.
Wait();
DemoPlaySyscallNext("poll", 2 * icount + 2, params, param_size);
DemoRecordSyscallNext("poll", 2 * icount + 2, params, param_size);
*__errno_location() = errno_;
Tick();
```

Figure 4.9: Record and replay setup for `poll`.

`poll` **(Figure 4.9)** For a program with multiple fds that represent a variety of different connections, the ability to check each of them for pending information is crucial. With `poll()`, a program can check as many fds as necessary, and even block waiting for any of them to receive data. Each fd will have an associated `pollfd` struct, which is used by the kernel to indicate to the program the status of the fd. Within each `pollfd` struct, only a couple of fields are used. These are extracted for recording, so long as the associated fd is an input fd. There are some problems however, for example, what happens if there is a mix of input and non-input fds? If the function was blocking waiting on a non-input fd, should the function return immediately during replay? A general solution to many of these system calls is not trivial, and one solution may not be acceptable in the general case.

4.4.5 Asynchronous Events

Asynchronous events are specific events that do not fit in with any of the other categories discussed. An important characteristic is that they are *not* wrapped in a `Wait()` and `Tick()`, either because it was infeasible to do so during recording, or because it would

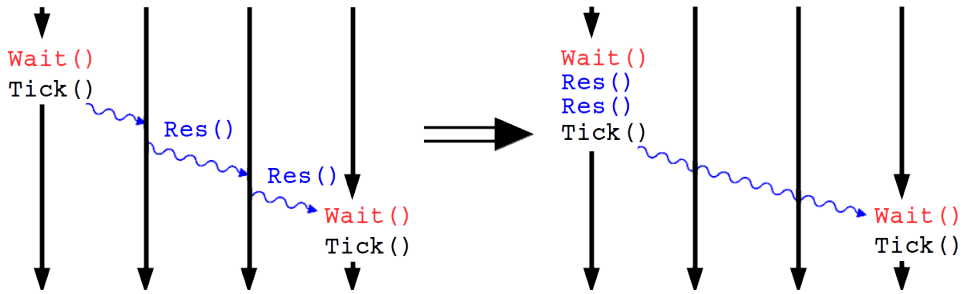


Figure 4.10: Right diagram shows how reschedules on the left are floated above the preceding `Tick()`.

create a lot of unnecessary overhead. These events still need to be replayed to ensure the replay remains synchronised. Currently there are two types of asynchronous events: the `Reschedule` event and the `Signal_wakeup` event.

The reschedule event was discussed in §4.3.3. It is necessary to include this to ensure that the PRNG will be called the same number of times in each critical section. To see why the signal wakeup event is necessary, consider again a signal being received in the context of the example of Figure 4.2. This time, suppose the `Responder` thread receives the signal while it is disabled trying to acquire the lock. Assume that the thread disabled itself on tick 10, the signal arrives and the thread re-enables itself during tick 12, and then enters the signal handler on tick 14. It is *not* OK for the thread to simply not disable itself on tick 10 during replay, as the pool of enabled threads for the scheduler to choose from during ticks 10 and 11 is different between recording and replaying, which will affect the choice the scheduler will make.

As with signals, all asynchronous events are replayed synchronously, with all events that occur between a `Tick()` and the following `Wait()` floating up to the previous `Tick()`. This is shown in Fig. 4.10. These events are stored in the ASYNC file.

4.5 Related Work

Controlled Scheduling A large amount of work has gone into the use of scheduling strategies as a form of state space exploration (e.g. [MQB+08, YNPP12, FF10b, MQ07, TDB14, FF10a]) and on techniques aimed at reducing the size of the state space, such as dynamic partial-order reduction [FG05, ZKW15]). A particularly notable controlled scheduling tool, in terms of successful practical application, is Microsoft's CHESS [MQB+08], which aims to systematically explore all interleavings of a test scenario. Similar to our approach, each visible instruction has an associated custom *wrapper* that intercepts the

real instruction, calling into the CHESS scheduler. Interception is dynamic: the program in test does not need to be modified to call the wrapper functions. The test scenario is wrapped in a callback and given to the runtime library, allowing it to repeatedly run the scenario and record information on the paths it takes. While CHESS is able to work directly on binaries, circumventing many of the issues with setup that many tools face, it is currently limited to unit tests.

Schedule bounding techniques, notably preemption- and delay-bounding [EQR11, MQ07], have been shown to be successful in prioritising the order in which thread schedules are explored during controlled concurrency testing. They prioritise exploring schedules that exhibit small numbers of preemptions between threads, in line with empirical evidence that bugs rarely require large numbers of preemptions in order to manifest [LPSZ08]. Combining such techniques with the tsan11rec algorithm is an appealing idea in principle, but is hindered by the assumption that the program under test takes a fixed input and that the scheduler is the only source of nondeterminism. This assumption allows running the program again and again trying different schedules. In the context of tsan11rec, which can be used to record and replay applications where the environment presents other forms of nondeterminism, the manner in which the program interacts with its environment is captured with respect to a particular thread schedule, and other thread schedules might involve completely different environmental interactions. A more reasonable approach would be to bring ideas from the probabilistic concurrency testing (PCT) algorithm [BKMN10] to the tsan11rec setting, to introduce a degree of skewing to the random strategy so that it explores more diverse schedules.

Record and Replay Record and replay has been a significant area of research, with many tools being created to facilitate it [DCD+14, MGT+17, HZD13, VLW+11, HCH17, LSW+18, HLZ10, LZTZ15, AS09, LWV+10, DKC+02, OJF+17, LVN10, TLH+07, BHCG10, Sai05, GASS06, GWT+08, MGT+17, OJF+16, HT14, PPS+10]. The general premise behind them is similar: identify *order nondeterminism* and *input nondeterminism*, and create techniques to capture them while recording and control them during replay.

Various tools extend the OS in some way or require specific hardware [DCD+14, VLW+11, AS09, LWV+10, DKC+02, LVN10, TLH+07, BHCG10, BG91, DLCO09]. This has the benefit of giving the tool access to much more of the system, such as memory pages and process information. For example, Scribe [LVN10] will directly modify the system scheduler, instead of coercing it, and achieves slowdowns as low as 1.05×. However, this severely hits the usability of the tool, as it requires the user to deploy a modified OS.

Other tools reside entirely in user space [MGT+17, HZD13, HCH17, LSW+18, HLZ10,

LZTZ15, OJF+17, Cha99, Sai05, GASS06, PPS+10, BCdJ+06, Und18, Got, LCFN12, WPP+14, LKZ14]. This is the category that tsan11rec falls into, trading performance for usability. Ease of use is particularly important in persuading users to adopt the tool, rr [OJF+17] in particular allows the user to record a program by simply passing the binary to rr as a parameter, and as such has become the definitive tool for record and replay. We have performed an extensive comparison with rr in §5.3, and note that while rr outclasses tsan11rec in some applications, and can handle applications that are out of scope for tsan11rec (see §5.3.7), rr shows significantly higher overhead compared with tsan11rec for a number of applications that rely on a high degree of parallelism for performance. Further, the sparse approach, with suitable workarounds, enables record and replay for graphical applications, for example, the SDL-based games of §5.3.6, that rr cannot currently handle.

Because it builds on tsan11, which itself uses compiler instrumentation and a modified libcxx, tsan11rec shares similarities with tools that depend on language implementation or library-level support [ACN+01, Mic18, Dev18, BBKE13, GWT+08, BBKE13]. Notable examples here include R2 [GWT+08], which requires manual annotation, and IntelliTrace [Mic18], which is built into the developer environment.

Whole system replay aims to record *all* system nondeterminism [DCD+14, DKC+02, LSW+18, DGHH+15, SJ12, EAW10, DLFC08, BJH+16]. Among these, the recent iReplayer tool [LSW+18] performs record and replay *in-situ*, avoiding many of the problems (e.g. memory layout issues) that otherwise come from running the record and replay executions under different processes.

Some tools focus on the order-nondeterminism, allowing them to retain their parallelism and thus reducing the overhead of multi-threaded application [DLFC08, XBH03, NPC05, HH08, MCT08, PDP+13]. Castor [MGT+17] will provide each thread with its own buffer for storing information, and serialize them at a later time. tsan11rec also fits into this category, as it both preserves parallelism of invisible operations and applies a scheduling strategy to resolve this nondeterminism.

An alternative to recording a program's nondeterminism is to *remove* it, making some or all aspects of the program deterministic [BAD+10, AWHF10, LCB11, CWG+11, CSL+13, DNB+11]. For example, Dthreads [LCB11] ensures that memory accesses are deterministic on each execution. Such approaches can have a significant *probe* problem by removing the behaviour necessary for certain bugs to manifest, in return for avoiding the performance overhead associated with handling order-nondeterminism.

