Assessing the standard-compliance for multi-threading primitives in C compilers

Rick Watertor
rick.watertor@outlook.com

July, 2020, 57 pages

Academic supervisor: Ana Lucia Varbanescu, a.l.varbanescu@uva.nl
Daily supervisors: Marcel Beemster and Nicola Rossi, marcel@solidsands.com, nicola@solidsands.com
Host organisation: Solid Sands, https://solidsands.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

At the core of any multi-threaded program are the multi-threading primitives. These primitives govern thread synchronization and communication. When the primitives fail to provide the agreed upon behaviour, programs relying on this behaviour may behave in unintended or undefined ways. It is the task of the compiler to translate the primitives. Compilers also transform the assembly code to execute more efficiently. In these steps, compilers can make mistakes.

We set out to construct a systematic method to test C/C++ compilers for standard-compliance with respect to atomics and fences. For this purpose, we build a simulator that is rooted in the view of a concurrent program as a set of interleavings of instructions of all threads. It exhaustively executes all interleavings in order to provide us with a deterministic result on whether the implementation of the atomics and fences was correct or not.

We also present an assessment of the simulator's completeness. In the assessment, we compare our simulator to a naive test where the program is run 50000 times. We find that our simulator is unable to detect atomicity violations that depend on same-time execution of multiple threads. We also found that our simulator is able to detect specific invalid execution orders that a naive test otherwise would not be able to. Finally, we found that the compilers we investigated are hesitant to perform transformations or optimizations on multi-threaded code.

Contents

Chapter 1

Introduction

Multi-threaded programs are notorious for being difficult to debug and test as concurrent programs can introduce non-determinism, timing-constraints, multiple data-streams, race-conditions, and deadlocks [1–3]. Even with these difficulties, multi-threaded programs are becoming more and more prevalent. Due to several hardware limitations, better performance can no longer be gained by waiting for new architectures [4]. Multi-threading can be a solution to achieve better performance, as it allows execution of multiple threads at the same time. Testing multi-threading code correctness may therefore become as important as testing non-concurrent software is today.

At the core of any multi-threaded program are multi-threading primitives. These primitives govern thread synchronization and communication. For example, in C, the multi-threading primitives are the thread creation, destruction and joining primitives, condition variables, mutexes, atomics, and fences. It is critical that these primitives function as specified. Due to the non-determinism associated with multi-threading, failures of these primitives can be non-obvious.

In this thesis, we specifically target atomics and fences. Atomics, fences, and their associated memory orders have undergone a significant development until their adoption in Java, and later in C11 and C++11. During this development, subtle issues with their definitions were discovered and rectified [5]. The memory orders are complex and difficult to implement, so a test-suite could help test their implementation.

The behaviour of the primitives is specified in a *standard*. A compiler converts code written in languages such as C, C++, or Java, to lower-level instructions, such as assembly or JVM bytecode, and has to adhere to the language standard. However, not everything is dictated by the standard. The compiler can still decide how it will generate the assembly code, as long as the eventual semantics are as specified. As such, transformations and reorderings of instructions are allowed. Tests can aid the development of the transformations by providing the intuition to what relationships and orderings are valid [6].

Compilers are complex software. To target a specific platform or enable certain optimizations, specific configurations can be used which enable and disable certain behaviour. Some institutions even develop their own compiler to more accurately suit their needs. If a compiler performs an improper compilation, then programs that are compiled with that compiler may have unintended or incorrect behaviour. Functionally testing a compiler is therefore important and, in some industries (e.g., the automotive industry, the industrial automation industry, and the medical industry), mandatory [7]. Especially in these cases, a deterministic compliance-testing method is essential. There should be no chance that the faults the test-suite tests for, can still fail in practice. Therefore, there should be no random sampling of the problem space, and the complete range of executions of a test-case should be tested.

1.1 Research questions

We aim to build a test-suite to validate the standard-compliance of compilers when generating code for atomics and fences. This test-suite must be capable of turning the non-deterministic properties of multi-threading into deterministic test results. We want to exhaustively test all the ways the code can be executed. To achieve this goal, we have two main research questions to answer:

RQ1: Can we define a systematic method to test C compilers for standard-compliance with respect to atomics and fences? We approach RQ1 by basing the testing method on an interleaved

view of the execution of a multi-threaded code. The consecutive instructions of each thread are executed in *some order*. An interleaving of two or more threads is a sequence of all their instructions in one such order. Our goal is to be able to determine all possible interleavings, and from there execute each of them. The challenges with this approach are: keeping problem sizes small enough to *evade combinatorial explosion* of possible interleavings, and *preserving correctness* of the interleaving compared to a real-world execution.

RQ2: How do we assess the completeness of this method? We assess the completeness of the interleaving approach by employing a quantitative approach. We investigate the differences of end-states between a naive test and our proposed approach. The naive test relies on the scheduler of the system, and is invoked by repeated execution of the program. Our proposed approach strategically guides the program to execute all possible interleavings. Comparing the two approaches should yield insight to whether our approach is capable of finding fewer or more standard violations in a program compared to the naive test, and can therefore serve as a measure of completeness.

1.2 Outline

This thesis touches on the fields of software testing and concurrency theory. The memory orders we discuss in this thesis are complex. Therefore, we provide a brief introduction of atomics, fences and memory orders in Chapter 2. Next, we introduce and analyze related work in Chapter 3. Chapter 4 describes the methodology of our interleaving-based approach, and discusses the implementation of this approach in the GNU Debugger (GDB). As part of our methodology we develop a test-suite which we present in Chapter 5. We evaluate our approach in Chapter 6, where we describe our evaluation method and discuss our main findings. Finally, we present our conclusions and answers to the research questions in Chapter 7 together with limitations of this study and suggested directions for future work.

Chapter 2

A Concurrency Primer

In this chapter we shortly introduce multi-threading and its associated primitives. We take a closer look at atomics and fences, and then move onto the C/C++ memory model. The memory model is complex, so we take a deep dive into the semantics and relations of the various memory orders paired with explanations and examples.

2.1 Definitions

To have a more accurate discussion, we start by defining some terms we use throughout the thesis. The definitions that contain a citation consist of a compilation of word-by-word quotes from the respective source.

Object An object represents a value of a given type.

Evaluation The computation of a value or a side-effect.

Side-effect An evaluation that affects the execution environment, for example by modifying an object or file.

Memory location An object of the integer, char, _Bool, floating or pointer types [8].

Synchronization operation A synchronization operation on one or more memory locations is an acquire operation, a release operation, both an acquire and release operation, or a consume operation. A synchronization operation without an associated memory location is a fence and can be an acquire fence, a release fence, or both an acquire and release fence [8].

Synchronizes with (relation) An atomic operation X that performs a release operation on an object M *synchronizes with* an atomic operation Y that performs an acquire operation on M and reads a value written by any side effect in the release sequence headed by X. Informally, performing the release operation X forces prior side effects on other memory locations to become visible to other threads that later perform an acquire or consume operation on M [8].

Data race Unsynchronized operations of two or more threads on the same memory location, of which at least one operation is a write. Additionally, relaxed atomic operations cannot contribute to data races (even though they are not synchronized).

As-if (relation) A relation that applies on two programs, where both result in the same observable behaviour and side-effects. This is a relation that a compiler can exploit to perform single-threaded transformations.

Happens before (relation) An evaluation A *happens before* an evaluation B if A is *sequenced before* B or A *inter-thread happens before* B [8].

Sequenced before (relation) Given any two evaluations A and B on a single thread, if A is *sequenced before* B, then the execution of A shall precede the execution of B [8].

Inter-thread happens before (relation) An evaluation A *inter-thread happens before* an evaluation B if A synchronizes with B, A is *dependency-ordered before* B, or, for some evaluation C [8]:

- A *synchronizes with* C and C is *sequenced before* B,
- A is *sequenced before* C and C *inter-thread happens before* B, or
- A *inter-thread happens before* C and C *inter-thread happens before* B.

Dependency-ordered before (relation) An evaluation A is *dependency-ordered before* an evaluation B if [8]:

- A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or
- for some evaluation C, A is *dependency-ordered before* C and C *carries a dependency* to B.

Carries a dependency An evaluation A *carries a dependency* to an evaluation B if [8]:

- the value of A is used as an operand of B, unless:
  - B is an invocation of the kill_dependency macro,
  - A is the left operand of a && or || operator,
  - A is the left operand of a ?: operator, or
  - A is the left operand of a , operator;

  or
- A writes to a memory location M, B reads from M the value written by A, and A is *sequenced before* B, or
- for some evaluation C, A *carries a dependency* to C and C *carries a dependency* to B.

2.2 Multi-threading

Shared Memory Multithreading (SMMT) is a technique where a program can create multiple *threads* of execution. As the name suggests, each of these threads can access the same memory. An advantage of sharing memory is that no heavyweight forking (copying of the entire program and memory space) needs to be done, and no explicit sending of shared information is required: all threads can access the information directly. However, when threads attempt to access and write to the same information concurrently, a *data race* can occur. While the main memory is shared, the processors contain local registers. An evaluation sequence is therefore not always immediately visible to the shared memory threads. For example, a CPU core could load a value from a memory location and store it into a register. The core then modifies the register and stores the value back to the same memory location. If during the operation another thread loads the same variable from memory, the operation is applied on the old value. Once the store completes, the operation of the original thread has been overwritten due to the unfortunate timing of both threads, resulting in unpredictable behaviour. Any data races result in undefined behaviour in C [8].

2.3 Atomics and Fences

To prevent data races, and to still allow threads to interchange information in the shared memory space, a form of synchronization is mandatory to ensure that concurrent editing of memory has defined behaviour. There are various forms of synchronization, like mutexes, atomics and fences.

In this thesis, we focus on atomics and fences. Atomic objects have the guarantee that all operations on a particular atomic object occur in a total order, their *modification order*. This order respects the *happens before* relation. This order is separate for each atomic object, but mandates that two operations on the same atomic cannot occur at the same time. As a result, atomic objects ensure that the operations on them are data race free. Fences have similar semantics to atomics, but are not applied on a memory location. Therefore, they only impose an order to the statements before and after the fence.

The modification order is not enough for atomics to meaningfully play alongside other atomics or non-atomics. As with sequential code, a compiler is completely free in reordering instructions (including atomics), such that the execution is better suited for a specific target, as long as the transformed program is *as-if* it was the original program. The problem on a multi-threaded level is that this relation is non-obvious: a thread can read from a memory location at any time during the execution, and thus behaviour could be observed at any point in the program. Instruction reshuffling of one thread by the compiler could mean that another thread is observing the reshuffled behaviour, breaking the *as-if* property. This suggests that compilers could not perform any transformations on atomics whatsoever, which could be negative for performance. Instead, by defining *memory orders* that span multiple atomics and non-atomics, transformations can be re-enabled, while the programmer has a contract to what behaviour the thread could possibly observe.

2.4 Memory model

The behaviour of a concurrent program is tightly intertwined with its memory. As such, a clearly defined model on memory orderings and when and how writes become visible is important to both the programmer and the compiler [9]. In 2011, C and C++ implemented a memory model into their specification. In particular, they adopted a variant of Sequential Consistency - Data Race Free (SC-DRF), and specified atomics into the language.

Even though sequential consistent execution allows the reintroduction of some compiler transformations, in some use-cases, it was considered too strong. To allow for even more transformations, the specification also contains *memory relaxation models* [10]. Specifying a weaker memory ordering means that the compiler is allowed to do more transformations, while simultaneously reducing the ordering guarantees. A sequentially consistent ordering (strongest) ensures that *all* atomic operations will become ordered in *one* total *happens before* order, while a relaxed ordering (weakest) ensures no inter-thread ordering. It only guarantees atomicity through the modification order on the relaxed atomic.

A memory model therefore dictates what and how a compiler can and cannot transform, and is therefore necessary for a compiler to efficiently handle code that contains atomics and other concurrency primitives.

2.4.1 Sequential consistency

In the default and strongest C/C++ memory model (SC-DRF), sequential consistency can only be preserved when there are no data races. Sequentially consistent (`memory_order_seq_cst`) atomic operations also provide the most guarantees. Quoting the C specification, this memory order is defined as [8]:

memory_order_seq_cst "A `memory_order_seq_cst` store operation performs a release operation on the affected memory location. A `memory_order_seq_cst` load operation performs an acquire operation on the affected memory location. There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the *happens before* order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:

- the result of the last modification A of M that precedes B in S, if it exists, or
- if A exists, the result of some modification of M that is not `memory_order_seq_cst` and that does not *happen before* A, or
- if A does not exist, the result of some modification of M that is not `memory_order_seq_cst`."

Informally, this means that the sequential consistency memory order imposes a *single happens-before* order on *all* sequential consistent atomics, rather than only on the operations of a single atomic. As a consequence, all threads must see the same order of operations on all atomics.

2.4.2 Relaxed memory orders

C also specifies the orders `memory_order_relaxed`, `memory_order_release`, `memory_order_acquire`, `memory_order_consume`, and `memory_order_rel_acq`. Each of these memory orders are weaker than the sequentially consistent memory order. For clarity, we omit the prefix `memory_order_` in the remainder of the text.

The `relaxed` order guarantees nothing about other atomics nor non-atomics, and therefore is also not included in the synchronization definition: it does not *synchronize with* any other operation. However, it does prevent data races by imposing the modification order on the atomic object. The use cases of `relaxed` are therefore limited, and only useful when the object concerns itself only, such as with counters. We show an example of such an use in Figure 2.1.

Shared Variables

```
atomic_int count = 0;
```

Thread 1

```
1 for(int i = 0; i < 100; i++) {
2     atomic_fetch_add_explicit(&count, 1,
      memory_order_relaxed);
3 }
```

Thread 2

```
for(int i = 0; i < 100; i++) {                1
    atomic_fetch_add_explicit(&count, 1,      2
    memory_order_relaxed);
}                                             3
```

Figure 2.1: Example of correct usage of `relaxed`. Both threads increment the shared atomic object, and do not depend on other (atomic) variables in their execution. The result is guaranteed to be 200.

2.4.3 Release + acquire

The orders `release` and `acquire` are paired, and make the ordering of surrounding operations explicit. We further refer to the combination as `release + acquire`. The `release` order performs a *release operation*, and can only be applied on writes. The `acquire` order performs an *acquire operation*, and can only be applied on reads. Informally, as a consequence of the relations, it is not allowed to move operations and side-effects in the same thread that are *sequenced before* a `release` atomic or fence to after the `release` atomic or fence. It is also not allowed to move operations and side-effects that are sequenced after an `acquire` atomic or fence to before the `acquire` atomic or fence.

In Figure 2.2 we show an example where at least `release + acquire` semantics are needed. These C examples are inspired by the C++ examples from cppreference [11]. The example shows the initialization of `data` on one thread, then setting an atomic `flag` using the `release` order, and then initializing `data2`. Because of the properties of `release`, all memory operations (atomic or not) that are *sequenced before* the `atomic_store` will be guaranteed to be visible to other threads before the atomic operation becomes visible to other threads.

Thread 2 uses the `acquire` memory order to synchronize with thread 1's `release`. This does not mean that the thread will wait until the release operation has happened. When the value is true, it is guaranteed that all operations and side-effects *sequenced before* the release are also visible. Thus a while loop to wait for the value to become true is necessary. After thread 2 sees `flag` become true, it can assert that `data` equals 42 with full confidence.

`data2` is not guaranteed to be 42. In thread 1, `release` only says something about the memory operations *sequenced before* the release. Thus, any operations following the `release` operations have no guarantees, and could be freely reordered before the release operation.

If in these cases the `relaxed` order was used, there would be no guarantee for `data` nor `data2`. `relaxed` provides no guarantees on the surrounding operations (atomic or not), and thus `atomic_store` in thread 1 could be freely reordered before the data initialization. Moreover, the asserts of `data` and `data2` could even have been reordered before the `atomic_load` in thread 2.

We thus see that to ensure something about the `data` variable in this example, we need to have memory ordering semantics at least as strong as `release + acquire`.

Shared Variables

```
atomic_bool flag = false;
int data = 0;
int data2 = 0;
```

Thread 1

```
1 data = 42;
2 atomic_store_explicit(&flag, true,
      memory_order_release);
3 data2 = 42;
```

Thread 2

```
while (!atomic_load_explicit(&flag,          1
    memory_order_acquire))
    ; // Wait until loaded                   2
                                              3
// Never fails                               4
assert(flag == true);                         5
assert(data == 42);                           6
// May fail                                   7
assert(data2 == 42);                          8
```

(a) Due to `release`, *preceding* memory operations become visible before the atomic flag is updated. Any thread `acquiring` this same atomic will see all preceding data stores as well.

(b) Due to `acquire`, all side-effects *preceding* the `release` in thread 1 will be visible before the flag is loaded as true. Thus, the shared variable `data` is guaranteed to hold the asserted value 42, but `data2` has no guarantees.

Figure 2.2: Example of correct and mandatory usage of `release + acquire`. Any stronger memory ordering is also correct. `relaxed` cannot be used here, as the store in thread 1 would not order the surrounding memory operations, and thus the assignment to `data` could be reordered after the atomic, yielding an incorrect result in thread 2's check. Example inspired by cppreference [11].

2.4.4 Release + consume

`release` can also be coupled with `consume`, which is weaker: only the *dependency-ordered before* relation has to be upheld. Informally, this implies that only those operations that a `consume` operation *carries a dependency* to are disallowed to be reordered before the operation. We refer to this combination as `release + consume`. This is weaker than `release + acquire`, since it only imposes constraints on the dependencies, rather than all operations sequenced after the atomic. Due to the weaker property, some architectures are able to more efficiently implement this relation than the `release + acquire` relation [12].

In Figure 2.3, we show an example where `release + consume` can be used. In this example, thread 2 uses `consume` instead of `acquire`. The `consume` memory ordering ensures that only `data` that is dependent on the atomic that is released is synchronized and ordered. `consume` is thus weaker than `acquire`, and `data` is not guaranteed to be 42 after the wait, because `data` does not have any relation to `flag`.

Shared Variables

```
atomic_bool flag = false;
int data = 0;
```

Thread 1

```
1 data = 42;
2 atomic_store_explicit(&flag, true,
      memory_order_release);
```

(a) Due to `release`, preceding memory operations become visible before the atomic `flag` is updated. Any thread acquiring this same atomic will see all preceding data stores as well.

Thread 2

```
1 while (!atomic_load_explicit(&flag,
      memory_order_consume))
2     ; // Wait until loaded
3
4 // Never fails
5 assert(flag == true);
6 assert(data == 42 && flag == true);
7 // May fail
8 assert(data == 42);
```

(b) Due to `consume`, only evaluations *carrying dependency* to `flag` will be guaranteed ordered after `flag` is loaded as true. Thus, the shared variable `data` has no guarantee to hold the asserted value 42 in the third assert, as its assert could have moved before the while loop, in contrast to the example in Figure 2.2. However, the second assert does guarantee a successful result, as flag *carries a dependency* onto that assert.

Figure 2.3: In the previous example, `data` was guaranteed to be 42. Since there is no *dependency* of `data` on `flag`, the use of `consume` still allows free reordering of the assert before the load. Therefore, here, `data` is only guaranteed to be 42 for evaluations that depend on the result of the `consume` operation. Example inspired by cppreference [11].

2.4.5 Acq_rel

Finally, `acq_rel` combines the properties of `acquire` and `release`, meaning that any operations cannot be reordered to before or after the `acq_rel` operation or fence. Still, this does not guarantee *sequential consistency* (provided by the `seq_cst` ordering), since it does not provide the *inter-thread happens before* relation for all atomics.

In Figure 2.4, an example is shown where `acq_rel` is not sufficient, and there is explicit need for `seq_cst`. This is because `acq_rel` does not enforce an *inter-thread happens-before* relation on multiple atomics. Thus, the view of thread 3 and 4 on the atomics `x` and `y` can differ, resulting in both threads skipping the `z` increment. Using `seq_cst` would guarantee that `z` is incremented at least once.

Shared Variables

```
atomic_bool x = false;
atomic_bool y = false;
atomic_int z = 0;
```

Thread 1

```
atomic_store_explicit(&x, true,
    memory_order_seq_cst);
```

Thread 2

```
atomic_store_explicit(&y, true,
    memory_order_seq_cst);
```

Thread 3

```
while (!atomic_load_explicit(&x,
    memory_order_seq_cst))
    ;
if (atomic_load_explicit(&y,
    memory_order_seq_cst)) {
    ++z;
}
```

Thread 4

```
while (!atomic_load_explicit(&y,
    memory_order_seq_cst))
    ;
if (atomic_load_explicit(&x,
    memory_order_seq_cst)) {
    ++z;
}
```

Asserts

```
// Never fails
atomic_load(&z) > 0
```

Figure 2.4: To observe consistent behaviour, the strongest memory order `seq_cst` is used. In no case can `z` be left on 0, because the operations on the atomics `x` and `y` have to be ordered by the *happens-before* relation. Thus, when `x` becomes true, and `y` remains false, thread 3 skips the increment. However, now `x` is true, so thread 4 will always enter the conditional. This also happens vice-versa, when thread 4 runs first, or when either thread runs partially. In other words, at least one thread must hit the increment of `z`. With `acq_rel`, there is no order on both `x` and `y` (but there is an order on each of them separately). Because `x` and `y` are written to in separate threads, the `acq_rel` memory order guarantees nothing about the order in which the operations are observed. `acq_rel` only guarantees a *happens before* relation on a *single* atomic and only a *sequenced before* ordering for multiple atomics on the same thread. It does not impose an order on *both* atomics over multiple threads. Thread 4 could therefore have a different view of the values than thread 3, and thus both could skip the `z` increment. Example inspired by cppreference [11].

Chapter 3

Related Work

In this chapter, we present previous work on the topic of testing multi-threading programs and primitives. For each paper we discuss their approach and limitations from the lens of our goal. We summarize the papers and our analysis at the end of this chapter.

3.1 Testing multi-threaded programs

Chen et al. take an approach where they adjust the *thread speeds* of the concurrent threads [13]. They achieve this by instrumenting read and write events, which then controls the thread speeds. This way they can reach different interleavings, but a limitation seems to be that the speeds are constant for one execution. This means that there is no possibility of testing *all* thread interleavings, such as when a thread first has to go fast, then slow, and then fast again.

A non-deterministic testing approach, exemplified by Edelstein et al. [14], is to inject heuristically calculated sleep statements within the to-be-tested code, resulting in a differently-ordered execution. Their ConTest adds this instrumentation code to a (Java) application's bytecode. Their choice of interleavings are driven by coverage, which means that they dynamically look for the most interesting interleavings. However, it seems limited in machine-code coverage because coverage does not include execution *orders*. Executions could show completely different behaviour on a different interleaving and therefore still yield bugs. Additionally, for functional safety, non-deterministic testing seems to be not enough: the test suite should not succeed while there may still be a scenario where the system fails.

Yang et al. perform structural testing of shared memory multi-threaded programs by replacing the Java Virtual Machine (JVM) with their own Virtual Machine (VM). Their VM re-implements and adds instrumentation to the primitives, such that a deterministic interleaving can be achieved [1]. The replacement of the VM means that the originally implemented primitives can no longer be tested. We also note that this approach is not generalizable to languages such as C/C++, since those languages do not rely on a VM.

MultithreadedTC is an approach that uses an external clock [15]. The external clock is incremented whenever all threads are blocked, and at least one thread is waiting for a tick. This allows the programmer to make assertions about how many ticks have passed, and allows a strict interleaving. This tool however loses the functionality of modular thread interleaving, as the interleavings are defined programmatically on a statement level. Every thread interleaving therefore has to be defined manually, which can be infeasible for larger tests.

IMUnit presents an easier approach of testing multi-threaded programs than MultithreadedTC by allowing the programmer to more easily define event orders [2]. This approach makes more explicit what the expected interleaving is, and events can be named, resulting in overall greater readability. They add a *runner* that is based on a runtime monitoring framework of java (JavaMOP). The runner blocks a thread whenever execution of that thread would cause the defined event orders to be broken. It also contains a deadlock detection unit. In our case, specifying the output for a specific order is difficult, as every compiler compiles code in a different way, with potentially different instructions and instruction orders. As such, we need an approach where we can test the end state, but we cannot check an end state *for a specific order*.

Lei et al. introduce *reachability testing* [16]. Reachability testing is a technique based on *prefix testing* where a test is partially executed deterministically (the prefix), and partially executed non-deterministically. They observed that reachability testing tools are portable, as they do not rely on the

virtual machine, runtime system or operating system. The authors also do note that it is complementary to specification-based testing, because reachability testing is based on the *implementation*: it can not detect if an interleaving should have been possible.

Koushik Sen proposes a random testing approach [17]. Naive random testing executes a test with a random interleaving. A drawback of random testing is that it often results in similar states being tested. Sen proposes an algorithm that samples the state space such that the interleavings are more uniformly distributed over all states. The paper empirically demonstrates a more uniform distribution of interleavings than naive random testing. We note again that for testing of safety-critical systems, random testing is insufficient: if tests have a probability of succeeding while the underlying system is faulty, it could be catastrophic if the unfortunate interleaving does appear in practice. When a system is faulty, a test should deterministically fail (assuming that the test-suite can detect the fault).

CONCURRIT combines model-checking and testing in a way where the model-checker is programmatically steerable [18]. Within the test code, a domain specific language (DSL) is used where the programmer can place constraints on the thread interleavings, resulting in executions the programmer expects.

Maple is a coverage-based testing approach [3]. A predictor determines which interleaving to test next, to aim for as high a coverage as possible. It uses a scheduler to enforce the interleaving. Maple does not test all possible thread interleavings. This is mostly remedied by the coverage-driven testing, but there are still possible interleavings that could result in faulty execution. The authors argue that systematic testing of the entire thread-interleaving space is still infeasible, even with partial order reduction techniques that for example Lei et al. use [16].

3.2 Testing memory models

Most of the previous work is concerned with testing multi-threaded *programs*. We are concerned with testing multi-threaded *primitives*, which is on a smaller scale and is less likely to hit the state explosion. However, we are faced with also testing the memory models, and whether the compilers adhere to them.

Litmus7 is a tool that can test the memory models of parallel systems [19]. The input of the litmus tool is an annotated pseudo-assembly code for the target system and a test condition on the final registers. Litmus then executes all threads simultaneously, collecting all output states that have been observed and checking it with the testing condition. This execution is done randomly, but can be influenced by adding synchronization and time offsets to the threads.

Litmus was extended by Herd, which allows for the testing of any arbitrary memory model [20]. It simulates the memory model defined by the user, and can run litmus tests ontop of that. We note that in this model, the C/C++ memory models could be implemented, and the litmus tests could take the output of a compiler as input, testing the compiler to the C/C++ memory models. For our purposes, we are looking for a deterministic approach to testing, and the random property of litmus does not satisfy this requirement.

Finally, CPPMEM, a tool introduced by Batty et al., can exhaustively generate all witnesses to a certain condition from a fragment of a C++ program [6]. It uses a symbolic operational semantics to find all memory accesses and constraints, and checks these with an axiomatic model. As a result, they can check whether their axiomatic model does not allow illegal states. This tool does not execute code or need the compiler and is therefore unable to be adapted to test the compiler itself.

3.3 Summary

We summarize the list of relevant papers and their applicable subjects in Table 3.1.

Through our analysis of the literature, we found that there is significant work on testing multi-threaded programs. A number of works take a random testing approach which, while beneficial to find bugs and faults, do not ensure *deterministic* finding of issues. In the context of our work, we would like for our tests to yield a deterministic outcome.

Additionally, we would like to be complete and execute all possible interleavings. In the domain of programs, this is difficult because programs are large, and therefore the number of interleavings is extremely large. In the domain of primitives, keeping the number of interleavings small may be more feasible, as indicated by CPPMEM [6].

A number of tools edit the execution environment and inject or edit the primitives for their approach to work. We cannot rely on editing the primitives or the compiler as we would be altering the compiler

Paper	Year	Determinism	Concurrent execution	Completeness	Replaces primitives[1]	Scheduling	Testing	Modularity[2]	Memory models	Programming model
Thread Speeds [13]	2018	✓	✓	×	✓	×	✓	✓	×	LLVM IR
ConTest [14]	2003	✓	✓	✓	✓	×	✓	✓	×	Java
Structural Testing [1]	1999	✓	✓	✓	×	×	✓	✓	×	Java
MultithreadedTC [15]	2007	✓	✓	✓	✓	✓	✓	×	×	Java
IMUnit [2]	2011	✓	✓	✓	×	✓	✓	✓	×	Java
Reachability [16]	2006	Reachability[3]	✓	✓	×	×	✓	✓	×	Java
Random Testing [17]	2007	×	✓	×	×	✓	✓	✓	×	Java
CONCURRIT [18]	2012	✓	✓	✓	×	×	✓	✓	×	C/C++
Maple [3]	2012	✓	✓	×	×	✓	✓	✓	×	C/C++ (on top of PIN [21])
Litmus [19]	2011	×	✓	×	×	×	✓	✓	✓	(Annotated) Assembly
Herd (+ Litmus) [20]	2014	×	✓	×	×	×	✓	✓	✓	Axiomatic memory model
CPPMEM [6]	2011	✓	✓	✓	×	×	✓	✓	✓	C++
This Thesis	2020	✓	✓	✓	×	✓[4]	✓	✓	✓	C/C++

Table 3.1: A summary of topics covered in scientific literature related to deterministic multi-threaded testing.

under test. A promising direction here is by altering only the execution environment and scheduling the instructions in some predefined order.

Finally, like herdtools [20] and CPPMEM [6], we focus on memory orders. In contrast to them, we wish to test the compiler rather than the models. Therefore, we need to include the compiler in the approach, which neither of the two tools do.

[1] Primitives are replaced/extended with instrumentation
[2] Testing suite allows for multiple interleavings without altering the test itself
[3] Reachability testing: mixed determinism and non-determinism
[4] We do not explicitly instruct the scheduler, but we force a schedule through GDB.

Chapter 4

Methodology

In this chapter we present our methodology of building a test-suite. Our methodology consists of deriving test-cases and designing a simulator that can execute the test-cases deterministically and exhaustively. We present a short overview of the method in the first section, and discuss specifics of the method in the remaining sections. In the final section, we present our implementation.

4.1 Method overview

Our goal is to test compilers for standard-compliance when generating code for atomics in C by using an exhaustive testing approach. We build a test-suite that tests the code generated by a compiler for the atomics in a way that all interleavings are tested. For example, when a synchronization primitive is to be executed by two threads, three scenarios can occur:

- Thread 1 reaches the primitive before thread 2.
- Thread 2 reaches the primitive before thread 1.
- Thread 1 and 2 reach the primitive at the same time[1].

Multi-threaded systems are non-deterministic by nature. Threads run separately, so any of the three cases can occur at any point in time. Therefore, a test as is usual in the sequential domain cannot assume anything about the schedule of the threads and might therefore be unable to test details and cannot ensure that the result holds for all scenarios. Moreover, it relies on the scheduler to execute different interleavings, and thus the scheduler needs to be sufficiently uniformly random to be able to hit all interleavings within a reasonable number of runs.

To test a multi-threading primitive, we follow a total of three main steps: (1) translate the its specification to a set of test-cases; (2) extract the behaviour of the test-cases for every thread interleaving; and (3) match the output of the test-cases to the expected behaviour. A schematic overview of these three steps can be seen in Figure 4.1.

For the first step, we develop a strategy for systematically transforming the specification to a test-suite. Because a specification is abstract, and it may not be directly apparent how the test-suite stems from the specification, we will also provide a description, the expected behaviour, and the expected outcomes for each of the tests. We discuss this in Section 4.2.

The second step is to create a simulator that can extract the results for all possible thread interleavings. We discuss the design in Section 4.3, and its implementation in Section 4.4.

In the third and final step, we need to match the observed output with the expected output, and therefore determine if the test succeeded or not. We discuss this as part of our simulator design in Section 4.3.

[1] At the same time in this case means that the instructions happen in such a short time-frame that the results of one thread are not visible to the other threads yet, e.g., execution in the same clock tick.

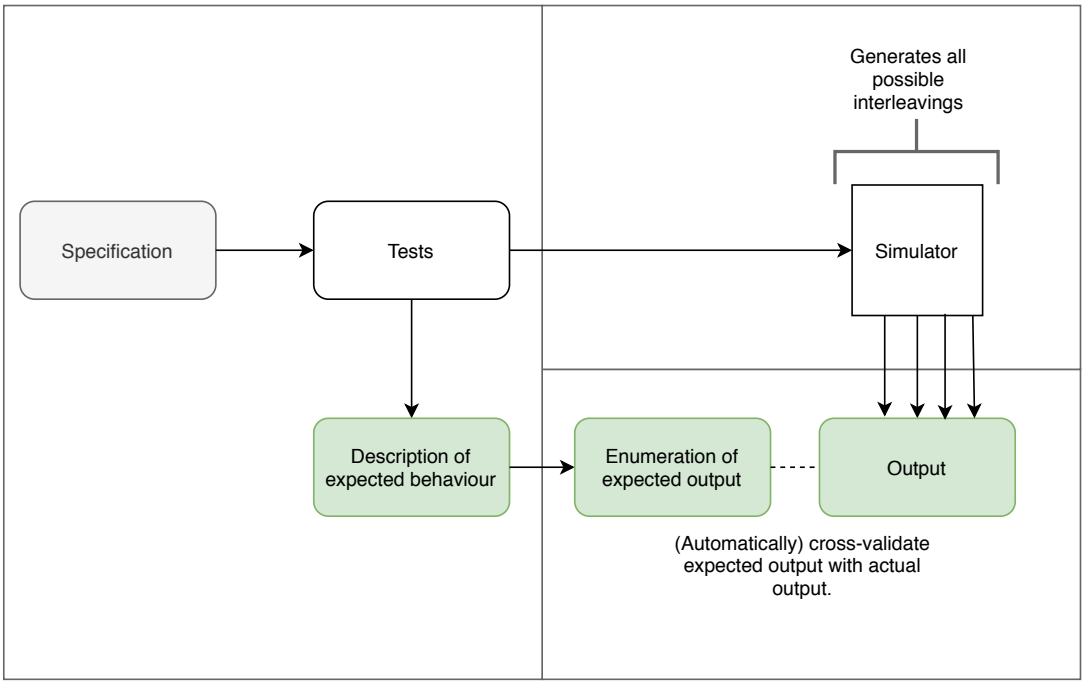

Figure 4.1: A schematic overview of the approach. First, we derive test-cases and describe the expected behaviour. Second, we build a simulator that allows us to execute all interleavings. Finally, we check whether the actual output matches the expected output of the test-case.

4.2 Defining test-cases

In order to define test-cases, we repurpose code samples from literature, and from the specification we gain an understanding of what can be valid output values. Designing new test-cases can be done with other contextual knowledge. In principle the C standard only defines a series of relations between atomic instructions that have to be upheld.

This by itself, however, does not cover a large enough space of things compilers can do: the realm of possible tests is also dependent on the transformations the compilers can produce. Compared to sequential code, different transformations are used when loops are introduced. Conditional cases (e.g., if statements) can be significantly altered by the compiler as well. Therefore, there is no guarantee that one test-case in one context still holds for more complex contexts like loops and branches.

We present by no means a comprehensive testing suite for the entirety of the C concurrency standard. As such, we make clear our method of obtaining tests, such that the suite can be extended. In this section, we describe our systematic approach on adapting examples from literature into tests that our simulator can run.

4.2.1 Obtaining a sample

We discuss our three ways of obtaining a sample: deriving from the specification, deriving from a compiler, and deriving from literature.

Deriving from the specification is the most formal approach of the three. By starting from the C specification, we can systematically derive samples from the rules. For example, when the specification explicitly states that a function cannot be called with a specific argument (e.g., `atomic_explicit_load` with the argument `memory_order_release`), we could define a test that checks whether this behaviour is actually forbidden by the compiler. Similarly, fully generated test suites could be derived from the relations defined in the specification (e.g., the happens-before relation). These are properties of a specific function that are to be upheld. By randomly generating tests, some of these properties can be tested to a much more in-depth level than a single static manually defined test could.