Multi-version (or multi-variant) execution (MVE) is a method for concurrently running multiple processes that are expected to behave in a semantically similar manner [HC15, KBG16, PAC18, VCS+17, VCV+16]. MVE can be used to detect security vulnerabilities

in applications: if a variant diverges, this could indicate that an attacker has modified the process in some way [KBG16, VCS+17, VCV+16]. It can also be used for running different analyses on identical processes, that would not work when run together on the same process, such as the clang sanitizers [PAC18]. Most MVE systems hinge on a special monitor thread that controls the generation an maintenance of a number of variants. Keeping the variants in sync with respect to nondeterministic behaviours presents many of the same problems that are associated with record and replay.

4.6 Summary

From the work shown in chapter 3, two issues became apparent: that the detection of data races was at the mercy of the system scheduler, and that without diagnostics, the root cause of the data races is difficult to pinpoint. The work shown in this chapter aims to remedy these issues.

A method of scheduling has been created with the following strengths: (1) Parallelism is preserved for sections of code that cannot interfere with other threads. (2) Different scheduling strategies can be easily implemented. (3) The expressiveness of the program under test is not diminished. As previous work has shown that different strategies can help find different kinds of bugs [TDB14], the introduction of an easily modifiable scheduler lays the groundwork for a dynamic analysis tool that can find a wide variety of bugs.

A record and replay system has been introduced that builds upon the scheduler. The system focusses on *sparseness*, that is, recording as little as possible. This has the benefits of keeping overheads low, and reducing the *knock-on* effect whereby recording one element forces you to record even more elements that depend on the previous element.

An implementation of the techniques outlined in this chapter is discussed in chapter 5.3, in which the ThreadSanitizer tool has been extended.

5 Extending ThreadSanitizer

The works shown in chapters 3 and 4 have been described largely independently of any implementation. This has kept them neutral with regards to any particular tool. The upside is that each technique has been explained without the baggage that a tool would typically bring. It also allows them to be more easily reasoned about and potentially built upon than if they had been described with respect to a particular implementation. Nevertheless, an implementation of some sort is crucial to see how well these techniques perform in practice.

The tool of choice, for both pieces of work, is ThreadSanitizer (tsan), which is described in detail in §5.1. The choice to extend tsan was made for several reasons: tsan scales well to large applications, tsan is open source and maintained as a part of LLVM, and tsan already provides instrumentation hooks for a rich set of primitives and system calls, so that building on it saves a lot of effort. It also comes with a basic vector clock implementation, which can easily be extended to accommodate the work of chapter 3.

The rest of this chapter is structured as follows: section 5.2 focusses on extending the standard tsan with the dynamic race detection methods described in §3.3 and §3.4 to produce tsan11. Section 5.3 discusses further extending tsan11 with the scheduling and record and replay methods described in §4.3 and §4.4 to produce tsan11rec. There is no related work section as relevant related work is already covered in sections 3.7 and 4.5.

Both implementation sections address common questions pertaining to the feasibility of each technique, focusing especially on time and space overheads and the ability to detect races. Both §5.2 and 5.3 aim to answer these questions by providing a detailed experimental evaluation over several applications.

5.1 ThreadSanitizer

ThreadSanitizer (tsan) is an efficient dynamic race detector tool aimed at C++ programs [SI09]. The tool originally targeted C++03 programs using platform-specific libraries for threading and concurrency, such as pthreads. Tsan does support C++11 atomic operations, but does not fully capture the semantics of the C++11 memory model when

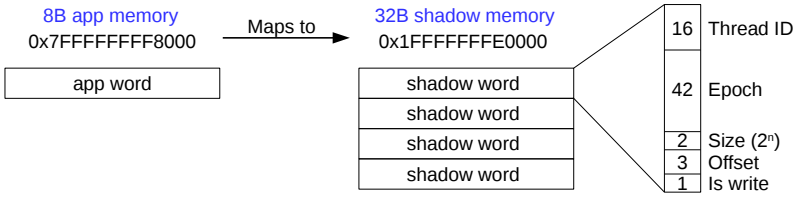


Figure 5.1: Each 8 bytes of application memory maps to 32 bytes of shadow memory, which contains 4 shadow words. Each shadow word stores information on a single access. The information stored in the shadow word is enough to determine if two accesses form a data race.

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.

Tsan performs compile-time instrumentation of the source program, in which all (atomic and non-atomic) accesses to potentially shared locations, libc functions and syscalls are instrumented with calls into a statically linked run-time library. For example, the following shows how the compiler instruments non-atomic accesses on x86:

```
                          push nax
nax = 1;   ──────→        call __tsan_write4
                          add esp, 8
                          mov [nax], 1
```

This will cause the compiled program to call `__tsan_write4(void *)`, a tsan instrumentation function that tracks non-atomic writes, just before performing the actual write.

The tsan library implements the Vector Clock (VC) algorithm outlined in §3.2. 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. Each shadow word is laid out as shown in Figure 5.1.

For glibc functions and syscalls, tsan will intercept each function to call into a specific function within the tsan library. For example, the `recv` glibc function will call into `in-`

`terceptor_recv` instead. This allows tsan to update its state, and will usually also call into the real function that would have been called had the function not been intercepted.

The tsan tool is part of the `compiler-rt` LLVM project,[1] and both tsan11 and tsan11rec are extensions to SVN revision 272792.

Limitations of tsan Recall from §3.1 that under certain conditions, a release sequence can be blocked according to the C++11 memory model. 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 3.4 in Chapter 3. 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 3.11c in chapter 3 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 3.5), tsan will not detect the race even if the instrumented program is executed on a non-SC architecture, such as x86, POWER or ARM.

As tsan is a dynamic analysis tool, it is only aware of the properties of the program during the current execution. For a data race to be detected, it must manifest itself. If a data race relies on an unusual or improbable execution, it will most likely be missed. Furthermore, tsan's diagnostic capabilities are limited, simply providing the two stack traces from the two threads at the point of the data race. This means that while tsan may be good at finding a race, it probably won't tell you why it happened.

5.2 Implementation of C++11 Data Race Detection

This section details the extension of tsan with the techniques outlined in §3. The extension to tsan is called tsan11. An evaluation of the effectiveness of tsan11 in practise is included, guided by the following research questions: **(1)** To what extent is tsan11 capable of finding known relaxed memory defects in moderate-sized benchmarks, and how does the tool

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

compare with existing state-of-the-art in this regard? **(2)** 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? **(3)** 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?

Details of the tsan11 extension of tsan are provided in §5.2.1. In §5.2.2, (1) is addressed by applying tsan11, the original tsan tool and CDSChecker to a set of benchmarks that were used in a previous evaluation of CDSChecker [ND13]. In §5.2.3, (2) and (3) are considered via analysis of the Firefox and Chromium web browsers.

5.2.1 The tsan11 Tool

The goal of this work is to apply efficient C++11-aware race detection to large programs. The enhanced VC algorithm of §3.3 and the instrumentation library described in §3.4 and formalised in §3.5 have been implemented as an extension to tsan. The original tsan tool supports concurrent C++ programs and provides instrumentation for C++11 atomic operations, but, as illustrated in §5.1, does not handle these atomic operations properly. Throughout this section, the original tsan is referred to as **tsan03**, due to the previous C++ standard before C++11 being C++**03**, and the extended version of tsan, that captures a large part of the C++**11** memory model, as **tsan11**.

The implementation details of tsan11 have been largely omitted, as they are fairly trivial. The enhanced VC algorithm builds on top of the original VC machinery by adjusting how and when a VC is modified, and the store buffering simply attaches a list of store *elements* to each atomic location. A few details and limitations will be clarified.

Bounding of store and load buffers In §3.4.1 and §3.5.2, a store buffer was introduced to facilitate reading from an earlier store to a location than the most recent. However, for each atomic location, there can potentially be an unlimited number of store elements. Consider a thread that repeatedly performs an atomic store in a loop, and another thread that performs an atomic load to the same location without synchronising with the storing thread. This load can read from any of these stores. To prevent unbounded memory overhead, the size of the store buffer must be bounded such that the oldest element of a full buffer is evicted when a new element is pushed. This restricts the stores that loads can read from, so the buffer size trades memory overhead for observable behaviours. This evaluation uses 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 The instrumentation controls the reads-from relation via the the algorithm of Figure 3.22, allowing for variety of randomised and systematic strategies for weak behaviour exploration. The implementation favours reading from older stores, choosing the oldest feasible store with 50% probability, the second-oldest with 25% probability, and so on.