Deriving tests by looking at compiler's source code helps determine the common issues and standard violations that compilers can introduce. Compilers are tools that can be in a lot of states, and can have very complex data structures. Depending on the exact context, intermediate representation, and flags

that are passed to the compiler, very different transformations could be made. Understanding the kind of transformation a compiler makes, and when, can be vital to making a good targeted test-suite that can verify a compiler's transformations.

Deriving tests from literature is another approach that could help in defining a basic test-suite. In literature, many interesting code samples have been shared. Especially in the field of memory orders, a lot of edge cases are being discovered and published. From these, we can derive test-cases that test the particular behaviour the community has touched upon.

For our sample test-suite we use the last approach to generate the samples, and we use the specification to systematically determine the valid outputs of these samples. To ensure that the code sample will work, and is a valid test-case, we filter the code samples according to following criteria:

- The number of total possible interleavings of the test region should not exceed 100000. To prevent state explosion, some additional guidelines can be helpful:
  - It should not contain non-inlinable function calls.
  - It should not contain a significant number of statements or expressions.
  - The code samples should be minimal.
- The test region should not rely on the side-effects introduced by the loops, unless these are trivial to unroll or are only waiting on a condition to hold.
- The code sample should be a valid C program, or easily adaptable as such (e.g., a C++ program or pseudocode).
- The code sample should provide verifiable and defined results, as specified by the C standard.

The reasoning behind each of the constraints is as follows. As the number of total interleavings increases, execution time of a test also increases. At some point, the execution time becomes infeasible. Therefore, we need to constraint the number of possible interleavings at a cut-off. From our initial experiments, we found that anything containing more than 50000 orders takes rather long (i.e., in the range of 30 minutes or longer). To be lenient, we decided to place the cutoff at 100000. Machines differ in execution speed, so some numbers may be more affordable to certain systems than others. We note that therefore this cutoff is very arbitrary, and may become even more lenient if the tests are run in parallel. Most of our test-cases still fit well within these constraints, however, as long as attention is paid to the guidelines.

Non-inlinable function calls significantly impact the number of interleavings. Whenever a function is called, a significant number of instructions deal with setting up the environment (e.g., stack and registers) for that function, which can significantly impact the number of interleavings. From the perspective of the simulator, it could be possible to determine these instructions and skip over them, as they are unlikely to be involved in any transformations or synchronization. However, we found that for our purposes, all related functions were inlinable, so there was no need to support this property.

As the number of statements and expressions increases, so do the number of instructions that have to be generated, and thus the number of possible interleavings. There is no strict limit on statements or expressions because they can be compiled into various numbers of instructions, depending on the compiler and flags. It is a good idea to minimize the number as much as possible. Ensuring that the code sample has as few statements and expressions as possible, means that we want to reduce potential duplication, redundant code, and excessive threads.

Variable length loops are difficult to support, because the number of interleavings is no longer static. Thus, any samples depending on variable length loops are excluded. Fixed length loops can still be included, but should be (manually) unrolled. We found that there are still a number of reasons to wait for a certain condition to hold until continuing execution. Thus, any loops waiting for a condition to change are allowed, but with the constraint that there are no accesses or side-effects in the body of that loop.

Finally, the program should contain verifiable results as defined by the C standard, otherwise the results can be undefined. It is not possible to test undefined results.

4.2.2 Deriving the end-states

To make a functional test out of an suitable sample, we have to determine the valid states the sample can be in after execution. We approach this systematically by reasoning about the code and using the specification in order to find all possible end-states.

We enumerate all states the system can end in by deriving all possible outcomes from the stores and operations that happen in the program. For example, if in the program only the values 0 and 1 are used, and no arithmetic is executed on them, we can determine that the end-states can only contain 0 or 1. This is a consequence of the standard [8]: "An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations."

Then, leveraging the standard and understanding of the test-case, we can identify which of the end-states are a valid execution or not. With this information we can compare the set of derived possible end-states and the set of states that we obtain by executing the program over all interleavings. If there is any state in the *actual* execution set, but not in the set of *expected* states, the test fails and the offending ordering can be reported.

4.3 Simulator design

Our simulator is rooted in the view of a concurrent program as a set of interleavings of instructions of all threads. For example, consider two threads (A and B) both have two instructions to execute (1 and 2), and we denote the particular instructions of the threads as A1, A2, B1 and B2. With these two instructions on both threads, we can make a total of six orderings:

```
1 A1 -> A2 -> B1 -> B2
2 A1 -> B1 -> A2 -> B2
3 A1 -> B1 -> B2 -> A2
4 B1 -> A1 -> A2 -> B2
5 B1 -> A1 -> B2 -> A2
6 B1 -> B2 -> A1 -> A2
```

To extend the example, consider the case that the instructions 1 are a load operation, and the instructions 2 are a store operation of a different value, both on the same memory location which is shared for the two threads. In pseudocode, where `$thread_id` is the name of the executing thread (A or B), this would look like:

```
shared variable y = 0
local register reg = 0

1: reg <- y
2: y <- $thread_id

out: reg
```

Depending on the interleaving, this program can have wildly different outcomes. For example, in the first ordering, thread A loads `y` when it is 0, stores A into `y`, then thread B loads `y`, and stores B into `y`. This would yield on thread A the outcome 0, and on thread B the outcome A. In the last ordering, this would be the inverse: thread B yields the outcome 0 and thread A yields the outcome B. Moreover, the remaining cases make both threads yield 0, because in each of these cases, first both threads load 0 into x, and only then other values are stored into y (thus no longer having impact on the resulting x).

Thus, to understand the precise behaviour that a piece of code can emit, we need to look at each possible interleaving, and determine its output state. This is the approach our simulator must take: it evaluates each interleaving for a small number of instructions on two or more threads and tests whether the output state is contained in the set of expected output states.

4.3.1 Generating the interleavings

A lot goes into setting up a program before it can execute its first user-defined code-line or instruction. And because the number of interleavings scales with a combinatorial explosion, generating the possible interleavings over the entire program will quickly become infeasible. To limit this problem, a test-case explicitly defines a test region with the use of predefined macros. The simulator will only interleave the instructions within that region, and outside of the region anything can run freely. This can be done, in principle, without loss of correctness, as long as the code outside of the region does not access the shared memory locations which are used inside the region[2].

[2]This is a too-strong property, as we only really require the input-state to be deterministic, and thus setting up local and global state could be allowed. But it is safer to continue with the above precondition, because in most cases there is no need for setting up inside the thread itself.

To determine the length of the test region, the number of instructions in any given path through the region has to be counted. Specifically, we need to count the number of instructions in the longest path from region start to region end. This ensures that any interleaving we generate will cover all possible paths through the region. We note that over-counting for a specific execution path is non-problematic: we can stop executing instructions once we have reached the end of the region. Therefore, we can count the longest path possible, and generate the interleavings from there.

The number of total interleavings N(t) for a number of threads t that can be generated is given by the following combinatorial formula, where n_i is the number of instructions (never fewer than 1) in thread i (starting from 1):

$$N(t) = \prod_{i=1}^{t} \binom{\sum_{j=1}^{i} n_j}{n_i}$$

We can prove its correctness with induction. Basis: we expect the result to be 1, as with a single thread there is only one ordering that can be made: running the thread from start to finish.

$$N(1) = \prod_{i=1}^{1} \binom{\sum_{j=1}^{i} n_j}{n_i} = \binom{n_1}{n_1} = 1$$

Structural Induction: we note that in order to acquire the ordering of an additional thread, we can use the orderings of the previous step, and for each ordering insert the instructions of the added thread at all possible combinations of positions. Thus, when adding a thread q, for each existing ordering we add $\binom{n_{subtotal}}{n_q}$ orderings, where $n_{subtotal}$ is the total number of instructions on the threads added up until now (including the current): $n_{subtotal} = \sum_{j=1}^{q} n_j$. Then the total number of orderings is given by:

$$\begin{aligned} N(k+1) &= N(k) * \binom{\sum_{j=1}^{k+1} n_j}{n_{k+1}} \\ &\overset{I.H.}{=} \prod_{i=1}^{k} \binom{\sum_{j=1}^{i} n_j}{n_i} * \binom{\sum_{j=1}^{k+1} n_j}{n_{k+1}} \\ &= \prod_{i=1}^{k+1} \binom{\sum_{j=1}^{i} n_j}{n_i} \end{aligned}$$

4.3.2 Executing a specific interleaving

Once we have generated all interleavings, we need to be able to execute them. Given an interleaving like (1,2,3,1,2,3,1,2,3), we would first let thread 1 execute a first instruction, then thread 2, then thread 3, then thread 1 again, and so on until we finish the order. For this purpose, all threads need to stop executing at the start of the region, and then execute instructions step-by-step as the order requires.

Our goal does not require the support of all types of loops. This is an intentional choice. Having a variable instruction length makes it impossible to determine all the interleavings that we can take in advance. Additionally, there should be no need for loops to exist within the simple test-cases: if we need repeating behaviour, we can manually unroll the loop. However, we do need 'waiting' loops, since atomics only guarantee *memory ordering*, but not *control*. Thus, 'waiting for' an atomic to update its value is impossible unless the value is constantly checked by the use of a loop.

Waiting loops require a system that determines whether a loop is currently being executed, and for each step that it should take the thread should execute the entire loop exactly once. This ensures that the thread inside a loop can exit the loop as soon as possible (as permitted by the ordering), and rejoin the normal interleaving. As the thread exits the loop, it can execute its normal interleaving again. Since this approach means that a thread would consume its step in the ordering, we append the step to the ordering again. An example of this (on a higher level) can be seen in Figure 4.2.

Shared Variables

```
atomic_bool flag = 0;
int n = 0;
```

Thread 1

```
1 n = 42;
2 atomic_store(&flag, true);
3 return n;
```

Thread 2

```
1 while(!atomic_load(&flag))
2     ;
3 int y = n;
4 return y;
```

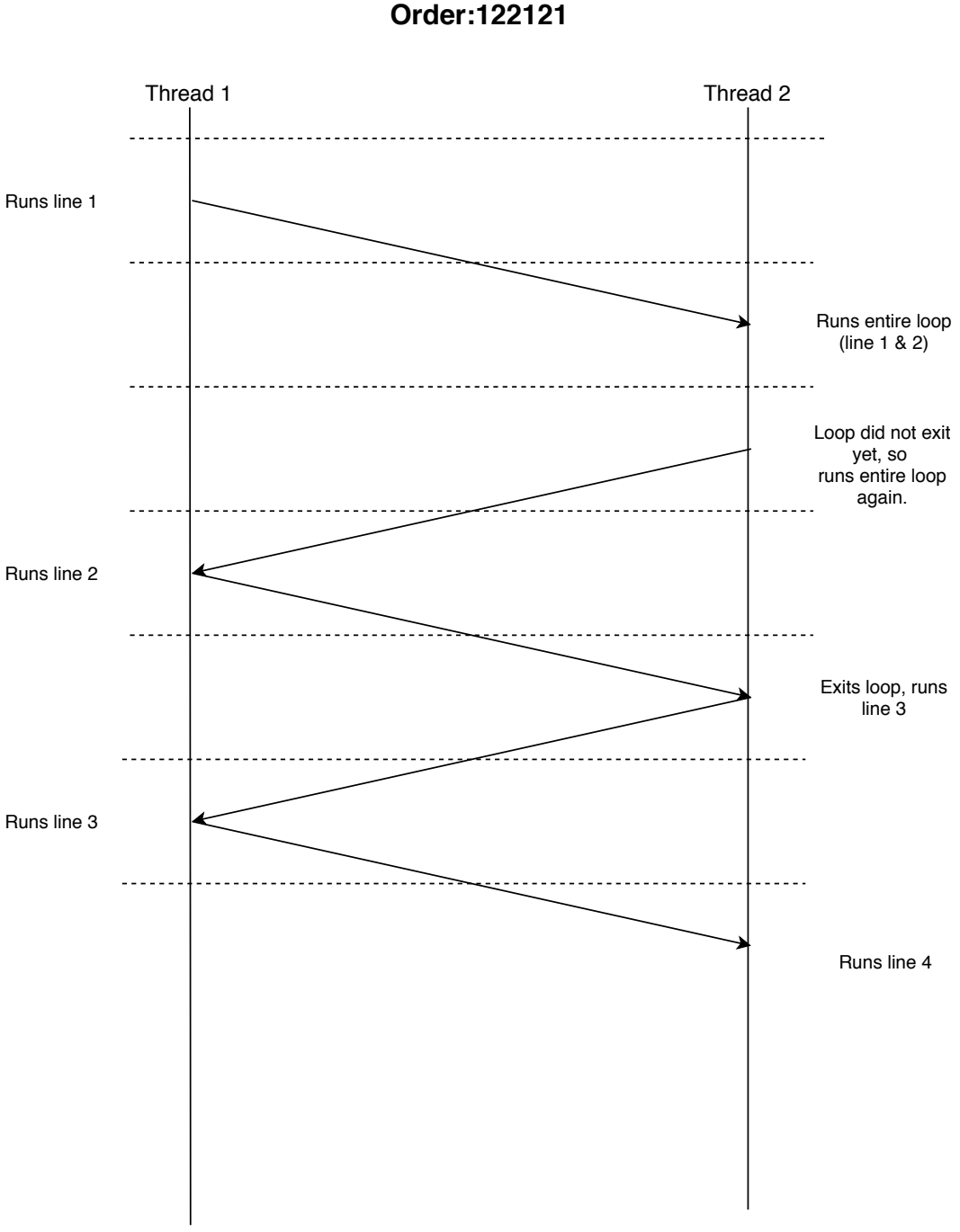

Figure 4.2: An example of an interleaving being executed through the simulator. For visualization purposes, it takes a line-by-line approach, where in reality it takes an instruction-by-instruction approach. Visualized here is how the order 122121 executes by controlling the threads. Each arrow is a context switch. Because the loop runs twice, an additional step of thread 2 has to be done at the end to finish the execution.

4.4 Implementation

Our approach requires step-by-step evaluation of a part of the program we are evaluating. A debugger allows for exactly this kind of functionality. Since we are looking to test C, we opted for one of the more widely used and open source debuggers for these languages: the GNU Debugger (GDB). We implemented the methodology in GDB using Python. In this section, we discuss the GDB specific considerations we had to take for the implementation. We present a schematical representation of our implementation in Figure 4.3.

4.4.1 GDB

A program is compiled down into an executable, which can be represented using assembly instructions. A debugger can insert a *trap* instruction at a specific address. The trap instruction is an interrupt. When executed, this instruction returns control to the debugger. The debugger is then able to change the code in the live environment, edit or inspect memory that is associated with the process and run arbitrary code in the (memory) context of the program [22]. In the context of a debugger, the trap instructions are placed at locations called 'breakpoints'. The moment a thread of execution *traps*, the program space is copied to a different location, much like a multi-thread context switch. All registers and the program code are thus stored in a memory location dedicated to holding stalled or paused threads. A debugger can debug a program by altering its code and operating in this same memory space. It can inject instructions and new trap instructions while the thread is parked in this memory space, but it can also query and change memory values in the registers and main memory.

When multiple threads are running, breakpoints function in much the same way as single-thread debugging. In GNU Debugger (GDB), a difference with single-thread debugging is that *all* threads of executions are halted as soon as a single thread breaks, which helps the user determine the current state in all threads (and thus state does not change while the user is debugging) [23]. When continuing the execution stream, all threads resume execution as well [24]. GDB provides a setting called scheduler-locking, which allows the debugger to only resume a single thread. In other words, when all threads break and the debugger tells the program to continue, only the thread that the debugger is currently inspecting is continued, while the rest remain stopped. This is a property we use to step instruction-wise on multiple threads, without the threads breaking free from our control.

GDB has commands such as `nexti` and `stepi` that allow instruction-wise stepping through threads. To utilize these instructions, we enable GDB's `scheduler-locking` option which allows for stepping on a single thread. Without this option, all threads run freely whenever the execution is continued. This is undesirable, because we want to explicitly run the threads of interest step-by-step, to achieve the targeted ordering.

4.4.2 Counting instructions

We make use of the GDB facilities 'write' and 'read' watchpoints that break the execution of the program whenever a variable is stored and read respectively. Our region indicators are set up in such a way that they write into a (globally unique volatile) variable at region start, and read from the same variable at region end. Using the watchpoints at these locations, we are able to stop at the exact first instruction in the region, and we are notified at the exact instruction when we leave the region.

Most compilers have a way to specify a compile-time barrier where instructions are not reordered around. In GCC and Clang, this is done by using `__asm__ __volatile__("" :: : "memory");`, which does not insert any additional instructions, but forces the compiler to not reorder any memory operations around this barrier. In ICC, the equivalent is `__memory_barrier();`. Adding these barriers after the region start and before the region end ensures that instructions will not be moved outside of the test region.

After hitting the entry watchpoints, we step instruction-wise until the ending watchpoint is hit, and increment the instruction count for each step[3].

Processors only support a limited number of hardware watchpoints, and we need one hardware watchpoint per thread at any given time. We can add the region end watchpoints after all region start watchpoints have been hit. While there are also software watchpoints, these are less precise: they break at the statement-level rather than instruction-level, and only break on the thread that is being viewed by

[3]We could have also used labels to track the regions, instead of applying watchpoints. However, labels can be freely removed by the compiler if unused. In contrast, our *volatile* tagged variables ensure that a compliant compiler will not remove the watchpoints.

Type	Instructions
Conditional	ja, jae, jb, jbe, jc, jcxz, jecxz, jrcxz, je, jg, jge, jl, jle, jna, jnae, jnb, jnbe, jnc, jne, jng, jnge, jnl, jnle, jno, jnp, jns, jnz, jo, jp, jpe, jpo, js, jz, loop, loope, loopne, loopcc
Jump	jmp

Table 4.1: The x86 instructions we use as indicators for decision points. We derived this set of instructions from the Intel instruction set reference [26].

the debugger [25]. Thus, when the threads are running freely, they will not stop at the region entrance, because the debugger is not viewing the threads. We therefore need hardware watchpoints. For the approach where each thread inserts their watchpoints at the same time with a processor only supporting four hardware watchpoints, only up to three threads are testable.

We worked around this limitation by observing that breakpoints are not as limited as watchpoints. Thus, we can break earlier in the function with the use of breakpoints, and then walk each thread one by one to the starting region watchpoint. This means that we only need 2 watchpoints at any given time. To simplify the execution later on, we record the exact address of the region start and region end for each thread, such that precise breakpoints can be added.

Counting the number of orderings for conditional programs is more difficult, since the number of instructions is dependent on the execution path. We count recursively, by splitting the counters into separate searchers at every decision point, much like depth-first-search. One of the counters continues counting at the next instruction, while the other counter takes the jump and continues there.

A decision point is defined by its instruction. We defined the instructions that represent a decision point for x86 in Table 4.1. We distinguish between conditional instructions and loop instructions. The former are decision points, and thus split the counter into separate searchers. The latter continue the execution at a different location. We only split the execution at conditional instructions.

We detect loops by checking if the target address of *any* of the instructions in Table 4.1 have already been visited (i.e., if any of the target addresses of conditional or jump instructions have been visited). If so, we mark the target instruction as a LOOP, and annotate it with the address that would have been visited if the conditional was not taken. The runtime can use this address as an indicator that the loop has been broken out of, and determine whether it is running a loop using the LOOP annotation.

4.4.3 Execution

The entire program can run freely until the region start breakpoints are hit on all threads. Then we can continue to interleave step-by-step until all threads have exited the region. We then immediately test the outcome of the region by calling an user-defined function. And finally, we let the threads run free until the program has exited.

When the execution encounters a LOOP annotation, it adds a breakpoint at the current instruction and at the address that the LOOP was annotated with. For a single iteration, the thread can then continue until it hits either breakpoint. If it remains inside the loop, it repeats this process on the next instruction invocation. If it escapes the loop, it continues execution as normal.

With this mechanism, we can give the simulator any interleaving, and it will execute exactly that ordering of instructions. It can then run a test and report its result. Thus, we can also identify precisely which orderings fail the test-case, and which orderings pass.

The simulator may not work as well on all compiler configurations. For example, for the optimization settings lower than `-O3`, icc generates a full function call sequence for each `atomic_store`. This significantly increases the number of instructions, and in turn makes the number of orderings infeasible. To keep the number of instructions and orderings within a feasible range, appropriate compiler configurations are required.

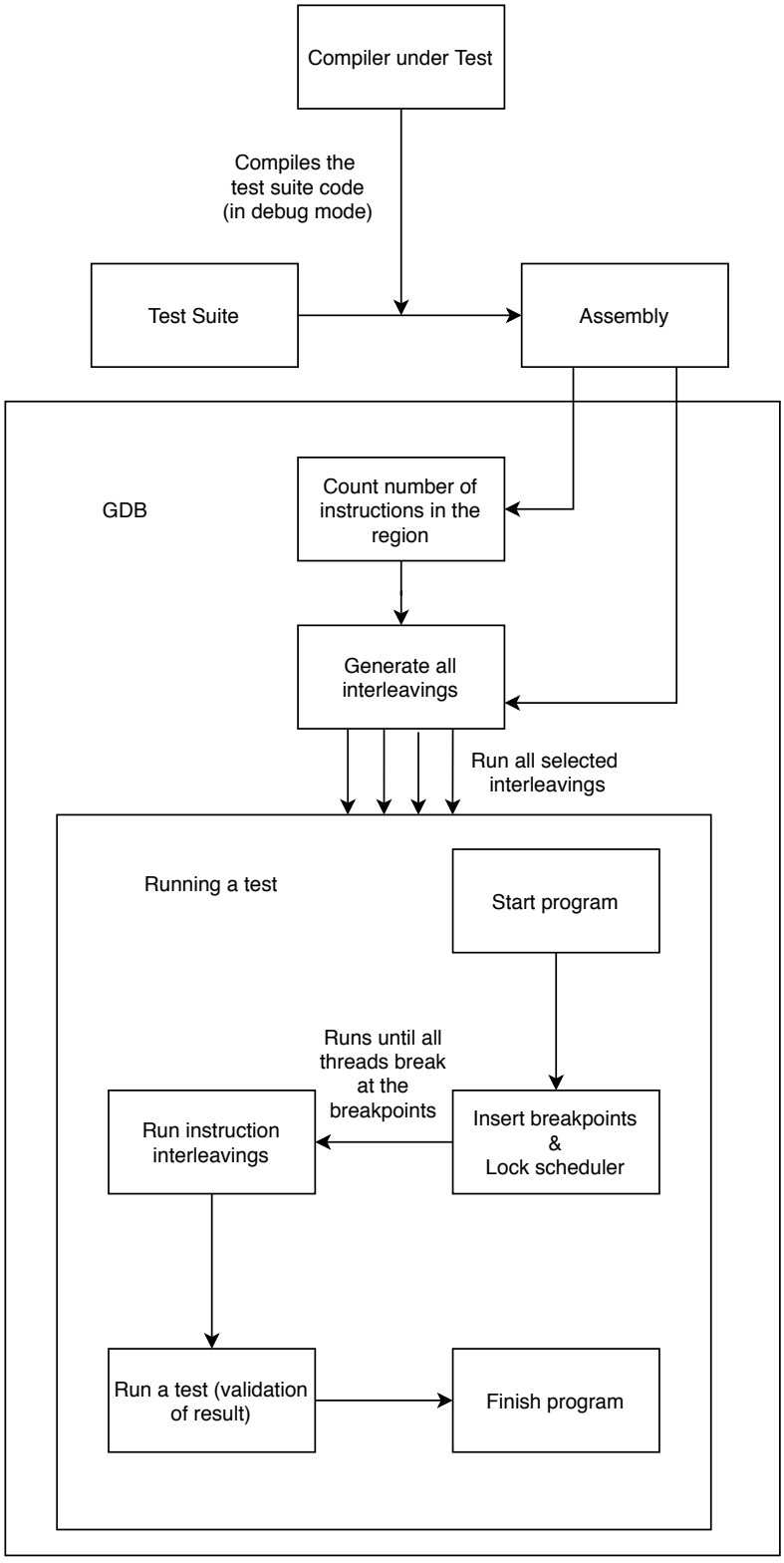

Figure 4.3: A schematic approach of the implementation of the simulator. First the compiler compiles the test-case. Then, we apply GDB to count the number of instructions in the test regions. From the number of instructions in each thread we generate all possible interleavings. Each interleaving is then run in GDB, where all threads move up until the region, then execute the interleaving, run the test-case and finish the execution.

Chapter 5

The Test-Suite

In this chapter, we present the test-suite that we systematically derived from the literature. We start with an overview and an explanation on how every test-case is structured. Then, we present and discuss the test-cases for each class of tests.

5.1 Overview

Our test-suite was designed by specifically looking at code samples from various sources [10, 27, 28], and deriving correct behaviour based on the C concurrency specification. We organize our tests into three classes: atomicity, thin air, and total modification order.

We slightly edit the C syntax used in this section for conciseness. In C, the atomic functions have both a non-explicit variant, were the memory order is sequential consistent, and an explicit variant where the memory order can be specified. We leave out the `_explicit` postfix in the examples, and always mention which exact memory order is used for the functions. We will also leave out the `memory_order_` prefixes. As an example, the C syntax `atomic_fetch_add_explicit(&x, 1, memory_order_seq_cst)` becomes `atomic_fetch_add(&x, 1, seq_cst)` in our figures.

We display each test-case in four parts. First, we show the shared variables with their initial values. Second, we show the C code that runs within the region on each thread. Third, we show the assertions we make on the outcome of the test, after all threads have run in some order. Fourth, and final, we show the derivation of these assertions in a table. We enumerate all states the system can be in (derived from the constants and operators in the program) and display for each case whether it would be valid or not. For the valid cases, we show a thread interleaving that could give us this result. For the invalid results, we describe in text why this result is invalid.

5.2 Atomicity

Atomicity tests are the tests that ensure that atomics are *atomic* - that is, indivisible operations. Atomic operations on the same object should be ordered according to their modification order. Therefore, there should be no data races. We also look at the effect of sequential consistent fences and any atomics surrounding them.

Basic operations (0_atomic)

Shared Variables

```
atomic_int x = 0;
```

Thread 1	Thread 2
`atomic_fetch_add(&x, 1, relaxed);`	`atomic_fetch_add(&x, 2, relaxed);`
`atomic_fetch_add(&x, 3, relaxed);`	`atomic_fetch_add(&x, 4, relaxed);`
`atomic_fetch_add(&x, 5, relaxed);`	`atomic_fetch_add(&x, 6, relaxed);`

Asserts

```
atomic_load(&x, seq_cst) == 21;
```

Value of x	Is result valid	Example thread execution order that leads to result
0	×	×
1	×	×
2	×	×
3	×	×
4	×	×
5	×	×
7	×	×
8	×	×
9	×	×
10	×	×
11	×	×
12	×	×
13	×	×
14	×	×
15	×	×
16	×	×
17	×	×
18	×	×
19	×	×
20	×	×
21	✓	$1 \rightarrow 2$.

We constructed this sample ourselves. An operation should never dismiss the value of another[1]. Thus, in this sample, the result is always 21, regardless of ordering of the operations.

We could change the operator (i.e., add) with any other, and replace the memory order with any stronger one as well. Regardless, due to the atomic behaviour, the result should be the same as if all operations were performed in an order consistent with the *modification order*.

[1] As defined in the standard: "Atomic read-modify-write operations shall always read the last value (in the modification order) stored before the write associated with the read-modify-write operation." [8]

No indivisible writes (1_rwc)

Shared Variables

```
atomic_int x = 0;
atomic_int y = 0;
int r1 = 0;
int r2 = 0;
int r3 = 0;
int r4 = 0;
```

Thread 1

```
1 atomic_store(&x, 1, relaxed);
```

Thread 2

```
1 atomic_store(&y, 1, relaxed);
```

Thread 3

```
1 r1 = atomic_load(&x, relaxed);
2 atomic_fence(seq_cst);
3 r2 = atomic_load(&y, relaxed);
```

Thread 4

```
1 r3 = atomic_load(&y, relaxed);
2 atomic_fence(seq_cst);
3 r4 = atomic_load(&x, relaxed);
```

Asserts

```
r1 == 0 || r1 == 1
r2 == 0 || r2 == 1
r3 == 0 || r3 == 1
r4 == 0 || r4 == 1

NOT (r1 == 1 && r2 == 0 && r3 == 1 && r4 == 0)
```

r1	r2	r3	r4	Valid	Execution example (threads)
0	0	0	0	✓	$3 \to 4 \to 1 \to 2$.
0	0	0	1	✓	$3 \to 1 \to 4 \to 2$.
0	0	1	0	✓	$3 \to 2 \to 4 \to 1$.
0	0	1	1	✓	$3 \to 1 \to 2 \to 4$.
0	1	0	0	✓	$4 \to 2 \to 3 \to 1$.
0	1	0	1	✓	3 (until line 2) $\to$ 4 (until line 2) $\to 1 \to 2 \to 3 \to 4$.
0	1	1	0	✓	$2 \to 3 \to 4 \to 1$.
0	1	1	1	✓	3 (until line 2) $\to 1 \to 2 \to 4 \to 3$.
1	0	0	0	✓	$4 \to 1 \to 3 \to 2$.
1	0	0	1	✓	$1 \to 3 \to 4 \to 2$.
1	0	1	0	×	×
1	0	1	1	✓	$1 \to 3 \to 2 \to 4$.
1	1	0	0	✓	$4 \to 1 \to 2 \to 3$.
1	1	0	1	✓	$1 \to 4 \to 2 \to 3$.
1	1	1	0	✓	$2 \to 4 \to 1 \to 3$.
1	1	1	1	✓	$1 \to 2 \to 3 \to 4$.

This sample is derived from Boehm et al. [27]. In this test-case, a thread writes to the shared atomic variable `x`, and another writes to the shared atomic variable `y`. Two other threads read the values in the shared variables, with a fence in between to prevent reordering. We specifically test the total order property of sequentially consistent fences here. Thread 3 and 4 should observe the same order of assignments. The fence enforces the *sequenced before* relations here, so there is no reordering involved.

Any `r*` variable could only be set to 0 or 1, therefore we determine the valid states by discussing all possible combinations of 1s and 0s[2]:

[2]This is a consequence of the standard: "An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations." [8]

We provide some intuition to the invalid state (1010). Due to sequential consistency, a total order has to be shared for all threads. Thus, either the write `x = 1` is ordered before `y = 1`, or vice versa. All threads should share the same view of this order, so it cannot be that one thread observes `x` being written to before `y`, and the other sees `y` being written before `x`. When `r1 = 1`, `r2 = 0`, `r3 = 1`, `r4 = 0`, this is exactly what happens: thread 3 sees `x` being written before `y`, while thread 4 sees `y` being written before `x`, thus violating the total order property.

Write-read consistency (2_wrc)

Shared Variables

```
atomic_int x = 0;
atomic_int y = 0;
int r1 = 0;
int r2 = 0;
int r3 = 0;
```

Thread 1

```
1 atomic_store(&x, 1, relaxed);
```

Thread 2

```
1 r1 = atomic_load(&x, relaxed);
2 atomic_fence(seq_cst);
3 atomic_store(&y, 1, relaxed);
```

Thread 3

```
1 r2 = atomic_load(&y, relaxed);
2 atomic_fence(seq_cst);
3 r3 = atomic_load(&x, relaxed);
```

Asserts

```
r1 == 0 || r1 == 1
r2 == 0 || r2 == 1
r3 == 0 || r3 == 1

NOT (r1 == 1 && r2 == 1 && r3 == 0)
```

r1	r2	r3	Valid	Execution example (threads)
0	0	0	✓	$3 \to 2 \to 1$.
0	0	1	✓	3 (until line 2) $\to 2 \to 1 \to 3$.
0	1	0	✓	$2 \to 3 \to 1$.
0	1	1	✓	$2 \to 1 \to 3$.
1	0	0	✓	$3 \to 1 \to 2$.
1	0	1	✓	$1 \to 3 \to 2$.
1	1	0	×	×
1	1	1	✓	$1 \to 2 \to 3$.

This sample is also derived from Boehm et al. [27]. Due to the sequentially consistent fence, the *happens before* relation should be preserved. The write to y in thread 2 cannot *happen before* the read of `x` into `r1`. All threads should therefore observe that order. If thread 2 reads 1 from `x`, and then writes `y`, all other threads now must read 1 from `x` if they read 1 from `y`, otherwise they can observe that the write to `y` happened before the write to `x`, which is violating the *happens before* property.

Read-write consistency (3_rwc)

Shared Variables

```
atomic_int x = 0;
atomic_int y = 0;
int r1 = 0;
int r2 = 0;
int r3 = 0;
```

Thread 1

```
1 atomic_store(&x, 1, relaxed);
```

Thread 2

```
1 r1 = atomic_load(&x, relaxed);
2 atomic_fence(seq_cst);
3 r2 = atomic_load(&y, relaxed);
```

Thread 3

```
atomic_store(&y, 1, relaxed);          1
atomic_fence(seq_cst);                 2
r3 = atomic_load(&x, relaxed);         3
```

Asserts

```
r1 == 0 || r1 == 1
r2 == 0 || r2 == 1
r3 == 0 || r3 == 1
NOT(r1 == 1 && r2 == 0 && r3 == 0)
```

r1	r2	r3	Valid	Execution example (threads)
0	0	0	✓	$2 \to 3 \to 1$.
0	0	1	✓	$2 \to 1 \to 3$.
0	1	0	✓	$3 \to 2 \to 1$.
0	1	1	✓	3 (until line 2) $\to 2 \to 1 \to 3$.
1	0	0	×	×
1	0	1	✓	$1 \to 2 \to 3$.
1	1	0	✓	$3 \to 1 \to 2$.
1	1	1	✓	$1 \to 3 \to 2$.

In this sample, also derived from Boehm et al. [27], instructions cannot be reordered over the sequentially consistent fence. Therefore, if thread 2 sees `r2` as 0, then thread 3 must not have passed the fence yet. If `r1` has read 1 before, `r3` must read 1 from `x` as well.

Of interest here are again the threads 2 and 3. Thread 2 reads from `x`, and then reads from `y`. Thread 3 writes 1 into `y`, and then read from `x`. If thread 2 reads 0 from `y`, it executed before the write into `y` from thread 3 has become visible. Thus, if thread 2 then had also read 1 from `x`, thread 3 should read `x` as 1. If this is violated, it violates the *happens-before* relation.

5.3 Thin air tests

Compilers and hardware can potentially introduce speculative writes and reads, where a value that is *likely* valid is written in advance, and only its validity is checked afterwards (where, if invalid, corrective actions are taken) [10]. These speculative reads and writes tend to happen around conditional branches (such as if-statements). In multi-threaded environments, these speculations are illegal as they can cause out-of-thin-air values[3]. For example, when a system assumes that a variable will take the value 1, it stores that value momentarily, after which other threads can take that value and do their actions with it. Only later is the invalid value reverted, but the invalid value has already propagated onto the other threads.

[3] An out-of-thin-air value is an arbitrary value that is assigned to a register or variable, generally due to branch or value prediction. This value could not have been assigned through normal execution.

Thin air (4_thinair)

Shared Variables

```
atomic_int x = 0;
atomic_int y = 0;
int r1 = 0;
int r2 = 0;
int r3 = 0;
```

Thread 1

```
1 r1 = atomic_load(&x, relaxed);
2 atomic_store(&y, r1, relaxed);
```

Thread 2

```
1 r2 = atomic_load(&y, relaxed);
2 atomic_store(&x, r2, relaxed);
```

Asserts

```
r1 == r2 == 0
```

r1	r2	Valid	Execution example (threads)
0	0	✓	$1 \rightarrow 2.$

In this case, derived from the C standard [8], the threads both load the value in the register, and store the value from the register in the other variable. As with the previous samples, it is therefore guaranteed to result in an end state where both `r1` and `r2` are 0. Any other value comes from thin-air and is invalid.