5.2.2 Evaluation Using Benchmark Programs

Benchmark programs To test the effectiveness of tsan11 on small benchmarks, a comparison was performed between tsan11, tsan03 and CDSChecker [ND13], another data race detection tool discussed below. To compare tsan11 with tsan03 and CDSChecker at a fine-grained level, each tool was applied to the benchmarks used to evaluate CDSChecker previously [ND13]. These are small C11 programs ranging from 70 LOC to over 150 LOC. These benchmarks had to be converted from C11 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 §3.6, 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. Therefore, data races were introduced 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 the experiments [LD17a].

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 the 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 were not supported by main-

stream compilers such as GCC and Clang[2] (C11 threads have since become supported by GCC and Clang). In contrast, tsan11 automatically instruments all memory operations, and supports C++11 concurrency primitives. Nevertheless, a best effort comparison is provided as CDSChecker is the most mature tool for analysis of C11 programs.

Experimental setup These experiments were run on an Intel i7-4770 8x3.40GHz with 16GB memory running Ubuntu 14.04 LTS. A short sleep statement has been added to the start of each thread in each benchmark in order to induce some variability in the schedules explored by the tsan tools. The Linux `time` command is used 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 is reported. In §5.3.3, controlled scheduling is used instead of sleep statements, giving a more accurate representation of wall-clock time on each benchmark. The tsan-instrumented benchmarks were compiled using Clang v3.9. The revision of CDSChecker used has the hash 88fb552.[3]

The results of the experiments are summarised in Table 5.1, where all times are in ms, and discussed below. For each benchmark, 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) is reported. For tsan11, the rate at which data races are detected, i.e. the proportion 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%**) is reported. 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. The configuration of CDSChecker flags used is what is recommended in the CDSChecker documentation for all benchmarks. For tsan11, the default system scheduler and the store buffer bound and reads-from strategy discussed in §5.2.1 is used.

[2] 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`.

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

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 the measured time is for the *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 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.

	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 5.1: Comparison of CDSChecker, tsan11 and tsan03; all times reported are in ms.

5.2.3 Evaluation Using Large Applications

Applications The programs being tested 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 8x2.60 GHz CPU with 32GB memory running Ubuntu 14.04 LTS, revision r298600 of Firefox[4] and the Chromium version tagged "tags/54.0.2840.71".[5] The browsers were compiled using Clang v3.9, following instructions for instrumenting each browser with tsan as provided by the developers of Firefox[6] and Chromium.[7] The browsers were run in a Docker container (using Docker v1.12.3, build 6b644ec) via ssh with X-forwarding.

Both browsers were tested with tsan03 and tsan11, and without instrumentation. For brevity, 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 the evaluation as reproducible as possible, the browsers were tested using JSBench v2013.1 [RGEV11].[8] JSBench runs a series of JavaScript benchmarks, sampled from real-world applications, presenting runtime data averaged over 23 runs. The peak memory usage was recorded via the Linux `time` command, reporting the "Maximum resident set size" data that this command records. For the browser versions instrumented with race analysis, all details of reported data races are recorded in a file. In the case of tsan11, during analysis, data on the number and kinds of atomic operations, including their memory orders, that are issued during execution are also recorded.

Results Table 5.2 shows results on memory usage, execution time and races reported running the 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.

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

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

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

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

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

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 5.2: Memory usage, runtime and number of races reported for the browser configurations running on JSBench.

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, the tsan11 instrumentation leads to significantly higher memory consumption. Performance-wise, the tsan11 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. It is not yet known whether the higher rate of races detected using tsan11 for both browsers is due to the additional behaviours that the tsan11 instrumentation exposes, or simply a result of the tsan11 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; a deeper investigation of this (requiring significant novel research) is left for future work.

When running FF11 and CR11 on JSBench, the number of each type of atomic operation that tsan11 intercepted was recorded. The full data is provided online, but the results are summarised in Table 5.3. 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. The table also shows the proportion of operations associated with each operation type—

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

load, store, RMW and fence. This indicates that fence operations were so scarce that 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. The 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.

There is currently no data on the distribution of executed atomic operations throughout the browser source code, nor the typical use cases for these operations; 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: the experiments presented 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 the evaluation does not settle is the question of which aspects of the 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.

5.3 Implementation of Controlled Scheduling With Record and Replay

This section details the extension of tsan11 with the techniques outlined in §4. Note that this extends the tsan11 shown in §5.2.1, not the basic tsan. This further extension of tsan11 is called tsan11rec. Like with tsan11, a number of research questions arise: **(1)** Do the controlled scheduling techniques, in this case random and queue, help or hinder the tool's ability to detect data races? **(2)** What is the runtime overhead with respect to either scheduling strategy, with and without record and replay? **(3)** What class of programs does the record and replay work on? **(4)** For programs that cannot be recorded "out of the box", how much effort is required to get them to work, if possible? **(5)** How reliable is the record and replay with regards to avoiding desynchronisation?

Details of the tsan11rec extension of tsan11 are provided in §5.3.1. (1) and (2) is covered throughout §5.3.3, §5.3.4 and §5.3.5. In §5.3.6, tsan11rec is applied to several videogames, and so covers all five research questions. These videogames make extensive use of I/O, thus several considerations must be made. (4) was covered extensively in §4.4.4, but is also touched on in §5.3.4 and §5.3.6, as neither Apache's httpd or the videogames could be played without modification to either tsan11rec or the program in question. In §5.3.7, several program that cannot be recorded are detailed, thus covering (3) and (5).

5.3.1 The tsan11rec Tool

The tsan11rec tool builds upon the tsan11 tool, instead of the basic tsan tool. This allows the scheduling and record and replay to work in tandem with the data race detection provided by tsan11. It also allows us to control the reads-from relation, removing the nondeterminism that comes with atomic reads.

The scheduling system and the record and replay system are built as a single system, due to the replay systems heavy reliance on controlling the scheduler. The scheduler can be thought of as an extra layer of instrumentation, on top of tsan's layer, due to tsan itself calling into parts of the scheduler. For example, if the user program performs an atomic load, it is not the user program that calls the `Wait()` and `Tick()`, but tsan's instrumentation code for atomic load. This simplifies the implementation of the scheduler, as it will mostly just interact with tsan.

Throughout §5.3, a description of how the tool evolved with respect to the sparse recording facilities of tsan11rec, and many practical challenges faced along the way is provided; these challenges, which seem fundamental to record and replay, are typically described

```
int socket = get_socket();
char buf[128];
int res = recv(socket, buf, 128, 0);
process(res, buf);
```

(a) User code fragment.

```
// Update tsan state before.
size_t res = REAL(recv)(fd, buf, len, flags);
scheduler.SyscallRecv(&res, fd, buf, len, flags);
// Update tsan state after.
return res;
```

(b) tsan instrumentation fragment code in `__interceptor_recv`.

```
void *params[3] = {ret, buf, &errno_};
uptr param_size[3] = {sizeof(size_t), len, sizeof(int)};
Wait();
DemoPlaySyscallNext("recv", 3, params, param_size);
DemoRecordSyscallNext("recv", 3, params, param_size);
Tick();
```

(c) Scheduler code fragment in `Scheduler::SyscallRecv`.

Figure 5.2: Interception for syscall `recv`. The user call to `recv` will instead call `__interceptor_recv`, which in turn will call the scheduler function for `recv` after the real syscall.

only briefly if at all in the literature. It is intended that the exposition provided will be valuable to other researchers.

Syscall interception As shown in §4.4.4, many syscalls introduce nondeterminism, and so require recording. The basic tsan instrumentation already intercepts the majority of syscalls, which can be further built upon. Most of these will be simple in that they call the real syscall function, and then call a dedicated scheduler function for that syscall. Figure 5.2 shows the interception for `recv`, one of the more simple syscalls to record.

Some syscalls require the real syscall to be called in a more controlled way. For example, `read` and `write` may be used by multiple threads within a process to communicate through a *pipe*. In such a case, instead of recording the contents of the data sent, the calls must be ordered, and so called in between a `Wait()` and `Tick()`. The `select` syscall, covered in §5.3.6, utilises fds extensively, and needs careful consideration.