Thin air (5_thinair)

Shared Variables

```
atomic_int x = 0;
atomic_int y = 0;
int r1 = 0;
int r2 = 0;
```

Thread 1

```
1     r1 = atomic_load(&x, relaxed);
```

Thread 2

```
1 r2 = atomic_load(&y, relaxed);
2 atomic_store(&x, r2, relaxed);
```

Thread 3

```
1     atomic_store(&y, 1, relaxed);
```

Asserts

```
r1 == 0 || r1 == 1
r2 == 0 || r2 == 1

NOT(r1 == 1, r2 == 0)
```

r1	r2	Valid	Execution example (threads)
0	0	✓	$1 \rightarrow 2 \rightarrow 3.$
0	1	✓	$1 \rightarrow 3 \rightarrow 2.$
1	0	×	×
1	1	✓	$3 \rightarrow 2 \rightarrow 1.$

This sample is derived from the Java causality test-cases [29]. In this sample it is more likely for the system to assume a thin-air state because the constant is actively used by other threads, and there are

valid executions where the value can be assumed. `r1` cannot be 1 when `r2` is 0, because `r2` is the bridge between the value 1 of `y` and `x`. `r1` can therefore only be loaded as the value 1 if `r2` was. Thus, if we load `r1` as 1 and `r2` as 0, a thin-air prediction happened.

Thin air array (6_thinair_array)

Shared Variables

```
atomic_int x = 0;
atomic_int y = 0;
int r1 = 0;
int r2 = 0;
int r3 = 0;
int a[2]; a[0] = 1; a[1] = 2;
```

Thread 1

```
r1 = atomic_load(&x, relaxed);
a[r1] = 0;
r2 = a[0];
atomic_store(&y, r2, relaxed);
```

Thread 2

```
r3 = atomic_load(&y, relaxed);
atomic_store(&x, r3, relaxed);
```

Asserts

```
r1 == 0 || r1 == 1
r2 == 0 || r2 == 1
r3 == 0 || r3 == 1

NOT(r1 == 1 && r2 == 1 && r3 == 0)
```

r1	r2	r3	Valid	Comments
0	0	0	✓	$1 \rightarrow 2$.
0	0	1	×	Something had to have assigned 1 to `y`, but `r2` is 0.
0	1	0	×	Thread 1 would have to keep `a[0]` as 1, which can only be done by setting `r1` to 1, or illegally transforming the code.
0	1	1	×	Same as above.
1	0	0	×	Something had to set `x` to 1, but `r3` and default are 0.
1	0	1	×	If `r1` is 1, `r2` cannot be 0 because `a[r1]` = 0 would affect `a[1]`, and not `a[0]`, thus `r2` would be 1.
1	1	0	×	Same as 1, 0, 0: `r1` cannot be set to 1 without `r3` or its default value being 1.
1	1	1	×	Internally consistent, but this can only happen as an effect of misprediction and self-fulfillment: as the first step, either `r1` or `r3` must load 0.

This sample was also derived from the Java causality test-cases [29]. Here, the use of arrays allow for different kinds of predictions. The only allowed state is when `r1`, `r2` and `r3` are 0. In the table we discuss reasons why each observed effect cannot take place. The final case, where all registers equal 1, is in principle self-consistent, but can only occur when one of two atomic loads loaded 1. The default states promise 0, and therefore a thin-air load has to have happened.

5.4 Total modification order

Tests for total modification order are concerned with the fact that atomic operations (and fences) order the non-atomic operations around them. As a consequence, well-defined behaviour is possible, even for shared variables over multiple threads, as long as the total modification order is preserved. These code samples come from Boehm [28].

TMO - Sequential consistent (7_tmo)

Shared Variables

```
atomic_bool x = false;
int n = 0;
int r1 = 0;
```

Thread 1

```
1 n = 23;
2 atomic_store(&x, 1, seq_cst);
```

Thread 2

```
while (!atomic_load(&x, seq_cst))
    ;
r1 = n;
```

Asserts

```
r1 == 23
```

r1	Valid	Execution example (threads)
0	×	×
23	✓	$1 \rightarrow 2$.

In this test-case, we test whether the atomic store ensures that non-atomic stores that are *sequenced-before* the release operation (i.e., atomic store in thread 1) also *happen-before* the atomic. Thus, in thread 2, we test whether `n` contains the value 23 after loading `x`. The values are either 0 or 23. `r1` cannot assume 0, as thread 2 waits for thread 1 to store 1 into `x`. The release-acquire semantics of the store and load guarantee that the non-atomic variable `n` has also fully completed its store before `x` becomes visible. Thus, `r1` must read 23.

TMO - Release acquire (8_tmo_weaker)

Shared Variables

```
atomic_bool x = false;
int n = 0;
int r1 = 0;
```

Thread 1

```
1 n = 23;
2 atomic_store(&x, 1, release);
```

Thread 2

```
while (!atomic_load(&x, acquire))
    ;
r1 = n;
```

Asserts

```
r1 == 23
```

r1	Valid	Execution example (threads)
0	×	×
23	✓	$1 \rightarrow 2$.

This case is the same as the previous test-case, but in this case expressed with the weakest possible correct memory order (`release + acquire`). This case is included to display the differences in generated code between the `seq_cst` code and the `release + acquire` code.

Chapter 6

Evaluation

In this chapter, we focus on the evaluation of our test-suite as a tool for assessing a compiler's standard-compliance. Specifically, we show how we assess different compilers with our test-suite; we further use an in-house method for fault injection to assess whether our method can outperform a naive testing approach in completeness.

6.1 Evaluation method

We evaluate our method by testing various compilers. We do this in four steps. For every compiler, we: (1) apply the simulator on the tests compiled with the compiler; (2) run a naive test on the compiler by running the program manually for 50000 times; (3) strip away atomic and multi-threading hints from the test, and run it with the simulator; and (4) run the naive test without multi-threading hints for 50000 times.

The first step functions as a normal test with our simulator. We compile the code, run the test using our simulator, and collect the results. This step may give us insights on: (a) bugs in our simulator by finding issues in its execution; (b) feasibility of the test on this compiler due to total interleaving count; and (c) potential compiler bugs.

The second step is a verification of the simulator, to determine whether we can *at least* find the standard violations that a naive test can. The naive test relies on the scheduler to yield different interleavings, and is invoked by repeated execution of the program. The naive test is therefore a random, non-coverage guided approach. It is representative of a real-world execution of the program, because it runs the program natively. If the naive test displays behaviour that our simulator cannot evoke, the simulator is unable to represent all real interleavings correctly. This means that either the simulator was implemented incorrectly, or that by explicitly ordering the instructions, only a subset of the real behaviours are kept. Vice versa, if we are able to find more incorrect executions with the simulator than with a naive execution, we show the value of applying our method on these test-cases.

The third step is stripping the test-cases from atomics and fences. This is our way of injecting faults by tricking compilers into doing invalid transformations, and determining if our simulator and test suite can find this. This alleviates the fact that for the first and second step there may be no errors in the actual implementation. In that case it is difficult to compare our simulator with a naive test, and thus implicitly breaking the code by allowing compilers to do single-threaded optimizations is more likely to introduce faults.

The fourth step is the same as the second step, except applied on the intentionally incorrect code as described in the third step.

Additional to the systematic approach to evaluation, we use an opportunistic approach where we apply manual transformations on the code to draw out relevant cases, and to check whether the simulator can detect the transformations the tests are protecting against, in the case that the compilers do not display this behaviour already.

6.2 Setup

We run everything on an Intel Core i5-6200U CPU. It contains two equivalent cores, in total able to execute four threads. This CPU is based on the x86 architecture with the Skylake microarchitecture.

Compiler	Version
icc	19.1.1.219 20200306
GCC	(Ubuntu 9.3.0-10ubuntu2) 9.3.0
clang	10.0.0-4ubuntu1 (Target: x86_64-pc-linux-gnu, Thread model: posix)
musl-gcc	(Ubuntu 9.3.0-10ubuntu2) 9.3.0

Table 6.1: Versions of the compilers used.

We use four compilers to evaluate. We choose GCC and Clang for their public availability: they are open source and popular choices as a compiler. We choose the Intel C compiler (icc) as it is developed by Intel and therefore may exploit our architecture more interestingly than GCC and Clang. Finally, we choose musl-GCC. musl-GCC uses GCC as the compiler, but has its own library 'musl' (not based on glib like GCC and Clang). The versions of each compiler is shown in Table 6.1.

We use the same flags for every compiler: `-static -g -O3 -std=c18 -pthread`. We use `-static` to statically preinclude the pthread library (if required), this prevents excessive startup time and thus significantly speeds up the execution of the test suite (as the programs are started very frequently). We use `-g` to instruct the compiler to include debugging annotations, which we need for GDB to function correctly. We use `-std=c18` to hint compilers that we are using the C18 standard for semantics. Finally, we use `-pthread` to link the pthread library for the compilers that use pthread as their threading backend (GCC, Clang and icc).

6.3 Region indicators

In the early phases of our experiments, we found that our region indicators were not strong enough, causing icc to either optimize the region indicators away, or to merge parts of the code inside the region with the preamble of the region. As discussed in the implementation, we prevent code from leaking out of the region by adding compiler barriers to the region indicators, and tagging the region indicator variables with `volatile`.

For each of the tests, we manually verify whether the code that should be inside of the region does not leak out. With the compilers that we test, we find that none violate this requirement. However, applying this on other compilers could have slightly different behaviour, since they may apply different types of transformations. When using the simulator for the first time on a different compiler, it is therefore recommended to check whether the instructions that are expected to fall within the region do not leak out. Generated code, and especially the debugger annotations, differ per compiler, so defining an uniform way of approaching this is difficult. In our experience, the debugger annotations include line numbers which can be used to determine whether the code falls before or after the region. Otherwise, verifying that no instructions are moved out of the region is done by mapping the assembly code back to the C instructions and determining if the relevant code falls within the region.

6.4 Findings

We first summarize our findings, and then dive into more details on the analysis that has led us to each of these findings. For the full evaluation results, refer to Appendix A.

Finding 1: We find that our simulator is unable to detect cases where behaviour relies on same-time execution of two threads.

Finding 2: The naive test is unable to detect some standard violations that our simulator can detect. Our simulator helps to deterministically find specific interleavings that are unlikely to happen in a naive test scenario.

Finding 3: The compilers are too cautious when it comes to transformations with atomics and fences.

Finding 4: We find no standard violations when it comes to atomics and fences in the compilers that we tested.

6.4.1 Atomicity

For Finding 1, we executed the first test in our test suite. We see all compilers succeed in this test (on x86) with and without our simulator. The GCC compiler generates the code:

```
THREAD 1:
lock addl       $1, x(%rip)
lock addl       $3, x(%rip)
lock addl       $5, x(%rip)
```

```
THREAD 2:
lock addl       $2, x(%rip)
lock addl       $4, x(%rip)
lock addl       $6, x(%rip)
```

In this case, the 'lock' ensures the atomic behaviour.

When we remove all atomic hints in the code, we see the code change to an optimized form:

```
THREAD 1
addl   $9, x(%rip)
```

```
THREAD 2
addl   $12, x(%rip)
```

We see that the three separate additions have been merged into one addition with a precomputed result. We find that similar transformations are not taken by the compiler when faced with atomics and fences (even on `-O3`). This might be because the atomic behaviour implies that it needs to be visible to other threads at some point (so also the intermediate state has to be represented at some point). Merging the instructions in this case has no effect on the semantics whatsoever.

We also see that, due to the lack of `lock`, this is a data race. We should see this behaviour in the results of our test: we should see any of 9, 12 or 21 as output state, violating the requirement that the output has to be 21. When we run the test naively 50000 times, we do indeed already see that the result is sometimes 9 or 12, and therefore not accepted. When we run our simulator, we do not see this behaviour, and all test orderings succeed. This implies that our simulator cannot detect cases where this type of atomicity is violated.

The reason we are unable to detect this type of atomic behaviour is due to fact that the `+=` operator can be implemented with a single instruction (`addl`). This has the effect that by instruction-wise stepping, we are too *coarse*, as we cannot control the behaviour that is happening during the instruction. On the x86 architecture, a non-locked `addl` instruction can buffer a write in the the store buffer. This means that its effects are not immediately visible, thus any operations happening while the value has not been pushed to main memory, operate on the old value. We therefore seem unable to detect standard violations that rely on same-time execution of two instructions.

The explicit switching of threads from within the GDB context may be the cause of this detection failure. After executing an instruction an interrupt is fired to return control to GDB. At that point, we can decide to switch threads and step further. This "context switching" takes a significant amount of time, and executes a number of instructions on the CPU. What this entails is that during this time, the store buffer in the CPU is flushed (either explicitly by GDB, or implicitly due to the delay / additional actions). As the store buffer is flushed, the memory operation is automatically committed to memory and completed. Thus, our simulator is unable to detect a violation of non-divisible behaviour of atomics.

6.4.2 Consistency tests

Running the tests 2_wrc and 3_rwc without atomic hints show that our simulator is able to find interleavings where the test condition is violated, while a naive test does not. We display the result in Figure 6.1. After removing the atomic hints in 2_wrc, GCC and musl-GCC move the assignment of 1 to `y` in thread 2 before the load of `x` into `r1`. The removal of atomic hints causes 4 out of 280 orders to fail, which we found using our simulator. When we run the same program with a naive test we find no occurrence of this behaviour, resulting in Finding 2.

Similarly, on 3_rwc, we see icc moving the load of `x` into `r3` before the store of 1 into `y` on thread 3[1]. Our simulator then finds a fault with the order of loads and stores being incorrect in 6 out of 280 orders. Once again, if we run this with a naive test, we find no occurrence of this behaviour.

```
THREAD 1 (4202140)
0x401e9c: movl   $0x1,0xe2e9a(%rip)        # 0x4e4d40 <x>
THREAD 2 (4202204)
0x401edc: movl   $0x1,0xe2e62(%rip)        # 0x4e4d48 <y>
0x401ee6: mov    0xe2e54(%rip),%eax        # 0x4e4d40 <x>
0x401eec: mov    %eax,0xe2e5e(%rip)        # 0x4e4d50 <r1>
THREAD 3 (4202284)
0x401f2c: mov    0xe2e16(%rip),%eax        # 0x4e4d48 <y>
0x401f32: mov    %eax,0xe2e14(%rip)        # 0x4e4d4c <r2>
0x401f38: mov    0xe2e02(%rip),%eax        # 0x4e4d40 <x>
0x401f3e: mov    %eax,0xe2e00(%rip)        # 0x4e4d44 <r3>
Maximum orders: 280.0
Completed running 280 orders.
Number failed: 4.
One or more test execution orders failed:
[[2, 3, 3, 3, 1, 2, 2, 3], [2, 3, 3, 3, 3, 1, 2, 2], [2, 3, 3, 3, 1, 2, 3, 2], [2, 3, 3,
    3, 1, 3, 2, 2]]

Thread 2: <y> moved before <x> and <r1>

Default run:
bash -c "for run in {1..50000}; do bin/2_wrc >> out/fingcc_2_wrc.txt; done"  55,06s user
    49,84s system 74% cpu 2:20,76 total
Failed: 0 / 50000
Incorrect result states: None
```

Figure 6.1: GCC and musl-GCC apply invalid transformations on the atomic-stripped rwc_2. 4 out of the 280 execution orders fail. These standard violations are not observed by the naive approach.

6.4.3 Optimizations

For the test-case 1_rwc, we find that the generated code and execution are:

```
THREAD 1 (4202204)
movl   $0x1,0xe2e5a(%rip)        # 0x4e4d40 <x>

THREAD 2 (4202268)
movl   $0x1,0xe2e26(%rip)        # 0x4e4d4c <y>

THREAD 3 (4202332)
mov    0xe2dde(%rip),%eax        # 0x4e4d40 <x>
mov    %eax,0xe2ddc(%rip)        # 0x4e4d44 <r1>
mfence
mov    0xe2ddb(%rip),%eax        # 0x4e4d4c <y>
mov    %eax,0xe2ddd(%rip)        # 0x4e4d54 <r2>

THREAD 4 (4202412)
mov    0xe2d9a(%rip),%eax        # 0x4e4d4c <y>
mov    %eax,0xe2d90(%rip)        # 0x4e4d48 <r3>
mfence
mov    0xe2d7f(%rip),%eax        # 0x4e4d40 <x>
mov    %eax,0xe2d89(%rip)        # 0x4e4d50 <r4>


Maximum orders: 33264.0
Completed running 33264 orders.
Number failed: 0.
All test orders succeeded!
```

On thread 3 and 4, the atomic variables are read from in a relaxed manner, and a sequentially consistent fence ensures the memory ordering. Due to this running on a x86, the mfence is actually excessive: all atomic reads in thread 3 and 4 already happen in a sequentially consistent manner. If no

[1] icc also moves the load of y in thread 2 before the store into r1, but the load order is preserved so the result is valid

fence was requested and instead sequentially consistent atomics were used, the mfence would not appear. Thus, in this example, there was potential room for the compiler for an optimization that it did not take[2].

Particularly, if we explicitly insert another fence directly after the original fence, we observe that the compiler enters two mfence instructions, as shown in Figure 6.2. According to Vafeiadis and Nardelli [30], fences are redundant if (1) it follows a fence or a locked instruction with no store in between, or (2) it precedes a fence or locked instruction with no reads in between. In this case, the fences were directly after each other, so optimizing them away would have been completely legal. This result indicates that the compiler is hesitant to perform an optimization with these fences.

In all other assembly code we see no reordering of instructions, removal of mfences or other transformations, where in sequential code we see a number of transformations that could be valid in the multi-threaded domain as well. These results point towards Finding 3.

```
r3 = atomic_load(&y, seq_cst);
atomic_thread_fence(seq_cst);
atomic_thread_fence(seq_cst);
r4 = atomic_load(&x, seq_cst);
```

compiles to

```
mov 0xe2d9a(%rip),%eax  # <y>
mov %eax,0xe2d90(%rip)  # <r3>
mfence
mfence
mov 0xe2d7c(%rip),%eax  # <x>
mov %eax,0xe2d86(%rip)  # <r4>
```

Figure 6.2: Adding an additional sequential consistent fence in C results in two mfences in the assembly.

6.4.4 Bug free

On all tests with atomic hints, we see no violations of the test condition on any compiler. We do not see this on the naive test either, suggesting Finding 4: for the cases we tested, the compilers seem bug free.

However, on some test-cases (1_rwc, 2_wrc, 3_rwc), icc generated multiple instructions for each atomic, resulting in too many interleavings for our simulator to handle. Due to this, we cannot show or test icc's results for the atomic variants of those test-cases.

We also ran into warnings with the icc compiler while compiling the test-suite. An example is as follows:

```
warning #2330: argument of type "atomic_int={_Atomic(int)} *" is incompatible with
    parameter of type "const volatile void *" (dropping qualifiers)
      r2 = atomic_load_explicit(&y, memory_order_relaxed);
```

Even with the warnings, icc's compiled binary executes fine. The way to fix the warnings in icc is to replace all `atomic_int` with `int`, as icc expects the primitive types in the `atomic_*` functions. The warnings are in conflict with the standard, as the C standard explicitly defines that the arguments passed into the `atomic_*` functions should be of an atomic type. When making the code warning-free for icc, it fails to compile on any other compiler we tested.

In general, we saw that the compilers are quite conservative in terms of transformations when it comes to code dealing with fences and atomics. This leaves less room for errors. In the future, as more sophisticated compilation schemes for multi-threaded code are developed, compilers may take more aggressive transformations, at which point the simulator and the detailed C standard will become even more important.

[2]Of course, the programmer explicitly requested a fence in this code, so it is arguable that there may be an external reason the programmer had in mind when introducing the fence.

Chapter 7

Conclusion

Enabling compilers to optimize code in a multi-threaded environment is difficult. In a sequential environment, the compiler can use the *as-if* relation to shuffle instructions around for a more efficient execution, while maintaining the exact behaviour the original program had. But in a multi-threaded environment, this is complicated because threads can, at any point, view the effects of the behaviour of other threads. As a consequence, the C standard introduced the specification of *memory models*, which restrict how and in what order these effects must become visible. In a multi-threaded environment, correct inter-thread communication can only be done through the use of atomics, fences, mutexes, and other synchronization mechanisms, ensuring that the remaining sequential code keeps the *as-if* property.

How compilers can transform code relating to atomics and fences is defined, implicitly, by memory orders. The memory orders are complex models that compilers need to work with in order to efficiently implement multi-threading primitives. Implementing and understanding these complex models is not always straightforward, and therefore there is a real opportunity for errors. A test-suite for memory orders on these compilers helps with their standard-compliance by by signaling discrepancies between the correct behaviour (prescribed by the test-suite) and the compiler behaviour (observed when running). This standard compliance is required in industries that deal with safety-critical code, such as the automotive industry [7].

7.1 Findings and Contributions

In this context, the goal of our research was to construct a method to deterministically test compilers for standard-compliance when generating code for atomics and fences. Our method stems from the idea of providing a test-suite that (a) illustrates the relevant aspects for the standard, and (b) comprehensively tests whether the compiler under test generates any incorrect outputs for any interleaving.

Based on the methods and results presented in this thesis, we can provide the following answers to our main research questions.

RQ1: Can we define a systematic method to test C compilers for standard-compliance with respect to atomics and fences? We defined our method in Chapter 4. Specifically, we presented a three-step method for developing a test-suite. First, we presented our systematic approach of deriving test-cases from literature, which provided a sufficient set of tests to explore the capabilities of the simulator and explore a number of corner cases of the compilers. Second, we designed a simulator that finds and executes all possible execution interleavings of the code generated by a compiler. We implemented the simulator using the GNU Debugger. Finally, we discussed our approach on comparing expected output from the test-case with actual output given by the simulator.

RQ2: How do we assess the completeness of this method? To assess the completeness of our method, we designed and developed a four-step approach to evaluate the completeness of the simulator (see Chapter 6). This approach compares the bug-finding capability of the simulator with a naive approach. The simulator interleaves the threads and exhaustively executes all interleavings, while the naive approach depends on the randomness of the system scheduler to exhibit interesting behaviour. To intentionally add faults into the compiled code, we inserted faults into the test-suite by removing all hints to atomicity, allowing the compiler to be more free in transforming the code. Through this evaluation, we found that the simulator could not find faults that rely on a same-time execution of two

threads. However, vice-versa, we found several cases where our simulator found an issue where the naive approach could not, indicating the importance of an exhaustive approach.

In summary, the contributions of this work are:

- We proposed a novel, exhaustive approach for testing atomics in C programs.
- We demonstrated the use of GDB for interleaving threads.
- We defined three approaches for deriving test-cases (from the specification, from the compiler source code, and from the literature), and demonstrated the derivation of test-cases from literature is feasible in practice.
- We found that the four compilers we tested are too cautious when it comes to transformations on atomics.
- We found that the compilers under test are standard-compliant with respect to atomics and fences.
- We demonstrated a method to assess the completeness of our approach (test-suite and GDB-based interleaving) in a real multi-threaded environment.

7.2 Limitations and threats to validity

In this section we discuss the potential limitations and the threads to validity of our methodology. We separate them into three categories: applicability, technical, and fundamental limitations.

7.2.1 Limitations in general applicability

Our approach is limited in a general application in three ways: we tested the approach on only one machine, we used only four compilers, and the focus of our research was only on atomics.

The only architecture we used was the x86 architecture on an Intel CPU. The x86 architecture has a strong memory model, which contains only one relaxation of sequential consistency (store buffering). As such, there is little opportunity for a compiler to use significantly different instructions for the different types of memory orders. Architectures like the Arm and PowerPC are relaxed. These architectures therefore may provide more interesting venues for the compilers, and may lead to interesting results. Our results for the compilers targeting the x86 architecture are therefore also not generalizable to other architectures.

We only used the icc, GCC, musl-GCC and Clang compilers to test our approach on. We already found a number of differences between the output of icc, GCC and Clang. Different compilers may generate completely different code, and therefore our results are not generalizable to other compilers or configurations.

We focused the research primarily on atomics and we adopted fences into our test-suites. Mutexes are also part of the multi-threading primitives, and we found our implementation to not be immediately applicable to them. We believe our general approach of interleaving will be helpful to determine correct behaviour for mutexes. However, from preliminary testing we found that our implementation suffered from an excessive number of function call instructions that were generated for the mutexes.

7.2.2 Technical limitations

Our approach relies on the enumeration of all possible interleavings. When test-cases exceed 100000 interleavings, we exclude them from the test-suite. In general, we were able to reduce or adapt most test-cases that we encountered to fall in this acceptable range. However, we encountered a number of code samples that exceed the 100000 interleavings. This was also present in some executions on icc, where it generated more instructions than other compilers for a code sample, making it infeasible to run. Our approach is therefore limited in the test-cases it can accept.

7.2.3 Fundamental limitations

Through our evaluation, we found a fundamental limitation with our methodology: it is unable to detect faults that depend on behaviour of instructions of multiple threads executing at the same time. Our approach by interleaving threads on the level of instructions therefore seems limiting.

7.3 Future work

This work can be extended in general applicability. Future work can also attempt to solve some of the limitations we discovered with this method. Our simulator can be applied in a quantitative study on compilers and various architectures. This requires adapting our set of decision and jump instructions (e.g., `je` and `jmp`) from the x86 architecture to the other architectures.

Additionally, the simulator can be extended to test mutex implementations. We found that compilers introduce a large number of instructions to set up the call for the mutex functions. Due to the combinatorial explosion, an immediate use of our simulator was prohibited. However, these instructions may prove to be unnecessary to interleave, as they are not directly related to data that is being used in the region. Therefore it might be possible to filter out those instructions, resulting in a smaller interleaving count.

Similarly, more test cases can be considered by reducing the number of interleavings. This can be done by developing equivalence classes for the instructions. Identifying which instructions can have an impact on the value of the to-be-tested atomic, and only stepping those interleaving-wise may significantly cut the number of interleavings. We can also increase the number of allowed interleavings by more efficiently executing each interleaving. Currently, we do this in a sequential manner. However, our design is embarrassingly parallel. Each execution is completely independent from the others, and can therefore be run in a different GDB instance or on a different machine (provided the machines have the same architecture and software).

The fundamental limitation of our method may be solved in three ways. First, one can reduce thread-swapping time and buffer pollution, and therefore potentially solving the problem. This may be possible by interfacing directly with libthread_db, the thread debugging library that GDB uses. This may give the programmer more control on running and stepping through threads. An alternative approach could be to investigate the values of the store buffers, by means of a hardware simulator. It may be possible to detect whether an atomic is assigned a value in the store buffer, but that the value has not been propagated to other accesses happening on the same variable on other threads. Finally, we found that the naive testing approach *was* capable of detecting this issue. Therefore, it may be possible to design a more sophisticated approach that makes use of our naive testing approach to detect these type of issues with a high degree of confidence.

Bibliography

[1] C.-S. D. Yang and L. L. Pollock, *Program-based, structural testing of shared memory parallel programs*. Citeseer, 1999.

[2] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved multithreaded unit testing", in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 223–233.

[3] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs", in *Acm Sigplan Notices*, ACM, vol. 47, 2012, pp. 485–502.

[4] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software", *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005.

[5] W. Pugh, "Fixing the java memory model", in *Proceedings of the ACM 1999 conference on Java Grande*, 1999, pp. 89–98.

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing c++ concurrency", *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 55–66, 2011.

[7] R. Palin, D. Ward, I. Habli, and R. Rivett, "Iso 26262 safety cases: Compliance and assurance", 2011.

[8] ISO 9899:2018, "Information technology — Programming languages — C", International Organization for Standardization, Standard, Jun. 2018.

[9] H.-J. Boehm, "Threads cannot be implemented as a library", *ACM Sigplan Notices*, vol. 40, no. 6, pp. 261–268, 2005.

[10] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model", in *ACM SIGPLAN Notices*, ACM, vol. 43, 2008, pp. 68–78.

[11] *Cpp reference on std::memory_order*, `https://en.cppreference.com/w/cpp/atomic/memory_order`, visited on 2020/05/28.

[12] J. S. Myers, L. Crowl, L. Torvalds, M. Batty, M. Matz, O. Giroux, P. Sewell, P. Zijlstra, R. Radhakrishnan, R. Biener, *et al.*, "N4036: Towards implementation and use of memory order consume", 2014.

[13] D. Chen, Y. Jiang, C. Xu, X. Ma, and J. Lu, "Testing multithreaded programs via thread speed control", in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2018, pp. 15–25.

[14] O. Edelstein, E. Farcahi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur, "Framework for testing multithreaded java programs", *Concurrency and Computation: Practice and Experience*, vol. 15, no. 3-5, pp. 485–499, 2003.

[15] W. Pugh and N. Ayewah, "Unit testing concurrent software", in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 513–516.

[16] Y. Lei and R. H. Carver, "Reachability testing of concurrent programs", *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 382–403, 2006.

[17] K. Sen, "Effective random testing of concurrent programs", in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, 2007, pp. 323–332.

[18] J. Burnim, T. Elmas, G. Necula, and K. Sen, "Concurrit: Testing concurrent programs with programmable state-space exploration", in *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, 2012.

[19] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running tests against hardware", in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2011, pp. 41–44.

[20] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 36, no. 2, pp. 1–74, 2014.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation", *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[22] T. W. Doeppner, *Operating systems in depth*. Wiley, 2011.

[23] *Debugging with gdb*, `https://sourceware.org/gdb/current/onlinedocs/gdb/`, visited on 2020/07/08.

[24] N. Sidwell, V. Prus, P. Alves, S. Loosemore, and J. Blandy, "Non-stop multi-threaded debugging in gdb", in *GCC Developers' Summit*, Citeseer, vol. 117, 2008.

[25] *Setting watchpoints*, `https://sourceware.org/gdb/current/onlinedocs/gdb/Set-Watchpoints.html`. (visited on 06/09/2020).

[26] *Intel 64 and ia-32 architectures software developer's manual volume 2: Instruction set reference*, `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf`, visited on 2020/07/10.

[27] S. V. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware", *Communications of the ACM*, vol. 53, no. 8, pp. 90–101, 2010.

[28] V. Ziegler. (2014). C++ memory model, [Online]. Available: `https://www.think-cell.com/en/career/talks/pdf/think-cell_talk_memorymodel.pdf` (visited on 05/28/2020).

[29] W. Pugh, *Causality test cases*, `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`, visited on 2020/05/28.

[30] V. Vafeiadis and F. Z. Nardelli, "Verifying fence elimination optimisations", in *International Static Analysis Symposium*, Springer, 2011, pp. 146–162.

Appendix A

Full Results

A.1 0_atomic

A.1.1 With atomics

icc

```
THREAD 1 (4201788)
0x401d3c: mov    $0x4ea860,%eax
0x401d41: mov    $0x1,%edx
0x401d46: lock add %edx,(%rax)
0x401d49: mov    $0x4ea860,%eax
0x401d4e: mov    $0x3,%edx
0x401d53: lock add %edx,(%rax)
0x401d56: mov    $0x4ea860,%eax
0x401d5b: mov    $0x5,%edx
0x401d60: lock add %edx,(%rax)
THREAD 2 (4201868)
0x401d8c: mov    $0x4ea860,%eax
0x401d91: mov    $0x2,%edx
0x401d96: lock add %edx,(%rax)
0x401d99: mov    $0x4ea860,%eax
0x401d9e: mov    $0x4,%edx
0x401da3: lock add %edx,(%rax)
0x401da6: mov    $0x4ea860,%eax
0x401dab: mov    $0x6,%edx
0x401db0: lock add %edx,(%rax)
Maximum orders: 48620.0
Completed running 48620 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/0_atomic >> out/accicc_0_atomic; done"  35,06s
    user 18,13s system 88% cpu 59,841 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC & Clang

```
THREAD 1 (4202044)
0x401e3c: lock addl $0x1,0xe2efc(%rip)        # 0x4e4d40 <x>
0x401e44: lock addl $0x3,0xe2ef4(%rip)        # 0x4e4d40 <x>
0x401e4c: lock addl $0x5,0xe2eec(%rip)        # 0x4e4d40 <x>
THREAD 2 (4202124)
0x401e8c: lock addl $0x2,0xe2eac(%rip)        # 0x4e4d40 <x>
0x401e94: lock addl $0x4,0xe2ea4(%rip)        # 0x4e4d40 <x>
0x401e9c: lock addl $0x6,0xe2e9c(%rip)        # 0x4e4d40 <x>
Maximum orders: 20.0
Completed running 20 orders.
Number failed: 0.

Default run:
```

```
bash -c "for run in {1..50000}; do bin/0_atomic >> out/acc_0_atomic.txt; done  51,11s
    user 46,59s system 83% cpu 1:57,31 total
Failed: 0 / 50000
Incorrect result states: None
```

A.1.2 Without atomics

icc & GCC & musl-GCC & Clang

```
THREAD 1 (4201788)
0x401d3c: addl   $0x9,0xe8b1d(%rip)        # 0x4ea860 <x>
THREAD 2 (4201836)
0x401d6c: addl   $0xc,0xe8aed(%rip)        # 0x4ea860 <x>
Maximum orders: 2.0
Completed running 2 orders.
Number failed: 0.

Thread 1: Precompute addition, removal of lock.
Thread 2: Precompute addition, removal of lock.

Default run:
bash -c "for run in {1..50000}; do bin/0_atomic >> out/fin_0_atomic.txt; done  51,51s
    user 48,01s system 83% cpu 1:58,98 total
Failed: 150 / 50000
Incorrect result states: 9, or 12
```

A.2 1_rwc

A.2.1 With atomics

icc

```
THREAD 1 (4201948)
0x401ddc: mov    $0x4ea860,%edx
0x401de1: mov    $0x1,%eax
0x401de6: mov    %eax,(%rdx)
THREAD 2 (4201996)
0x401e0c: mov    $0x4ea86c,%edx
0x401e11: mov    $0x1,%eax
0x401e16: mov    %eax,(%rdx)
THREAD 3 (4202044)
0x401e3c: mov    $0x4ea860,%eax
0x401e41: mov    (%rax),%eax
0x401e43: mov    %eax,0xe8a2b(%rip)        # 0x4ea874 <r1>
0x401e49: mfence
0x401e4c: mov    $0x4ea86c,%eax
0x401e51: mov    (%rax),%eax
0x401e53: mov    %eax,0xe8a17(%rip)        # 0x4ea870 <r2>
THREAD 4 (4202108)
0x401e7c: mov    $0x4ea86c,%eax
0x401e81: mov    (%rax),%eax
0x401e83: mov    %eax,0xe89db(%rip)        # 0x4ea864 <r3>
0x401e89: mfence
0x401e8c: mov    $0x4ea860,%eax
0x401e91: mov    (%rax),%eax
0x401e93: mov    %eax,0xe89cf(%rip)        # 0x4ea868 <r4>
Maximum orders: 2660486400.0

(Infeasible to run)

Default run:
bash -c "for run in {1..50000}; do bin/1_rwc >> out/accicc_1_rwc; done"  41,10s user
    24,58s system 86% cpu 1:15,98 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC & Clang

```
THREAD 1 (4202204)
0x401edc: movl   $0x1,0xe2e5a(%rip)        # 0x4e4d40 <x>
THREAD 2 (4202268)
0x401f1c: movl   $0x1,0xe2e26(%rip)        # 0x4e4d4c <y>
THREAD 3 (4202332)
0x401f5c: mov    0xe2dde(%rip),%eax        # 0x4e4d40 <x>
0x401f62: mov    %eax,0xe2dec(%rip)        # 0x4e4d54 <r1>
0x401f68: mfence
0x401f6b: mov    0xe2ddb(%rip),%eax        # 0x4e4d4c <y>
0x401f71: mov    %eax,0xe2dd9(%rip)        # 0x4e4d50 <r2>
THREAD 4 (4202412)
0x401fac: mov    0xe2d9a(%rip),%eax        # 0x4e4d4c <y>
0x401fb2: mov    %eax,0xe2d8c(%rip)        # 0x4e4d44 <r3>
0x401fb8: mfence
0x401fbb: mov    0xe2d7f(%rip),%eax        # 0x4e4d40 <x>
0x401fc1: mov    %eax,0xe2d81(%rip)        # 0x4e4d48 <r4>
Maximum orders: 33264.0
Completed running 33264 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/1_rwc >> out/acc_1_rwc.txt; done"  57,90s user
    45,39s system 48% cpu 3:34,66 total
Failed: 0 / 50000
Incorrect result states: None
```

A.2.2 Without atomics

icc

```
THREAD 1 (4201948)
0x401ddc: movl   $0x1,0xe8a7a(%rip)        # 0x4ea860 <x>
THREAD 2 (4201996)
0x401e0c: movl   $0x1,0xe8a56(%rip)        # 0x4ea86c <y>
THREAD 3 (4202044)
0x401e3c: mov    0xe8a1e(%rip),%eax        # 0x4ea860 <x>
0x401e42: mov    0xe8a24(%rip),%edx        # 0x4ea86c <y>
0x401e48: mov    %eax,0xe8a26(%rip)        # 0x4ea874 <r1>
0x401e4e: mov    %edx,0xe8a1c(%rip)        # 0x4ea870 <r2>
THREAD 4 (4202108)
0x401e7c: mov    0xe89ea(%rip),%eax        # 0x4ea86c <y>
0x401e82: mov    0xe89d8(%rip),%edx        # 0x4ea860 <x>
0x401e88: mov    %eax,0xe89d6(%rip)        # 0x4ea864 <r3>
0x401e8e: mov    %edx,0xe89d4(%rip)        # 0x4ea868 <r4>
Maximum orders: 6300.0
Completed running 6300 orders.
Number failed: 0.

Thread 2: Moved <y> before <r1>
Thread 3: Moved <x> before <r3>

Default run:
bash -c "for run in {1..50000}; do bin/1_rwc >> out/finicc_1_rwc.txt; done"  56,74s user
     47,32s system 61% cpu 2:48,42 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC & Clang

```
THREAD 1 (4202204)
0x401edc: movl   $0x1,0xe2e5a(%rip)        # 0x4e4d40 <x>
THREAD 2 (4202268)
0x401f1c: movl   $0x1,0xe2e26(%rip)        # 0x4e4d4c <y>
THREAD 3 (4202332)
0x401f5c: mov    0xe2dde(%rip),%eax        # 0x4e4d40 <x>
0x401f62: mov    %eax,0xe2dec(%rip)        # 0x4e4d54 <r1>
0x401f68: mov    0xe2dde(%rip),%eax        # 0x4e4d4c <y>
```

```
0x401f6e: mov    %eax,0xe2ddc(%rip)        # 0x4e4d50 <r2>
THREAD 4 (4202412)
0x401fac: mov    0xe2d9a(%rip),%eax        # 0x4e4d4c <y>
0x401fb2: mov    %eax,0xe2d8c(%rip)        # 0x4e4d44 <r3>
0x401fb8: mov    0xe2d82(%rip),%eax        # 0x4e4d40 <x>
0x401fbe: mov    %eax,0xe2d84(%rip)        # 0x4e4d48 <r4>
Maximum orders: 6300.0
Completed running 6300 orders.
Number failed: 0.