5.3.2 Experimental Overview

To evaluate the controlled scheduling abilities of tsan11rec, the strategies were compared on the CDSchecker benchmark suite (§5.3.3). Larger applications are used to compare

tsan11rec with rr [OJF+17], a state-of-the-art record and replay tool with similar aims to that of tsan11rec, and one of the only such tools to have been made publicly available. The programs used involve real challenges related to networking, signals, I/O and real-time constraints: Apache's httpd web server (§5.3.4), the PARSEC benchmarks and pbzip (§5.3.5), and two first-person shooter games built on the SDL library (§5.3.6). The SDL case studies showcase applications that tsan11rec can handle that are out of scope for rr, due to communication between the game and the OpenGL interface. In contrast, this section also discusses the practical limitations of tsan11rec that rr does not face related to the SQLite database application and Firefox's SpiderMonkey (§5.3.7).

Common experimental setup All experiments were run under Ubuntu 14.04 LTS on an Intel i7-4770 8x3.40GHz platform with 16GB RAM. The tsan11 and tsan11rec tools were built on top of clang revision 286384. The version of rr used is 5.1.0. As a key goal of this work is to apply race detection to record and replay with controlled concurrency testing, most of the testing is done with race detection enabled, even when using rr. Times are still given for rr without race detection for reference. For each result, *native*, *rr*, *tsan11*, *tsan11+rr* and *tsan11rec* to refer to a program running without instrumentation, under rr, with tsan11 instrumentation, under rr with tsan11 instrumentation, and under tsan11rec, respectively.

5.3.3 CDSchecker Litmus Tests

Overview The small programs (roughly 100 LOC each) used in prior work to evaluate CDSchecker [ND13], and used to compare tsan11 with tsan03 and CDSChecker in §5.2.2, are useful to assess whether tsan11rec's controlled scheduling improves on tsan11's ability to find races (including races related to weak memory). As these programs are closed, the scheduler and memory model are the sources of nondeterminism.

Experimental setup The experiments are run in the following four modes: *tsan11*, where tsan11 (which does not use controlled concurrency testing) finds races; *tsan11 + rr*, where tsan11 finds races with rr recording; and *tsan11rec rnd* and *tsan11rec queue*, where tsan11rec finds races using the random and queue strategies, respectively. The results given show the runtime of each tool on each benchmark, averaged over 1000 runs, reporting the standard deviation and remarking on the coefficient of variation (CV)—the ratio of the standard deviation to the mean.

Results and discussion Table 5.4 summarises the results. The Time columns show mean execution times, with standard deviation. Because these are short-running tests, whose behaviour depends intimately on the manner in which threads interleave, the variance across runs is fairly high, with the CV usually exceeding 1, with the exception of the longer running results for rr, for which the CV is always less than 1 and usually less than 0.5. Rate columns show the percentage of all executions that exposed a data race.

Comparing the tsan11rec rnd results with the tsan11 and tsan11rec queue results shows that randomized controlled scheduling allows tsan11rec to find more races across all benchmarks, except chase-lev-deque and dekker-fences. This is because tsan11 runs at the mercy of the OS scheduler, which tends to explore similar schedules on repeated runs, and in these small programs typically causes the main thread to run to completion before other threads are scheduled. The price for this is higher runtime, e.g. mcs-lock and ms-queue suffer slow-downs of around 2× compared with tsan11; this can be attributed to the total ordering of visible operations imposed by tsan11rec. The rr results show huge increases due to a constant overhead applied to all programs. But as rr is designed for larger applications, this overhead will usually become insignificant in more realistic examples.

An examination of a trace from chase-lev-deque shows why tsan11rec rnd detects fewer races than tsan11. From the creation of thread 2 to the point of the race, thread 1 must perform 29 operations before thread 2 performs just 4 operations in order for the race to manifest itself. The probability of this happening under uniform random scheduling is very low. A race report can be coerced from the program by moving the creation of thread 2 to later in the program. This shows that different scheduling strategies will affect how effective we are at finding data races, and that probabilistic concurrency testing (PCT) can be effective at prying out concurrency bugs.

Test	tsan11 + rr		tsan11		tsan11rec rnd		tsan11rec queue	
	Time	**Rate**	**Time**	**Rate**	**Time**	**Rate**	**Time**	**Rate**
barrier	590 (14.45)	0.0%	8 (9.14)	0.0%	4 (5.23)	37.5%	6 (4.99)	0.0%
chase-lev-deque	579 (166.69)	0.5%	0 (1.94)	5.9%	1 (3.07)	0.2%	2 (3.78)	0.0%
dekker-fences	1776 (1208.54)	49.9%	2 (4.02)	50.3%	4 (4.83)	38.7%	3 (4.46)	52.8%
linuxrwlocks	579 (126.45)	0.3%	2 (4.18)	0.1%	4 (4.93)	62.4%	3 (4.50)	0.0%
mcs-lock	574 (13.71)	0.0%	3 (4.45)	0.0%	5 (5.23)	77.0%	3 (4.47)	0.1%
mpmc-queue	574 (13.71)	0.0%	3 (4.49)	0.0%	5 (5.00)	60.5%	3 (4.46)	0.0%
ms-queue	3093 (100.19)	100.0%	91 (64.13)	100.0%	93 (80.68)	100.0%	52 (62.60)	100.0%

Table 5.4: Comparison over CDSchecker benchmarks between tsan + rr, tsan11 and tsan11rec with controlled random and queue scheduling. Each benchmark was executed 1000 times in each mode. The "Time" columns shows the mean execution time (ms) and standard deviation (in parentheses). The "Rate" column shows the proportion of runs that exhibited data races.

5.3.4 httpd

Overview Apache's httpd [Apa18] is a widely-used modular http server that makes heavy use of concurrency to handle many simultaneous connections. For record and replay it is of further interest due to its dependence on external network input. tsan11rec is able to handle httpd by capturing the system calls described in §4.4.4, with one workaround: the accept system call, which listens for incoming connections, relies on epoll_wait to listen for events. This returns user-allocated pointers, file descriptors, and other data in a union with no easy way of knowing the active member, something which tsan11rec cannot currently handle. This problem can be avoided by using httpd's option to switch to a simpler but slightly less efficient syscall, poll, which instead simply listens to file descriptors; the results presented here employ this workaround. A strength of rr is that it can handle httpd without this workaround, due to its non-sparse record and replay mechanism.

Experimental setup We tested httpd version 2.4.28 in single-process-multiple-thread mode using ab, an Apache-provided program for server stress testing. We sent 10,000 queries across 10 concurrent threads to an httpd server for each of the setups shown in Table 2, averaging results over 10 runs. We report on standard deviation and again remark on the CV. In the table, rnd and queue refer to configurations of tsan11rec, and the presence or absence + rec indicates whether recording was enabled.

The standard variation and CV is also shown. In the table, *rnd* and *queue* refer to configurations of tsan11rec, and the presence or absence of *+ rec* indicates whether recording was enabled.

Results and discussion The results are shown in Table 5.5. The columns under *Race reports* show regular results with race reporting enabled; the data under *No reports* shows results where race-checking-capable tools do perform race checking behind the scenes, but do not actually emit race reports. This distinction is necessary because tsan11 detects such a large number of races that the overhead associated with generating reports noticeably affects performance; results when fewer races are detected are more representative of the performance one would expect using a future version of httpd in which many race issues are fixed. The *Throughput* columns indicate the mean number of queries the server responds to per second. The *Rate* column is the mean number of race reports given during execution (only relevant for tsan11-based configurations). For each configuration, the standard deviation is shown in parentheses. Variance, as measured by CV, is low: below

0.8 in all cases and usually less than 0.5. The Overhead columns indicate how much slower performance is compared with native execution.

Without reporting, tsan11 already incurs a 3× overhead compared to native. Comparing results for *native* with *rnd* and *rnd+rec* shows that adding controlled random scheduling increase this overhead massively, by 79–89× depending on whether recording is enabled. This is in the same ball park as the overhead associated with rr: 61× without race checking and 160× with tsan11 instrumentation (but still with the actual reporting of race disabled). In contrast, when the queue strategy is used, the overhead compared with *native* drops to 9× and 21× with recording disabled vs. enabled. This can be attributed to the gap between rr/rnd and queue to httpd's heavy reliance on parallelism and frequent use of shared mutexes. This parallelism is removed by rr because the tool sequentialises the execution of threads, while tsan11rec's random scheduler only allows the thread that it has chosen to be scheduled next to execute a visible operation, even if many other threads are ready to execute visible operations. In contrast, the queue strategy allows threads to perform visible operations largely on demand. Turning to the results with race reporting enabled, the queue strategy has the highest race detection rate, improving on uncontrolled tsan11. All other race detecting configurations lower the race detection rate; it is likely that this is because rr and rnd reduce the number of queries being responded to concurrently.