<Nothing changed> (mfence is gone because atomic hints are gone)

Default run:
bash -c "for run in {1..50000}; do bin/1_rwc >> out/fingcc_1_rwc.txt; done"  59,63s user
     50,58s system 49% cpu 3:43,05 total
Failed: 0 / 50000
Incorrect result states: None
```

A.3 2_wrc

A.3.1 With atomics

icc

```
THREAD 1 (4201884)
0x401d9c: mov    $0x4ea860,%edx
0x401da1: mov    $0x1,%eax
0x401da6: mov    %eax,(%rdx)
THREAD 2 (4201932)
0x401dcc: mov    $0x4ea860,%eax
0x401dd1: mov    (%rax),%eax
0x401dd3: mov    %eax,0xe8a97(%rip)        # 0x4ea870 <r1>
0x401dd9: mfence
0x401ddc: mov    $0x4ea868,%edx
0x401de1: mov    $0x1,%eax
0x401de6: mov    %eax,(%rdx)
THREAD 3 (4201996)
0x401e0c: mov    $0x4ea868,%eax
0x401e11: mov    (%rax),%eax
0x401e13: mov    %eax,0xe8a53(%rip)        # 0x4ea86c <r2>
0x401e19: mfence
0x401e1c: mov    $0x4ea860,%eax
0x401e21: mov    (%rax),%eax
0x401e23: mov    %eax,0xe8a3b(%rip)        # 0x4ea864 <r3>
Maximum orders: 2333760.0

(Infeasible to run)

Default run:
bash -c "for run in {1..50000}; do bin/2_wrc >> out/accicc_2_wrc.txt; done"  61,74s user
     54,57s system 39% cpu 4:52,26 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC & Clang

```
THREAD 1 (4202140)
0x401e9c: movl   $0x1,0xe2e9a(%rip)        # 0x4e4d40 <x>
THREAD 2 (4202204)
0x401edc: mov    0xe2e5e(%rip),%eax        # 0x4e4d40 <x>
0x401ee2: mov    %eax,0xe2e68(%rip)        # 0x4e4d50 <r1>
0x401ee8: mfence
0x401eeb: movl   $0x1,0xe2e53(%rip)        # 0x4e4d48 <y>
THREAD 3 (4202284)
0x401f2c: mov    0xe2e16(%rip),%eax        # 0x4e4d48 <y>
0x401f32: mov    %eax,0xe2e14(%rip)        # 0x4e4d4c <r2>
0x401f38: mfence
0x401f3b: mov    0xe2dff(%rip),%eax        # 0x4e4d40 <x>
```

```
0x401f41: mov    %eax,0xe2dfd(%rip)        # 0x4e4d44 <r3>
Maximum orders: 1260.0
Completed running 1260 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/2_wrc >> out/accgcc_2_wrc.txt; done"  59,13s user
     53,71s system 36% cpu 5:06,39 total
Failed: 0 / 50000
Incorrect result states: None
```

A.3.2 Without atomics

icc

```
THREAD 1 (4201884)
0x401d9c: movl   $0x1,0xe8aba(%rip)        # 0x4ea860 <x>
THREAD 2 (4201932)
0x401dcc: mov    0xe8a8e(%rip),%eax        # 0x4ea860 <x>
0x401dd2: mov    %eax,0xe8a98(%rip)        # 0x4ea870 <r1>
0x401dd8: movl   $0x1,0xe8a86(%rip)        # 0x4ea868 <y>
THREAD 3 (4201996)
0x401e0c: mov    0xe8a56(%rip),%eax        # 0x4ea868 <y>
0x401e12: mov    0xe8a48(%rip),%edx        # 0x4ea860 <x>
0x401e18: mov    %eax,0xe8a4e(%rip)        # 0x4ea86c <r2>
0x401e1e: mov    %edx,0xe8a40(%rip)        # 0x4ea864 <r3>
Maximum orders: 280.0
Completed running 280 orders.
Number failed: 0.

Thread 3: Moved <x> before <r2>

Default run:
bash -c "for run in {1..50000}; do bin/2_wrc >> out/finicc_2_wrc.txt; done"  57,05s user
     51,67s system 62% cpu 2:53,94 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC

```
THREAD 1 (4202140)
0x401e9c: movl   $0x1,0xe2e9a(%rip)        # 0x4e4d40 <x>
THREAD 2 (4202204)
0x401edc: movl   $0x1,0xe2e62(%rip)        # 0x4e4d48 <y>
0x401ee6: mov    0xe2e54(%rip),%eax        # 0x4e4d40 <x>
0x401eec: mov    %eax,0xe2e5e(%rip)        # 0x4e4d50 <r1>
THREAD 3 (4202284)
0x401f2c: mov    0xe2e16(%rip),%eax        # 0x4e4d48 <y>
0x401f32: mov    %eax,0xe2e14(%rip)        # 0x4e4d4c <r2>
0x401f38: mov    0xe2e02(%rip),%eax        # 0x4e4d40 <x>
0x401f3e: mov    %eax,0xe2e00(%rip)        # 0x4e4d44 <r3>
Maximum orders: 280.0
Completed running 280 orders.
Number failed: 4.
One or more test execution orders failed:
[[2, 3, 3, 3, 1, 2, 2, 3], [2, 3, 3, 3, 3, 1, 2, 2], [2, 3, 3, 3, 1, 2, 3, 2], [2, 3, 3,
     3, 1, 3, 2, 2]]

Thread 2: Moved <y> before <x> and <r1>

Default run:
bash -c "for run in {1..50000}; do bin/2_wrc >> out/fingcc_2_wrc.txt; done"  55,06s user
     49,84s system 74% cpu 2:20,76 total
Failed: 0 / 50000
Incorrect result states: None
```

Clang

```
THREAD 1 (4201629)
0x401c9d: movl   $0x1,0xe5099(%rip)        # 0x4e6d40 <x>
THREAD 2 (4201677)
0x401ccd: mov    0xe506d(%rip),%eax        # 0x4e6d40 <x>
0x401cd3: mov    %eax,0xe5077(%rip)        # 0x4e6d50 <r1>
0x401cd9: movl   $0x1,0xe5065(%rip)        # 0x4e6d48 <y>
THREAD 3 (4201741)
0x401d0d: mov    0xe5035(%rip),%eax        # 0x4e6d48 <y>
0x401d13: mov    %eax,0xe5033(%rip)        # 0x4e6d4c <r2>
0x401d19: mov    0xe5021(%rip),%eax        # 0x4e6d40 <x>
0x401d1f: mov    %eax,0xe501f(%rip)        # 0x4e6d44 <r3>
Maximum orders: 280.0
Completed running 280 orders.
Number failed: 0.
All test orders succeeded!

<Nothing changed>

Default run:
bash -c "for run in {1..50000}; do bin/2_wrc >> out/finclang_2_wrc.txt; done"  56,55s
    user 51,00s system 71% cpu 2:31,22 total
Failed: 0 / 50000
Incorrect result states: None
```

A.4 3_rwc

A.4.1 With atomics

icc

```
THREAD 1 (4201884)
0x401d9c: mov    $0x4ea860,%edx
0x401da1: mov    $0x1,%eax
0x401da6: mov    %eax,(%rdx)
THREAD 2 (4201932)
0x401dcc: mov    $0x4ea860,%eax
0x401dd1: mov    (%rax),%eax
0x401dd3: mov    %eax,0xe8a97(%rip)        # 0x4ea870 <r1>
0x401dd9: mfence
0x401ddc: mov    $0x4ea868,%eax
0x401de1: mov    (%rax),%eax
0x401de3: mov    %eax,0xe8a83(%rip)        # 0x4ea86c <r2>
THREAD 3 (4201996)
0x401e0c: mov    $0x4ea868,%edx
0x401e11: mov    $0x1,%eax
0x401e16: mov    %eax,(%rdx)
0x401e18: mfence
0x401e1b: mov    $0x4ea860,%eax
0x401e20: mov    (%rax),%eax
0x401e22: mov    %eax,0xe8a3c(%rip)        # 0x4ea864 <r3>
Maximum orders: 2333760.0

(Infeasible to run)

Default run:
bash -c "for run in {1..50000}; do bin/3_rwc >> out/accicc_3_rwc.txt; done"  52,40s user
     40,09s system 74% cpu 2:04,32 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC & Clang

```
THREAD 1 (4202140)
0x401e9c: movl   $0x1,0xe2e9a(%rip)        # 0x4e4d40 <x>
THREAD 2 (4202204)
0x401edc: mov    0xe2e5e(%rip),%eax        # 0x4e4d40 <x>
```

```
0x401ee2: mov    %eax,0xe2e68(%rip)        # 0x4e4d50 <r1>
0x401ee8: mfence
0x401eeb: mov    0xe2e57(%rip),%eax        # 0x4e4d48 <y>
0x401ef1: mov    %eax,0xe2e55(%rip)        # 0x4e4d4c <r2>
THREAD 3 (4202284)
0x401f2c: movl   $0x1,0xe2e12(%rip)        # 0x4e4d48 <y>
0x401f36: mfence
0x401f39: mov    0xe2e01(%rip),%eax        # 0x4e4d40 <x>
0x401f3f: mov    %eax,0xe2dff(%rip)        # 0x4e4d44 <r3>
Maximum orders: 1260.0
Completed running 1260 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/3_rwc >> out/accgcc_3_rwc.txt; done"  56,21s user
     48,15s system 75% cpu 2:18,27 total
Failed: 0 / 50000
Incorrect result states: None
```

A.4.2 Without atomics

icc

```
THREAD 1 (4201884)
0x401d9c: movl   $0x1,0xe8aba(%rip)        # 0x4ea860 <x>
THREAD 2 (4201932)
0x401dcc: mov    0xe8a8e(%rip),%eax        # 0x4ea860 <x>
0x401dd2: mov    0xe8a90(%rip),%edx        # 0x4ea868 <y>
0x401dd8: mov    %eax,0xe8a92(%rip)        # 0x4ea870 <r1>
0x401dde: mov    %edx,0xe8a88(%rip)        # 0x4ea86c <r2>
THREAD 3 (4201996)
0x401e0c: mov    0xe8a4e(%rip),%eax        # 0x4ea860 <x>
0x401e12: movl   $0x1,0xe8a4c(%rip)        # 0x4ea868 <y>
0x401e1c: mov    %eax,0xe8a42(%rip)        # 0x4ea864 <r3>
Maximum orders: 280.0
Completed running 280 orders.
Number failed: 6.
One or more test execution orders failed:
[[3, 1, 2, 2, 2, 2, 3, 3], [3, 1, 2, 2, 3, 2, 2, 3], [3, 1, 2, 2, 2, 3, 3, 2], [3, 1, 2,
     2, 2, 3, 2, 3], [3, 1, 2, 2, 3, 3, 2, 2], [3, 1, 2, 2, 3, 2, 3, 2]]

Thread 2: Moved <y> before <r1>
Thread 3: Moved <x> before <y>

Default run:
bash -c "for run in {1..50000}; do bin/3_rwc >> out/finicc_3_rwc.txt; done"  56,31s user
     47,85s system 74% cpu 2:18,99 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC & Clang

```
THREAD 1 (4202140)
0x401e9c: movl   $0x1,0xe2e9a(%rip)        # 0x4e4d40 <x>
THREAD 2 (4202204)
0x401edc: mov    0xe2e5e(%rip),%eax        # 0x4e4d40 <x>
0x401ee2: mov    %eax,0xe2e68(%rip)        # 0x4e4d50 <r1>
0x401ee8: mov    0xe2e5a(%rip),%eax        # 0x4e4d48 <y>
0x401eee: mov    %eax,0xe2e58(%rip)        # 0x4e4d4c <r2>
THREAD 3 (4202284)
0x401f2c: movl   $0x1,0xe2e12(%rip)        # 0x4e4d48 <y>
0x401f36: mov    0xe2e04(%rip),%eax        # 0x4e4d40 <x>
0x401f3c: mov    %eax,0xe2e02(%rip)        # 0x4e4d44 <r3>
Maximum orders: 280.0
Completed running 280 orders.
Number failed: 0.

<Nothing changed (removed mfence due to removed hint)>
```

```
Default run:
bash -c "for run in {1..50000}; do bin/3_rwc >> out/fingcc_3_rwc.txt; done"  53,29s user
     41,02s system 68% cpu 2:18,14 total
Failed: 0 / 50000
Incorrect result states: None
```

A.5 4_thinair

A.5.1 With atomics

icc

```
THREAD 1 (4201756)
0x401d1c: mov    $0x4ea860,%eax
0x401d21: mov    (%rax),%eax
0x401d23: mov    $0x4ea864,%edx
0x401d28: mov    %eax,0xe8b3e(%rip)        # 0x4ea86c <r1>
0x401d2e: mov    %eax,(%rdx)
THREAD 2 (4201804)
0x401d4c: mov    $0x4ea864,%eax
0x401d51: mov    (%rax),%eax
0x401d53: mov    $0x4ea860,%edx
0x401d58: mov    %eax,0xe8b0a(%rip)        # 0x4ea868 <r2>
0x401d5e: mov    %eax,(%rdx)
Maximum orders: 252.0
Completed running 252 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/4_thinair >> out/accicc_4_thinair; done"  37,76s
    user 22,09s system 82% cpu 1:12,80 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC & Clang

```
THREAD 1 (4201996)
0x401e0c: mov    0xe2f2e(%rip),%eax        # 0x4e4d40 <x>
0x401e12: mov    %eax,0xe2f34(%rip)        # 0x4e4d4c <r1>
0x401e18: mov    %eax,0xe2f26(%rip)        # 0x4e4d44 <y>
THREAD 2 (4202076)
0x401e5c: mov    0xe2ee2(%rip),%eax        # 0x4e4d44 <y>
0x401e62: mov    %eax,0xe2ee0(%rip)        # 0x4e4d48 <r2>
0x401e68: mov    %eax,0xe2ed2(%rip)        # 0x4e4d40 <x>
Maximum orders: 20.0
Completed running 20 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/4_thinair >> out/accgcc_4_thinair; done"  36,55s
    user 20,45s system 86% cpu 1:05,90 total
Failed: 0 / 50000
Incorrect result states: None
```

A.5.2 Without atomics

icc & GCC & musl-GCC & Clang

```
THREAD 1 (4201996)
0x401e0c: mov    0xe2f2e(%rip),%eax        # 0x4e4d40 <x>
0x401e12: mov    %eax,0xe2f34(%rip)        # 0x4e4d4c <r1>
0x401e18: mov    %eax,0xe2f26(%rip)        # 0x4e4d44 <y>
THREAD 2 (4202076)
0x401e5c: mov    0xe2ee2(%rip),%eax        # 0x4e4d44 <y>
0x401e62: mov    %eax,0xe2ee0(%rip)        # 0x4e4d48 <r2>
0x401e68: mov    %eax,0xe2ed2(%rip)        # 0x4e4d40 <x>
```

```
Maximum orders: 20.0
Completed running 20 orders.
Number failed: 0.

<Nothing changed (except for icc)>
<Systems are unlikely to implement this wrongly>

Default run:
bash -c "for run in {1..50000}; do bin/4_thinair >> out/fingcc_4_thinair; done"  36,55s
    user 20,45s system 86% cpu 1:05,90 total
Failed: 0 / 50000
Incorrect result states: None
```

A.6 5_thinair

A.6.1 With atomics

icc

```
THREAD 1 (4201788)
0x401d3c: mov    $0x4ea860,%eax
0x401d41: mov    (%rax),%eax
0x401d43: mov    %eax,0xe8b23(%rip)        # 0x4ea86c <r1>
THREAD 2 (4201836)
0x401d6c: mov    $0x4ea864,%eax
0x401d71: mov    (%rax),%eax
0x401d73: mov    $0x4ea860,%edx
0x401d78: mov    %eax,0xe8aea(%rip)        # 0x4ea868 <r2>
0x401d7e: mov    %eax,(%rdx)
THREAD 3 (4201884)
0x401d9c: mov    $0x4ea864,%edx
0x401da1: mov    $0x1,%eax
0x401da6: mov    %eax,(%rdx)
Maximum orders: 9240.0
Completed running 9240 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/5_thinair >> out/accicc_5_thinair; done"
  40,27s user 24,72s system 85% cpu 1:16,41 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC & Clang