Comparing demo file sizes when recording is enabled, the tsan11rec demo files are around 48MB for both strategies, dropping to 4.8MB when only 1000 queries are issued, suggesting that demo file size increases linearly with the number of requests at a rate of 4.8KB per request. This could be reduced further with a more aggressive compression strategy, but would likely increase the time overhead. The demo file for rr is significantly smaller: 6.6MB for 10000 queries, which goes down to 3.9MB with 1000 queries, implying a rate of 0.3KB per request plus a constant 3.6MB.

5.3.5 PARSEC and pbzip

Overview The PARSEC benchmark suite [BKSL08] and the pbzip application [pbz18] are both widely used for evaluating concurrency analysis tools. For PARSEC, the benchmarks used to evaluate iReplayer [LSW+18] have been used, however, three of these are unusable (dedup and swaptions do not compile, and canneal crashes).

Experimental setup Each PARSEC (version 3.0) benchmark was run with the 'simlarge' test size shipped with the benchmarks, using 4 threads. Pbzip (version 2-1.1.13) was used to compress a 400MB file with 4 threads. Each benchmark was run 10 times

Setup	Race reports Throughput	Overhead	Rate	No reports Throughput	Overhead
Native	N/A	N/A	N/A	28895 (4622.56)	1×
rr	N/A	N/A	N/A	475 (6.08)	61×
tsan11	3687 (294.28)	8×	113 (19.85)	9824 (1432.01)	3×
tsan11 + rr	86 (63.20)	336×	34 (16.80)	181 (46.37)	160×
rnd	141 (8.92)	205×	162 (38.78)	367 (33.26)	79×
queue	818 (310.33)	35×	381 (73.62)	3261 (843.67)	9×
rnd + rec	142 (11.85)	203×	155 (31.03)	326 (38.94)	89×
queue + rec	513 (85.34)	56×	360 (64.28)	1387 (249.61)	21×

Table 5.5: Comparison of throughput and race rate between native, tsan11, rr and tsan11rec for Apache's httpd. Results are averaged over 10 runs. "Throughput" shows mean throughput in queries per second, "Rate" is the number of races detected per run (where relevant). Standard deviations are shown (in parentheses). Overhead is computed relative top native throughput.

per tool configuration, reporting average runtimes. As before, the standard deviation and CV is provided and remarked upon. A small number of races were discovered for some benchmarks, and the race detecting tools largely agreed on the number of races, this is not detailed further.

Results and discussion Table 5.6 shows the average time taken to run each benchmark with each tool configuration, with standard deviation. Variance, as measured by CV, is reasonably low (CV is always below 1). For the tsan11rec results, *+ rec* indicates whether recording was enabled. Table 5.7 is computed from the data of Table 5.6, and reports the overhead associated with running each tool configuration compared with native execution.

With the exception of bodytrack and fluidanimate, the overhead tsan11rec brings over that of tsan11 is small, and for all benchmarks whether recording is enabled or not makes little difference. However, the overhead associated with tsan11 + rr (i.e., running tsan11-instrumented code under rr) is significant compared with the tsan11 overhead alone, despite the fact that running under rr without race detection is generally efficient. Interestingly, rr *without* race detection performs poorly on blackscholes compared with the tsan11rec configurations. Digging into this reveals that the benchmark distributes work between threads at the start of execution and then lets threads run with little interaction. This high parallelism/low communication execution plays to the strengths of tsan11rec, where invisible operations are left to run in parallel, but is bad for rr, which forces sequentialisation across all operations. As discussed above, race reports (omitted) were few, and similar between scheduling strategies, with the queue strategy faring best.

					tsan11rec			
Program	**native**	**tsan11**	**rr**	**tsan11 + rr**	**rnd**	**queue**	**rnd + rec**	**queue + rec**
pbzip	9.2 (0.31)	11.7 (0.49)	66.4 (3.11)	77.2 (1.87)	18.1 (0.30)	12.3 (0.66)	18.2 (0.30)	12.9 (1.29)
blackscholes	0.4 (0.03)	0.8 (0.07)	1.0 (0.01)	2.0 (0.00)	0.7 (0.07)	0.7 (0.06)	0.7 (0.07)	0.7 (0.07)
fluidanimate	0.8 (0.04)	16.0 (1.27)	2.1 (0.01)	38.5 (0.49)	46.4 (3.39)	39.0 (2.67)	50.4 (2.09)	39.8 (1.96)
streamcluster	1.7 (0.19)	38.8 (4.41)	113.3 (3.13)	181.1 (1.79)	103.0 (0.59)	48.5 (2.69)	102.8 (0.18)	41.6 (2.61)
bodytrack	0.5 (0.02)	7.2 (0.36)	3.8 (0.94)	32.7 (0.94)	50.0 (0.40)	7.3 (0.33)	50.0 (0.78)	7.8 (0.23)
ferret	1.2 (0.07)	14.0 (0.86)	8.7 (0.44)	81.5 (2.20)	16.4 (0.66)	14.6 (0.60)	16.7 (0.71)	14.5 (0.37)

Table 5.6: Execution times (s) for tool configurations across the pbzip and PARSEC benchmarks, averaged across 10 runs. Standard deviation is shown (in parentheses).

					tsan11rec			
Program	**native**	**tsan11**	**rr**	**tsan11 + rr**	**rnd**	**queue**	**rnd + rec**	**queue + rec**
pbzip	1.0×	1.3×	7.2×	8.4×	2.0×	1.3×	2.0×	1.4×
blackscholes	1.0×	2.0×	2.7×	5.3×	1.9×	1.9×	1.9×	1.8×
fluidanimate	1.0×	20.3×	2.7×	48.9×	59.0×	49.7×	64.2×	50.6×
streamcluster	1.0×	22.4×	65.4×	104.5×	59.5×	28.0×	59.3×	24.0×
bodytrack	1.0×	13.5×	7.2×	61.4×	93.8×	13.8×	93.9×	14.7×
ferret	1.0×	11.9×	7.4×	69.5×	14.0×	12.5×	14.3×	12.4×

Table 5.7: Overhead compared with native execution for tool configurations across the pbzip and PARSEC benchmarks, computed from the results of Table 5.6.

5.3.6 SDL-based Games

Overview Simple DirectMedia Layer (SDL) is a library that consolidates input, graphics and various other forms of I/O under a single interface [Sim18], and is typically used for games. On Ubuntu 16.04, SDL communicates with X11 for I/O, pulseaudio for sound and OpenGL for display. Two SDL-based games are investigated: Zandronum [Sam18], a multi-player Doom port (≈400kLOC), and QuakeSpasm [Qua18] (≈88kLOC), a port of Quake. While these games support custom record and replay by logging high level commands, by working at the threading and system call level tsan11rec can facilitate record and replay of bugs that rely on low-level interactions to manifest. One such example, discussed below, is the successful record and replay of a bug in Zandronum that arises due to communication of game data between the game client and server, which is not present in the game's native replay.

Initial attempts to replay these SDL-based games failed due to communication between the application and the closed and proprietary NVIDIA OpenGL module via `ioctl` syscalls. This was worked around by ignoring `ioctl` during recording, and letting it run natively during replay. This works because communication with the display driver has no affect on the game logic. Display interaction led to further problems with initialization of the input module. To handle this, the scheduler was adapted to let the application run uninstrumented until the SDL module initialization had completed, adding a custom scheduler hook to allow the application to notify the scheduler when this happens. These problems are not specific to this approach or tool—indeed rr cannot handle these SDL-based games for similar reasons—but are rather a fundamental limitation of recording and replaying applications that make heavy use of I/O. To handle such applications, one either needs to fully mock out I/O components, requiring a tremendous engineering effort, or carefully determine those components that should not be instrumented and specify this either within the instrumentation library or via program annotations.

With these workarounds tsan11rec is able to handle both games, with the nice property that gameplay is displayed on screen during replay; something that would not be possible if the I/O subsystem had been mocked out. This visibility might be useful in the human-assisted debugging of bugs related to visual artifacts.

Experimental setup Measuring game performance in a way that allows us to compare the overhead of the scheduling strategies is non-trivial. The only available metric is the frame rate (fps), which is the number of frames drawn to the screen each second. QuakeSpasm and Zandronum are capped at 60 fps, and will try to maintain this frame-

rate, dipping if they cannot keep up. If the frame rate is reduced too much, the game becomes unplayable. As a best-effort evaluation mechanism, the playability of the games are analysed under various tool combinations. Additionally, it is possible to remove the frame rate cap for Quakespasm, and thus find ball-park figures for the overheads of various tool configurations when playing Quakespasm uncapped. Removing the frame rate cap for Zandronum was not possible. As mentioned previously, there is no comparison with rr, as it cannot record or replay the games. The Zandronum revision is 10013:dd3c3b57023f updated to use SDL2, QuakeSpasm version is 0.93.0, and SDL version is 2.0.5.

Results and discussion With the random tsan11rec scheduler, Zandronum is unplayable even with recording disabled, the frame rate drops to below 1 fps. This is due to the random scheduler starving the main thread by frequently scheduling other less critical threads (e.g. the audio thread). In contrast, the queue scheduler could maintain the full 60 fps with recording enabled; for 100 seconds of play the demo size grew to just under 8MB, of which 6.5MB was for syscalls.

To test tsan11rec's ability to replay network communication, a previously fixed Zandronum bug [Zan15] that relies on an error in this communication to manifest was reintroduced. This bug involves incorrect game state information being sent from the server to the client during a map change. This bug was replicated with a server and two clients, one of which was recording. After about 12 minutes the bug appeared and resulted in a demo file of 43MB. The demo was then replayed and the bug appeared as expected. This demonstrates that tsan11rec can be used to accurately capture and facilitate replay of bugs in large networked applications.

For QuakeSpasm, it is possible to play the game without dropping below 60 fps using tsan11 and all tsan11rec configurations. To further investigate the overhead of each tool configuration on this case study, the fps cap is removed, and the game is then played 5 times per tool configuration, for 90 seconds per play, enabling a mode where the game's fps is periodically appended to a file. A best effort is made to play the game in a similar manner on each run, but inevitably there will still be high variation in game activity between plays. Indicative results are shown in Table 5. The rnd and queue configurations refer to tsan11rec with the random and queue strategies, respectively, and with recording disabled, while the "+ rec" tool configurations are similar but with recording enabled. The "Overhead" column shows the overhead observed compared with native execution. The take-away from these results is that the instrumentation overhead for both tsan11 and tsan11rec is surprisingly modest (generally less than 2×), and that the additional overhead associated with enabling recording in tsan11rec is low.

5.3.7 Limitations: SQLite and SpiderMonkey

A downside of the sparse approach to record and replay is that different applications may have incompatible requirements regarding what should be recorded and what must *not* be recorded. For example, recording memory layout and attempting to enforce the same layout on replay would not only slow down the SDL games (see §5.3.6) to the point of being unplayable, but would also cause problems related to communication with the display driver. Yet, the behaviour of some programs will depend on the memory layout, such as iterating over an ordered C++ container that holds pointers.

To demonstrate the limitations, tsan11rec was applied to the SQLite database management library [SQL18] and to SpiderMonkey, Firefox's JavaScript management engine [Moz18]. While tsan11rec was applicable for controlled scheduling of these applications, the replay would rapidly desynchronise due to memory layout nondeterminism causing conditionals that rely on the values of pointers to evaluate differently during replay. Tools such as rr can handle these programs reliably by enforcing the same memory layout. This is a trade-off: the non-sparse approach of rr can lead to higher overheads, as demonstrated in §5.3.4 and §5.3.5. An alternative to adapting the record-and-replay tool so that it always enforces memory layout determinism would be to adapt the application of interest so that default memory allocation is replaced with a deterministic memory allocator.

5.4 Summary

After the provision of two dynamic analysis techniques in chapters 4 and 5, this chapter discusses the implementation of these techniques into an already existing dynamic analysis tool, tsan. Each technique has been covered separately, but they do build on top of each other.

The first technique covered is the C++11 aware data race detection, showing that it is possible to have efficient dynamic race detection that can be scaled up to large applications such as FireFox and Chromium, that is aware of many of the nuances of the C++11 memory model, and is capable is exhibiting many of the weak behaviours it allows.

The second technique is is way of applying a custom scheduling strategy to the execution of a program, along with a method to record and replay such executions. This recording mechanism is deemed sparse, due to it recording as little data as possible. Testing has shown that different strategies are possible and give rise to variable time overheads and data race rates. The reliability of this record and replay mechanism is not perfect however,

with many programs requiring special attention to get work, if possible. For the program that do work, space and time overhead varies wildly. In some case, these overheads are lower than rr, but in others, they greatly exceed rr.

6 Conclusion

The work presented in this thesis explores the area of dynamic analysis on concurrent programs, particularly those that utilise weak memory models. These analysis have focussed in part on practicality, demonstrating their usability in real world tools, such as ThreadSanitizer.

Reflections on the contributions of the thesis are presented in §6.1, ideas for future work in §6.2, and a concluding summary in §6.3.

6.1 Thesis Reflections

The dynamic race detection work of Chapter 3 shows that dynamic race detection can still be applied to weak memory models such as C++11, and that simple techniques such as a software store buffer can be used to explore some of the weak behaviours that C++11 allows. This does not mean that catering to the full generality of the C++11 memory model is trivial however, as there are still some aspects which have not been solved, such as how to efficiently explore load buffering in a dynamic analysis. Regardless, the work presented here provides the groundwork that can be extended or modified to work with other weak memory models.

Chapter 4 takes a dive down the rabbit hole of record and replay. In particular, recording and replaying with as little information about the program as possible, known as *sparse* record and replay. The inspiration for this came from many of the old PC games from the '90s. For example, if you load up Quake on a modern system, you are immediately greeted with a replay of a demo recorded in 1996. The demo files that allow this are small, as they only contain information on which keys are pressed and when. It should be noted however that this is only possible because these games have been carefully engineered such that non-determinism only comes from the user input. Outside of games, many programs, either deliberately or accidentally, make use of many forms of non-determinism. This can be simple to record, e.g. requiring the results of certain system calls to be captured, or very difficult, e.g. requiring communication with a closed source system driver to be modelled. In some cases, recording the result of one system call will have a cascading effect on other

system calls, requiring those to be recorded as well, which is the case for many of the file operations. This has ultimately led to mixed results in terms of reliability. While some programs required very little effort to get working, such as litmus tests and Apache httpd, others required months of work, with some not working at all, such as SQL. This work should therefore be considered as a guide, and not a formal technique, that future works may use to avoid many of the problems involved in this field.

In Chapter 5, the tsan11 and tsan11rec tools demonstrate that the approaches of Chapters 3 and 4 can in fact be implemented, and that they can scale to large applications. While the results present some evidence that the techniques can help with finding bugs, the focus of the analyses has mainly been on the performance overhead associated with the instrumentation, and that there is a lot of room for more empirical work on bug finding ability.

6.2 Future Work

Several avenues for future work related to the contributions of this thesis are now presented.

Full coverage of happens before in C11 The extended vector clock algorithm of §3.3.3 almost fully covers the C11 memory model. The remaining part not covered concerns the *consume* memory ordering. This is an unusual ordering that is only possible on certain systems, and only utilised in places where speed is critical, such as the Linux kernel. Most prior work in this area forgoes this ordering however, due to the esoteric nature and difficulty of handling it. In order to include it, data dependency must be tracked, as happens before is a superset of $((a, b) \circ \textit{data-dependency}^{+})$, where $(a, b) \in \textit{reads-from}$, a is a release store, and b is a load with consume ordering.

Full coverage of C11 exploration In §3.4, a technique for allowing the exploration of certain weak behaviours under the presence of sequential consistency was shown. Specifically, it showed that storing the stores to atomic locations in a buffer allowed for load buffering. What was not covered however was load buffering, a behaviour whereby a load reads from some future store. This is difficult to handle because enabling it would require some degree of speculative execution, with the possibility of rolling back the execution if the speculation proved to be wrong. With both load buffering and store buffering made possible, even a system that enforces sequential consistency would be able to explore all of the behaviours allowable in C11/C++11.

Improving the reliability of sparse record and replay The main limitation of the sparse record and replay idea presented in Chapter 4 is exposed by the evaluation in §5.3, which shows that while sparse record and replay can replay programs that cannot be recorded by current tools, it lacks the reliability and out-of-the-box functionality of the current state of the art tool rr [OJF+16]. Further work on the tsan11rec would be necessary for it to be considered a viable alternative to rr.

This is a difficult topic to approach, as the work shown in §4 and §5.3 suggests that there is no generalised solution for the set of system calls that will make tsan11rec work out-of-the-box for most applications. For example, recording the value of pointers returned from memory allocation is necessary for applications such as SpiderMonkey, but would be detrimental for the SDL games.

One possible idea would be to run the program normally, collecting data about the program for later use during record and replay. For example, analysing control flow information to determine parts of the program that are affected by certain system calls, or performing data flow analysis to determine where the data returned from certain system calls is used.