```
THREAD 1 (4202028)
0x401e2c: mov    0xe2f0e(%rip),%eax        # 0x4e4d40 <x>
0x401e32: mov    %eax,0xe2f14(%rip)        # 0x4e4d4c <r1>
THREAD 2 (4202092)
0x401e6c: mov    0xe2ed2(%rip),%eax        # 0x4e4d44 <y>
0x401e72: mov    %eax,0xe2ed0(%rip)        # 0x4e4d48 <r2>
0x401e78: mov    %eax,0xe2ec2(%rip)        # 0x4e4d40 <x>
THREAD 3 (4202172)
0x401ebc: movl   $0x1,0xe2e7e(%rip)        # 0x4e4d44 <y>
Maximum orders: 60.0
Completed running 60 orders.
Number failed: 0.

bash -c "for run in {1..50000}; do bin/5_thinair >> out/accgcc_5_thinair; done"   41,34s
     user 26,82s system 77% cpu 1:28,47 total
Failed: 0 / 50000
Incorrect result states: None
```

A.6.2 Without atomics

icc & GCC & musl-GCC & Clang

```
THREAD 1 (4201788)
0x401d3c: mov    0xe8b1e(%rip),%eax        # 0x4ea860 <x>
0x401d42: mov    %eax,0xe8b24(%rip)        # 0x4ea86c <r1>
THREAD 2 (4201836)
0x401d6c: mov    0xe8af2(%rip),%eax        # 0x4ea864 <y>
0x401d72: mov    %eax,0xe8af0(%rip)        # 0x4ea868 <r2>
0x401d78: mov    %eax,0xe8ae2(%rip)        # 0x4ea860 <x>
THREAD 3 (4201884)
0x401d9c: movl   $0x1,0xe8abe(%rip)        # 0x4ea864 <y>
Maximum orders: 60.0
Completed running 60 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/5_thinair >> out/fingcc_5_thinair; done"  39,12s
    user 23,92s system 93% cpu 1:07,26 total
Failed: 0 / 50000
Incorrect result states: None
```

A.7 6_thinair_array

A.7.1 With atomics

icc

```
THREAD 1 (4201772)
0x401d2c: mov    $0x4e447c,%eax
0x401d31: mov    (%rax),%eax
0x401d33: mov    %eax,0xe274b(%rip)        # 0x4e4484 <r1>
0x401d39: mov    $0x4e4480,%edx
0x401d3e: movslq %eax,%rax
0x401d41: movl   $0x0,0x4ea880(,%rax,4)
0x401d4c: mov    0xe8b2e(%rip),%eax        # 0x4ea880 <a>
0x401d52: mov    %eax,0xe2730(%rip)        # 0x4e4488 <r2>
0x401d58: mov    %eax,(%rdx)
THREAD 2 (4201852)
0x401d7c: mov    $0x4e4480,%eax
0x401d81: mov    (%rax),%eax
0x401d83: mov    $0x4e447c,%edx
0x401d88: mov    %eax,0xe26fe(%rip)        # 0x4e448c <r3>
0x401d8e: mov    %eax,(%rdx)
Maximum orders: 2002.0
Completed running 2002 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/6_thinair_array >> out/accicc_6_thinair_array;
    done"
  39,04s user 24,23s system 80% cpu 1:18,86 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC

```
THREAD 1 (4202012)
0x401e1c: movslq 0xdd5bd(%rip),%rax        # 0x4df3e0 <x>
0x401e23: lea    0xe2f36(%rip),%rdx        # 0x4e4d60 <a>
0x401e2a: mov    %eax,0xdd5a8(%rip)        # 0x4df3d8 <r1>
0x401e30: movl   $0x0,(%rdx,%rax,4)
0x401e37: mov    0xe2f23(%rip),%eax        # 0x4e4d60 <a>
0x401e3d: mov    %eax,0xdd591(%rip)        # 0x4df3d4 <r2>
0x401e43: mov    %eax,0xdd593(%rip)        # 0x4df3dc <y>
THREAD 2 (4202108)
0x401e7c: mov    0xdd55a(%rip),%eax        # 0x4df3dc <y>
0x401e82: mov    %eax,0xdd548(%rip)        # 0x4df3d0 <r3>
0x401e88: mov    %eax,0xdd552(%rip)        # 0x4df3e0 <x>
Maximum orders: 120.0
```

```
Completed running 120 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/6_thinair_array >> out/accgcc_6_thinair_array;
    done"  38,38s user 24,44s system 81% cpu 1:17,21 total
Failed: 0 / 50000
Incorrect result states: None
```

Clang

```
THREAD 1 (4201629)
0x401c9d: mov    0xdf719(%rip),%eax        # 0x4e13bc <x>
0x401ca3: mov    %eax,0xdf71b(%rip)        # 0x4e13c4 <r1>
0x401ca9: cltq
0x401cab: movl   $0x0,0x4e6d60(,%rax,4)
0x401cb6: mov    0xe50a4(%rip),%eax        # 0x4e6d60 <a>
0x401cbc: mov    %eax,0xdf706(%rip)        # 0x4e13c8 <r2>
0x401cc2: mov    %eax,0xdf6f8(%rip)        # 0x4e13c0 <y>
THREAD 2 (4201709)
0x401ced: mov    0xdf6cd(%rip),%eax        # 0x4e13c0 <y>
0x401cf3: mov    %eax,0xdf6d3(%rip)        # 0x4e13cc <r3>
0x401cf9: mov    %eax,0xdf6bd(%rip)        # 0x4e13bc <x>
Maximum orders: 120.0
Completed running 120 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/6_thinair_array >> out/accclang_6_thinair_array;
    done"  38,47s user 23,67s system 90% cpu 1:08,76 total
Failed: 0 / 50000
Incorrect result states: None
```

A.7.2 Without atomics

icc

```
THREAD 1 (4201772)
0x401d2c: movslq 0xe2749(%rip),%rax        # 0x4e447c <x>
0x401d33: mov    %eax,0xe274b(%rip)        # 0x4e4484 <r1>
0x401d39: movl   $0x0,0x4ea880(,%rax,4)
0x401d44: mov    0xe8b36(%rip),%edx        # 0x4ea880 <a>
0x401d4a: mov    %edx,0xe2738(%rip)        # 0x4e4488 <r2>
0x401d50: mov    %edx,0xe272a(%rip)        # 0x4e4480 <y>
THREAD 2 (4201852)
0x401d7c: mov    0xe26fe(%rip),%eax        # 0x4e4480 <y>
0x401d82: mov    %eax,0xe2704(%rip)        # 0x4e448c <r3>
0x401d88: mov    %eax,0xe26ee(%rip)        # 0x4e447c <x>
Maximum orders: 84.0
Completed running 84 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/6_thinair_array >> out/finicc_6_thinair_array;
    done"  38,23s user 23,95s system 90% cpu 1:08,99 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC

```
THREAD 1 (4202012)
0x401e1c: movslq 0xdd5bd(%rip),%rax        # 0x4df3e0 <x>
0x401e23: lea    0xe2f36(%rip),%rdx        # 0x4e4d60 <a>
0x401e2a: mov    %eax,0xdd5a8(%rip)        # 0x4df3d8 <r1>
0x401e30: movl   $0x0,(%rdx,%rax,4)
0x401e37: mov    0xe2f23(%rip),%eax        # 0x4e4d60 <a>
0x401e3d: mov    %eax,0xdd591(%rip)        # 0x4df3d4 <r2>
```

```
0x401e43: mov    %eax,0xdd593(%rip)        # 0x4df3dc <y>
THREAD 2 (4202108)
0x401e7c: mov    0xdd55a(%rip),%eax        # 0x4df3dc <y>
0x401e82: mov    %eax,0xdd548(%rip)        # 0x4df3d0 <r3>
0x401e88: mov    %eax,0xdd552(%rip)        # 0x4df3e0 <x>
Maximum orders: 120.0
Completed running 120 orders.
Number failed: 0.

<No changes>

Default run:
bash -c "for run in {1..50000}; do bin/6_thinair_array >> out/fingcc_6_thinair_array;
    done"
  39,92s user 26,08s system 77% cpu 1:24,96 total
Failed: 0 / 50000
Incorrect result states: None
```

Clang

```
THREAD 1 (4201629)
0x401c9d: movslq 0xdf718(%rip),%rax        # 0x4e13bc <x>
0x401ca4: mov    %eax,0xdf71a(%rip)        # 0x4e13c4 <r1>
0x401caa: movl   $0x0,0x4e6d60(,%rax,4)
0x401cb5: mov    0xe50a5(%rip),%eax        # 0x4e6d60 <a>
0x401cbb: mov    %eax,0xdf707(%rip)        # 0x4e13c8 <r2>
0x401cc1: mov    %eax,0xdf6f9(%rip)        # 0x4e13c0 <y>
THREAD 2 (4201709)
0x401ced: mov    0xdf6cd(%rip),%eax        # 0x4e13c0 <y>
0x401cf3: mov    %eax,0xdf6d3(%rip)        # 0x4e13cc <r3>
0x401cf9: mov    %eax,0xdf6bd(%rip)        # 0x4e13bc <x>
Maximum orders: 84.0
Completed running 84 orders.
Number failed: 0.

Thread 1: No cltq

Default run:
bash -c "for run in {1..50000}; do bin/6_thinair_array >> out/finclang_6_thinair_array;
    done"  37,24s user 21,93s system 83% cpu 1:11,23 total
Failed: 0 / 50000
Incorrect result states: None
```

A.8 7_tmo

A.8.1 With atomics

icc

```
THREAD 1 (4201756)
0x401d1c: mov    $0x4e4484,%edx
0x401d21: mov    $0x1,%eax
0x401d26: movl   $0x17,0xe274c(%rip)        # 0x4e447c <n>
0x401d30: xchg   %al,(%rdx)
THREAD 2 (4201820)
0x401d5c: mov    $0x4e4484,%eax
0x401d61: mov    (%rax),%al
0x401d63: test   %al,%al
0x401d65: jne    0x401d72 <threadFunc2+34>
0x401d72: mov    0xe2704(%rip),%eax        # 0x4e447c <n>
0x401d78: mov    %eax,0xe2702(%rip)        # 0x4e4480 <y>
0x401d67: LOOP 0x401d72
0x401d6c: mov    (%rax),%al
0x401d6e: test   %al,%al
0x401d70: je     0x401d67 <threadFunc2+23>
Maximum orders: 1001.0
Completed running 1001 orders.
Number failed: 0
```

```
Default run:
bash -c "for run in {1..50000}; do bin/7_tmo >> out/accicc_7_tmo; done"  45,43s user
    26,34s system 80% cpu 1:28,93 total
Failed: 0 / 50000
Incorrect result states: None
```

GCC & musl-GCC

```
THREAD 1 (4201996)
0x401e0c: movl   $0x17,0xdd5c2(%rip)        # 0x4df3d8 <n>
0x401e16: movb   $0x1,0xdd5b3(%rip)        # 0x4df3d0 <x>
0x401e1d: mfence
THREAD 2 (4202076)
0x401e5c: nopl   0x0(%rax)
0x401e60: LOOP 0x401e6b
0x401e67: test   %al,%al
0x401e69: je     0x401e60 <threadFunc2+32>
0x401e6b: mov    0xdd567(%rip),%eax        # 0x4df3d8 <n>
0x401e71: mov    %eax,0xdd55d(%rip)        # 0x4df3d4 <y>
Maximum orders: 84.0
Completed running 84 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/7_tmo >> out/accgcc_7_tmo; done"  42,17s user
    24,30s system 88% cpu 1:15,35 total
Failed: 0 / 50000
Incorrect result states: None
```

Clang

```
THREAD 1 (4201629)
0x401c9d: movl   $0x17,0xdf715(%rip)        # 0x4e13bc <n>
0x401ca7: mov    $0x1,%al
0x401ca9: xchg   %al,0xdf715(%rip)        # 0x4e13c4 <x>
THREAD 2 (4201677)
0x401ccd: nopl   (%rax)
0x401cd0: LOOP 0x401cdb
0x401cd7: test   $0x1,%al
0x401cd9: je     0x401cd0 <threadFunc2+16>
0x401cdb: mov    0xdf6db(%rip),%eax        # 0x4e13bc <n>
0x401ce1: mov    %eax,0xdf6d9(%rip)        # 0x4e13c0 <y>
Maximum orders: 84.0
Completed running 84 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/7_tmo >> out/accclang_7_tmo; done"  44,94s user
    25,66s system 84% cpu 1:23,64 total
Failed: 0 / 50000
Incorrect result states: None
```

A.8.2 Without atomics

Same as 8_tmo_weaker. See Section A.9.2.

A.9 8_tmo_weaker

A.9.1 With atomics

icc

```
THREAD 1 (4201756)
0x401d1c: mov    $0x4e4484,%edx
```

```
0x401d21: mov    $0x1,%eax
0x401d26: movl   $0x17,0xe274c(%rip)        # 0x4e447c <n>
0x401d30: mov    %al,(%rdx)
THREAD 2 (4201820)
0x401d5c: mov    $0x4e4484,%eax
0x401d61: mov    (%rax),%al
0x401d63: test   %al,%al
0x401d65: jne    0x401d72 <threadFunc2+34>
0x401d72: mov    0xe2704(%rip),%eax        # 0x4e447c <n>
0x401d78: mov    %eax,0xe2702(%rip)        # 0x4e4480 <y>
0x401d67: LOOP 0x401d72
0x401d6c: mov    (%rax),%al
0x401d6e: test   %al,%al
0x401d70: je     0x401d67 <threadFunc2+23>
Maximum orders: 1001.0
Completed running 1001 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/8_tmo_weaker >> out/accicc_8_tmo_weaker; done"
  45,31s user 26,20s system 83% cpu 1:25,85 total
```

GCC & musl-GCC & Clang

```
THREAD 1 (4201996)
0x401e0c: movl   $0x17,0xdd5c2(%rip)        # 0x4df3d8 <n>
0x401e16: movb   $0x1,0xdd5b3(%rip)        # 0x4df3d0 <x>
THREAD 2 (4202060)
0x401e4c: nopl   0x0(%rax)
0x401e50: LOOP 0x401e5b
0x401e57: test   %al,%al
0x401e59: je     0x401e50 <threadFunc2+32>
0x401e5b: mov    0xdd577(%rip),%eax        # 0x4df3d8 <n>
0x401e61: mov    %eax,0xdd56d(%rip)        # 0x4df3d4 <y>
Maximum orders: 28.0
Completed running 28 orders.
Number failed: 0.

Default run:
bash -c "for run in {1..50000}; do bin/8_tmo_weaker >> out/accgcc_8_tmo_weaker; done"
  45,91s user 26,07s system 80% cpu 1:29,51 total
Failed: 0 / 50000
Incorrect result states: None
```

A.9.2 Without atomics

icc & GCC & musl-GCC

```
THREAD 1 (4201756)
0x401d1c: movl   $0x17,0xe2756(%rip)        # 0x4e447c <n>
0x401d26: movb   $0x1,0xe2757(%rip)        # 0x4e4484 <x>
THREAD 2 (4201804)
0x401d4c: cmpb   $0x0,0xe2731(%rip)        # 0x4e4484 <x>
0x401d53: jne    0x401d57 <threadFunc2+23>
0x401d57: mov    0xe271f(%rip),%eax        # 0x4e447c <n>
0x401d5d: mov    %eax,0xe271d(%rip)        # 0x4e4480 <y>
0x401d55: jmp    0x401d55 <threadFunc2+21>
Maximum orders: 21.0
Completed running 21 orders.
Number failed: 20.
One or more test execution orders failed:
[[2, 2, 1, 1, 2, 2, 2], [2, 1, 1, 2, 2, 2, 2], [2, 2, 2, 1, 1, 2, 2], [2, 2, 2, 2, 2, 1,
     1], [2, 2, 2, 2, 1, 1, 2], [2, 2, 2, 2, 1, 2, 1], [2, 2, 2, 1, 2, 2, 1], [2, 2, 2,
    1, 2, 1, 2], [1, 2, 2, 2, 2, 1, 2], [1, 2, 2, 1, 2, 2, 2], [1, 2, 1, 2, 2, 2, 2],
    [2, 1, 2, 2, 2, 1, 2], [2, 2, 1, 2, 2, 2, 1], [1, 2, 2, 2, 1, 2, 2], [2, 2, 1, 2, 1,
     2, 2], [2, 1, 2, 2, 2, 2, 1], [1, 2, 2, 2, 2, 2, 1], [2, 1, 2, 1, 2, 2, 2], [2, 2,
    1, 2, 2, 1, 2], [2, 1, 2, 2, 1, 2, 2]]

Optimized to include infinite loop if first jne was not taken.
```

```
Default run:
bash -c "for run in {1..50000}; do timeout --signal=SIGINT 0.2 bin/8_tmo_weaker >> out/
    fingcc_8_tmo_weaker; done"   924,08s user 42,90s system 98% cpu 16:18,03 total
Failed: 4282.
Incorrect result states: Infinite loop
```

Clang

```
THREAD 1 (4201629)
0x401c9d: movl   $0x17,0xdf715(%rip)        # 0x4e13bc <n>
0x401ca7: movb   $0x1,0xdf716(%rip)        # 0x4e13c4 <x>
THREAD 2 (4201677)
0x401ccd: cmpb   $0x0,0xdf6f0(%rip)        # 0x4e13c4 <x>
0x401cd4: je     0x401d00 <threadFunc2+64>
0x401d00: LOOP 0x401cd6
0x401cd6: mov    0xdf6e0(%rip),%eax        # 0x4e13bc <n>
0x401cdc: mov    %eax,0xdf6de(%rip)        # 0x4e13c0 <y>

Loops if je is not taken, but not infinite.

Default run:
bash -c "for run in {1..50000}; do timeout --signal=SIGINT 0.2 bin/8_tmo_weaker >> out/
    finclang_8_tmo_weaker; done"  1086,40s user 45,84s system 97% cpu 19:16,13 total
Failed: 5125.
Incorrect result states: Infinite loop
```