This is both a technical and philosophical challenge for which it is clear there is no perfect solution.

6.3 Summary

Given the difficult nature of finding and explaining bugs, the need for techniques to aid in the discovery of such bugs is ever present. With programs becoming progressively larger and more complicated, improving upon existing techniques is crucial. The work presented here has demonstrated that the introduction of weak memory models is not impossible to work around, and that there is still much room for improvement with regards to record and replay. With a little determination, tools can be created to help stamp out bugs in even the most complicated programs.

Bibliography

[AC10] Remzi H. Arpaci-Dusseau and Brad Chen, editors. *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX Association, 2010.

[ACN+01] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, and John M. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, USA, April 23-27, 2001*, page 23. IEEE Computer Society, 2001.

[Adv10a] Sarita Adve. Data races are evil with no exceptions: Technical perspective. *Commun. ACM*, 53:84–84, 2010.

[Adv10b] Sarita V. Adve. Data races are evil with no exceptions: technical perspective. *Commun. ACM*, 53(11):84, 2010.

[AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 512–532. Springer, 2013.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[Apa18] Apache Software Foundation. Apache httpd. `https://httpd.apache.org/dev/devnotes.html`, 2018.

[AS09] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In Jeanna Neefe Matthews and Thomas E. Anderson, editors,

Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009, pages 193–206. ACM, 2009.

[AWHF10] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In Arpaci-Dusseau and Chen [AC10], pages 193–206.

[BAD+10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In Hoe and Adve [HA10], pages 53–64.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In Ferrante and McKinley [FM07], pages 12–21.

[BBKE13] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In Shahram Izadi, Aaron J. Quigley, Ivan Poupyrev, and Takeo Igarashi, editors, *The 26th Annual ACM Symposium on User Interface Software and Technology, UIST'13, St. Andrews, United Kingdom, October 8-11, 2013*, pages 473–484. ACM, 2013.

[BCdJ+06] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM.

[BDW16] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and opencl. In Bodík and Majumdar [BM16], pages 634–648.

[BF13] Hans-Juergen Boehm and Cormac Flanagan, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013.

[BG91] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '91, pages 194–206, New York, NY, USA, 1991. ACM.

[BHCG10] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In Arpaci-Dusseau and Chen [AC10], pages 177–191.

[BJH+16] Anton Burtsev, David Johnson, Mike Hibler, Eric Eide, and John Regehr. Abstractions for practical virtual machine replay. In Vishakha Gupta-Cledat, Donald E. Porter, and Vivek Sarkar, editors, *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Atlanta, GA, USA, April 2-3, 2016*, pages 93–106. ACM, 2016.

[BKMN10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In Hoe and Adve [HA10], pages 167–178.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In Andreas Moshovos, David Tarditi, and Kunle Olukotun, editors, *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, pages 72–81. ACM, 2008.

[BM16] Rastislav Bodík and Rupak Majumdar, editors. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016.

[BMO+12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 509–520. ACM, 2012.

[BOS+10] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency: The post-Rapperswil model. Technical Report N3132=10-0122, JTC1/SC22/WG21 – The C++ Standards Committee, 2010.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In Ball and Sagiv [BS11], pages 55–66.

[BS11] Thomas Ball and Mooly Sagiv, editors. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011.

[BWB+11] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 113–124. ACM, 2011.

[Cha99] M. E. Chastain. Mec. `https://lwn.net/1999/0121/a/mec.html`, January 1999.

[CRSB16] Man Cao, Jake Roemer, Aritra Sengupta, and Michael D. Bond. Prescient memory: exposing weak memory model behavior by looking into the future. In Christine H. Flood and Eddy Zheng Zhang, editors, *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, Santa Barbara, CA, USA, June 14 - 14, 2016*, pages 99–110. ACM, 2016.

[CSL+13] Heming Cui, Jirí Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 388–405. ACM, 2013.

[CSW18] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, power, arm, and C++. In Foster and Grossman [FG18], pages 211–225.

[CV16] Soham Chakraborty and Viktor Vafeiadis. Validating optimizations of concurrent C/C++ programs. In Björn Franke, Youfeng Wu, and Fabrice Rastello, editors, *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 216–226. ACM, 2016.

[CWG+11] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In Wobber and Druschel [WD11], pages 337–351.

[Dar96] Ian F. Darwin. *Checking C programs with lint - C programming utility.* O'Reilly, 1996.

[DCD+14] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 525–540. USENIX Association, 2014.

[Dev18] P. Deva. Chronon. `http://chrononsystems.com/blog/design-and-architecture-ofthe-chronon-record-0`, 2018.

[DGHH+15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, pages 4:1–4:11, New York, NY, USA, 2015. ACM.

[DKC+02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In David E. Culler and Peter Druschel, editors, *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002.* USENIX Association, 2002.

[DLCO09] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In Mary Lou Soffa and Mary Jane Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 85–96. ACM, 2009.

[DLFC08] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In David Gregg, Vikram S. Adve, and Brian N. Bershad, editors, *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008*, pages 121–130. ACM, 2008.

[DNB+11] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: a relaxed consistency deterministic computer. In Gupta and Mowry [GM11], pages 67–78.

[DV16] Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 413–430. Springer, 2016.

[DvR08] Richard Draves and Robbert van Renesse, editors. *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008.

[EA03] Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 237–252. ACM, 2003.

[EAW10] Jakob Engblom, Daniel Aarno, and Bengt Werner. Full-system simulation from embedded to high-performance systems. In Rainer Leupers and Olivier Temam, editors, *Processor and System-on-Chip Simulation*, pages 25–45, Boston, MA, 2010. Springer US.

[EQR11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In Ball and Sagiv [BS11], pages 411–422.

[EQT10] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race-aware Java runtime. *Commun. ACM*, 53(11):85–92, 2010.

[FF09] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 121–133. ACM, 2009.

[FF10a] Cormac Flanagan and Stephen N. Freund. Adversarial memory for detecting destructive races. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 244–254. ACM, 2010.

[FF10b] Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In Sorin Lerner and Atanas Rountev, editors, *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010*, pages 1–8. ACM, 2010.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

[FG18] Jeffrey S. Foster and Dan Grossman, editors. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 2018.

[FM07] Jeanne Ferrante and Kathryn S. McKinley, editors. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 2007.

[GASS06] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications (awarded best paper!). In Atul Adya and Erich M. Nahum, editors, *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, pages 289–300. USENIX, 2006.

[GB15] David Grove and Steve Blackburn, editors. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 2015.

[GM11] Rajiv Gupta and Todd C. Mowry, editors. *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. ACM, 2011.

[go] Go languge specification. `https://golang.org/ref/spec`.

[God05] Patrice Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[Goo16] Google. KernelThreadSanitizer, a fast data race detector for the Linux kernel, visited November 2016. `https://github.com/google/ktsan`.

[Got] C. Gottbrath. Reverse debugging with the totalview debugger. In *Cray User Group Conference 2008, Helsinki, Finland, May 5–8, 2009.*

[GPB+06] Brian Goetz, Tim Peierls, Joshua J. Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice.* Addison-Wesley, 2006.

[GWT+08] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: an application-level kernel for record and replay. In Draves and van Renesse [DvR08], pages 193–208.

[HA10] James C. Hoe and Vikram S. Adve, editors. *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010.* ACM, 2010.

[HC15] Petr Hosek and Cristian Cadar. VARAN the unbelievable: An efficient n-version execution framework. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 339–353. ACM, 2015.

[HCH17] Shiyou Huang, Bowen Cai, and Jeff Huang. Towards production-run heisenbugs reproduction on commercial hardware. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 403–415, 2017.

[HH08] Derek Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 265–276, 2008.

[HH12] Gernot Heiser and Wilson C. Hsieh, editors. *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012.* USENIX Association, 2012.

[HLZ10] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In Gruia-Catalin Roman

and André van der Hoek, editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 207–216. ACM, 2010.

[HP07] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In Manuvir Das and Dan Grossman, editors, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*, pages 9–14. ACM, 2007.

[HT14] Nima Honarmand and Josep Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 455–456. IEEE Computer Society, 2014.

[HZD13] Jeff Huang, Charles Zhang, and Julian Dolby. CLAP: recording local executions to reproduce concurrency failures. In Boehm and Flanagan [BF13], pages 141–152.

[ISO11a] ISO/IEC. Programming languages – C++. International standard 14882:2011, 2011.

[ISO11b] ISO/IEC. Programming languages – C. International standard 9899:2011, 2011.

[ISZ99] Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. Toward integration of data race detection in DSM systems. *J. Parallel Distrib. Comput.*, 59(2):180–203, 1999.

[JYS12] Huafeng Jin, Tuba Yavuz-Kahveci, and Beverly A. Sanders. Java memory model-aware model checking. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2012.

[KBG16] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *46th Annual IEEE/IFIP International Conference on Dependable Systems*

and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016, pages 431–442. IEEE Computer Society, 2016.

[KW15] Robbert Krebbers and Freek Wiedijk. A typed C11 semantics for interactive theorem proving. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 15–27. ACM, 2015.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In Wobber and Druschel [WD11], pages 327–336.

[LCFN12] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: hybrid program analysis for determinism. In Vitek et al. [VLT12], pages 463–474.

[LD] Christopher Lidbury and Alastair F. Donaldson. Sparse record and replay with controlled scheduling. `https://www.doc.ic.ac.uk/~afd/homepages/papers/pdfs/2019/PLDI.pdf`.

[LD17a] Christopher Lidbury and Alastair F. Donaldson. Companion webiste for reproducibility of experiments, 2017. `http://multicore.doc.ic.ac.uk/projects/tsan11/`.

[LD17b] Christopher Lidbury and Alastair F. Donaldson. Dynamic race detection for C++11. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 443–457. ACM, 2017.

[LGV16] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In Bodík and Majumdar [BM16], pages 649–662.

[LKZ14] Kyu Hyung Lee, Dohyeong Kim, and Xiangyu Zhang. Infrastructure-free logging and replay of concurrent execution on multiple cores. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 232–256. Springer, 2014.

[LLCD15] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In Grove and Blackburn [GB15], pages 65–76.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 329–339, 2008.

[LSW+18] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. ireplayer: in-situ and identical record-and-replay for multithreaded applications. In Foster and Grossman [FG18], pages 344–358.

[LVN10] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In Vishal Misra, Paul Barford, and Mark S. Squillante, editors, *SIGMETRICS 2010, Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, New York, USA, 14-18 June 2010*, pages 155–166. ACM, 2010.

[LWV+10] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In Hoe and Adve [HA10], pages 77–90.

[LZTZ15] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. Light: replay via tightly bounded recording. In Grove and Blackburn [GB15], pages 55–64.

[Mat88] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[MCT08] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution effciently. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 289–300, 2008.

[MGT+17] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazières, and Mendel Rosenblum. Towards practical default-on multi-core record/replay.

In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 693–708. ACM, 2017.

[Mic18] Microsoft. Understanding intellitrace part i: What the @#$% is intellitrace? `https://blogs.msdn.microsoft.com/zainnab/2013/02/12/understanding-intellitrace-part-i-what-the-is-intellitrace`, 2018.

[Moz18] Mozilla. Spidermonkey. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`, 2018.

[MPN13] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In Boehm and Flanagan [BF13], pages 187–196.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Ferrante and McKinley [FM07], pages 446–455.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In Draves and van Renesse [DvR08], pages 267–280.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming - a POSIX standard for better multiprocessing.* O'Reilly, 1996.

[ND13] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 131–150, 2013.

[ND16] Brian Norris and Brian Demsky. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.*, 38(3):10, 2016.

[NMS16] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for C/C++11 concurrency. In Eelco Visser and Yannis Smaragdakis,

editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 111–128. ACM, 2016.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA*, pages 284–295. IEEE Computer Society, 2005.

[OJF+16] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Lightweight user-space record and replay. *CoRR*, abs/1610.02144, 2016.

[OJF+17] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 377–389, 2017.

[Ora10] 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.

[PAC18] Luís Pina, Anastasios Andronidis, and Cristian Cadar. *FreeDA*: deploying incompatible stock dynamic analyses in production via multi-version execution. In David R. Kaeli and Miquel Pericàs, editors, *Proceedings of the 15th ACM International Conference on Computing Frontiers, CF 2018, Ischia, Italy, May 08-10, 2018*, pages 1–10. ACM, 2018.

[pbz18] pbzip2 development team. pbzip2. `https://launchpad.net/pbzip2`, 2018.

[PDP+13] Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Emile Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T. King, and Josep Torrellas. Quickrec: prototyping an intel architecture extension for record and replay of multithreaded programs. In Avi Mendelson, editor, *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 643–654. ACM, 2013.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 320–331. ACM, 2006.

[PK16] Daniel Poetzl and Daniel Kroening. Formalizing and checking thread refinement for data-race-free execution models. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2016.

[PPS+10] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors, *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 2–11. ACM, 2010.

[PS03] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multihreaded C++ programs. In Rudolf Eigenmann and Martin C. Rinard, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, pages 179–190. ACM, 2003.

[PS07] Eli Pozniansky and Assaf Schuster. Multirace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[PS16] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In Bodík and Majumdar [BM16], pages 622–633.

[PSN16] Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016.

[Qua18] QuakeSpasm. Quakespasm: An engine for id software's quake. `http://quakespasm.sourceforge.net/`, 2018.

[RGEV11] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of javascript benchmarks. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 677–694. ACM, 2011.

[Rod85] David P. Rodgers. Improvements in multiprocessor system design. In Thomas F. Gannon, Tilak Agerwala, and Charles Freiman, editors, *Proceedings of the 12th Annual Symposium on Computer Architecture, Boston, MA, USA, June 1985*, pages 225–231. IEEE Computer Society, 1985.

[Sai05] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In Clinton Jeffery, Jong-Deok Choi, and Raimondas Lencevicius, editors, *Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005, Monterey, California, USA, September 19-21, 2005*, pages 69–76. ACM, 2005.

[Sam18] Torr Samaho. Zandronum. `https://zandronum.com/`, 2018.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In Heiser and Hsieh [HH12], pages 309–318.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA 2009, New York, NY, USA*, pages 62–71, 2009.

[Sim18] Simple DirectMedia Layer. Sdl 2.0 library. `https://www.libsdl.org/download-2.0.php`, 2018.

[SJ12] Deepa Srinivasan and Xuxian Jiang. Time-traveling forensic analysis of vm-based high-interaction honeypots. In Muttukrishnan Rajarajan, Fred Piper,

Haining Wang, and George Kesidis, editors, *Security and Privacy in Communication Networks*, pages 209–226, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[SMO+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In Vitek et al. [VLT12], pages 311–322.

[SQL18] SQLite. Sqlite 3.24.0. `https://sqlite.org/releaselog/3_24_0.html`, 2018.

[SS15] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in C++. In Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars, editors, *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, pages 46–55. IEEE Computer Society, 2015.

[Ste93] Nicholas Sterling. WARLOCK - A static data race analysis tool. In *USENIX Winter*, pages 97–106, 1993.

[TDB14] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using schedule bounding: an empirical study. In José E. Moreira and James R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 15–28. ACM, 2014.

[TLH+07] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 131–144. ACM, 2007.

[Und18] Undo. Reversible debugging tools for c/c++ on linux & android. 2018.

[VBC+15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In Sriram K. Rajamani

and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 209–220. ACM, 2015.

[VCS+17] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. Taming parallelism in a multi-variant execution environment. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 270–285. ACM, 2017.

[VCV+16] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, pages 167–179. USENIX Association, 2016.

[VJL07] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 205–214. ACM, 2007.

[VLT12] Jan Vitek, Haibo Lin, and Frank Tip, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, 2012.

[VLW+11] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In Gupta and Mowry [GM11], pages 15–26.

[vVZN+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[Vyu16] Dmitry Vyukov. Relacy race detector, visited November 2016. `http://www.1024cores.net/home/relacy-race-detector`.

[WBBD15] John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson. Remote-scope promotion: clarified, rectified, and verified. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 731–747. ACM, 2015.

[WD11] Ted Wobber and Peter Druschel, editors. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. ACM, 2011.

[WPP+14] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In David R. Kaeli and Tipp Moseley, editors, *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, page 98. ACM, 2014.

[XBH03] Min Xu, Rastislav Bodík, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In Allan Gottlieb and Kai Li, editors, *30th International Symposium on Computer Architecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA*, pages 122–133. IEEE Computer Society, 2003.

[YNPP12] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 485–502. ACM, 2012.

[Zan15] Zandronum. Zandronum bug tracker. `https://zandronum.com/tracker/view.php?id=2380`, 2015. Bug 0002380.

[ZKW15] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In Grove and Blackburn [GB15], pages 250–259.