Verification of Multithreading Primitives in C

Ruben van den Berge
rubenvandenberge@gmail.com

April 9, 2024, 37 pages

Academic supervisor: Ana Lucia Varbanescu, a.l.varbanescu@uva.nl
Daily supervisor: Vladislav Yaglamunov, vlad@solidsands.nl
Host organisation/Research group: Solid Sands B.V., https://solidsands.com/

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

Abstract

Multithreading is an essential part of modern day programming, as powerful modern multi-core hardware can be fully utilized by parallel programs. To ensure that such programs work as intended, multithreading primitives were introduced in the C11 standard library. These primitives are responsible for the safe communication of threads, synchronization, locking and more. They form the foundation of every multithreaded program.

However, testing these primitives raises issues that come with every form of multithreaded testing. Non-determinism and race conditions make it difficult to eliminate every possible vulnerability. Furthermore, compilers translate source code to machine code. In this process, they also apply certain optimizations for performance and efficiency purposes. It is therefore highly important that both the compiler and the primitives adhere to the agreed upon standards, to prevent undefined behaviour.

In this work, we present a novel method of black-box testing for compilers and C libraries on their behaviour in a concurrent environment. For this purpose, our method executes multiple threads concurrently in fast succession while monitoring potential errors and interleavings. We found that our method is an improvement over a naive test. We also found that our method is able to consistently generate many interleavings with good performance.

Contents

Acknowledgements

I would like to thank my academic supervisor, Ana-Lucia Varbanescu, for her guidance, support and understanding during the process of writing this thesis, which has not always been smooth. I would also like to thank Vladislav Yaglamunov, who guided me through the project with countless ideas. Without your guidance and our discussions about the project, I would have never been able to do this. Finally, thanks to Solid Sands for providing me with the great opportunity to do this project, I have really learned a lot and I wish you all the best of luck in the future.

Chapter 1

Introduction

Multithreading is becoming more and more relevant in software engineering due to ever-increasing hardware performance and parallelisation capabilities, as most modern CPUs are parallel processors [1, 2]. To extract the most out of these modern systems and their capabilities, programmers need to design and develop multithreaded programs. More recently, due to interest in AI rapidly increasing, programmers have also been utilizing the more powerful processing capabilities of GPUs for things other than graphics purposes, like training machine learning models [3, 4].

Besides their performance advantages, multithreaded programs also come with hurdles such as non-determinism and race conditions, which make them hard to test and debug. Specifically, as it is often the OS scheduler that ultimately decides in what order the threads are going to be executed, multithreading often leads to sporadic bugs that only happen when specific conditions are met.

To ensure that multithreaded programs behave properly, C11 [5] has introduced specialized multithreading *primitives* that handle thread synchronization and take care of shared memory, such that thread communication becomes race-free. Such guarantees - race-free behaviour - are essential for safety-critical domains, who are actively using C/C++ for their development.

One such safety-critical domain that utilizes multithreading is the automotive industry, where it is important that that code must operate without fail. It is therefore highly important that these primitives work correctly, but it is difficult to verify whether these primitives are bug-free. Ultimately, this has to do with how the compiler translates these primitives into correct machine-level implementations, ideally with little to no performance loss.

In the automotive industry these compilers have to comply with all relevant safety standards, such as ISO 26262 [6]. The ISO standard describes a *Tool Confidence Level* (TCL) [7] metric, where a higher level means that the tool must be checked more thoroughly. This level consists of two parts: how easy it is to detect that a tool made an error, and how impactful the error is in a tool on the final product. For compilers both of these are at the highest level - it is very hard to notice that compiled machine code has an error and this error can directly cause failure of the system. Therefore, compilers need rigorous testing. Still, compilers are very complex pieces of software that turn source code into machine instructions. Checking whether this translation is correct by design is virtually impossible, and checking it through testing is very difficult.

As most modern software nowadays is compiled with optimizations [8], an extra layer of difficulty is added to testing and debugging: these optimizations often cause the executable code to be significantly different from the source code.

In this work, we aim to address this double challenge - the combined difficulties of compiler testing/verification and non-determinism in buggy multithreaded code - by proposing a new black-box testing approach to find bugs in multithreading C primitives. Black-box testing is a form of software testing that focuses on the input and the output of a program, rather than its internal workings.

The project is realized in collaboration with Solid Sands, a company that builds test suites to assess the adherence of compilers to the C/C++ standards. Companies utilize Solid Sands' software products to asses the C compilers they use. These compilers and libraries may contain company-specific software and hardware, therefore the testing tools need to be portable i.e., it must run on different types of chip architectures and work with a variety of compilers.

In previous work, a lot of multithreaded bug detection has been done on a hardware level, or through code instrumentation. This often requires some target-program- or hardware-specific implementations. In this work, we propose a more general approach: we build a method to test the correctness of compilers

and C library implementations when exposed to a concurrent environment.

To reach our goal, we propose the following requirements for our method:

- it must run regardless of chip architecture (e.g. ARM, x86)
- it must run regardless of the target library, given the library runs on the hardware it is tested on.
- it must be fast – a maximum of a few seconds per run on modern hardware is reasonable, such that our work can be part of a larger test suite.

Responding to these requirements, the goal of this work is to design a method to test C compilers and library functions on their correctness if we expose them to circumstances where data races are most likely to occur. Forcing them to operate in this environment may increase the likelihood of leading to undefined behaviour i.e., behaviour that is not defined in the C standard. The proposed approach can be used to observe incorrect output as a result of data races when testing a non-atomic instruction.

1.1 Research questions and approach

We aim to build a black-box testing method to test C compilers and library functions, including multithreading primitives, on their correctness in a multithreaded environment. This method must be able to monitor the thread order and check whether threads are likely to operate concurrently, which indicates potential data races. If we can find a correlation between the threads operating concurrently and incorrect results, we can use this method to verify the correctness of compilers and C libraries in their standard complicance in a multithreaded environment.

We build this method in a step-by-step manner, answering the following three research questions (RQs):

RQ1: On a source code level, how can we monitor if threads are interleaving?
To be able to propose a suitable method for testing, we first need a way to measure if threads are operating concurrently, and, as a result, might be interleaving. Under these circumstances data races are likely to occur, thus it is relevant for us to monitor when these cases happen.

RQ2: How can we systematically test C library functions on their vulnerability to data races in a multithreaded environment?
We propose to have multiple threads execute an instruction or function simultaneously, many times. Our goal is to have a method to test compilers and functions on their behaviour and vulnerability to data races in a *data-racy environment*. Therefore, the challenges of this approach are to have the threads operate in a narrow window and generate as many combinations of thread orders as possible.

RQ3: How can our method be used to detect multithreading vulnerabilities in a C library? Our approach to this question is to take an atomic function from a C library, and purposefully create a vulnerability in that function. A vulnerability is a flaw in code that can lead to bugs when the right set of circumstances is met. Regular testing methods will rarely trigger such bugs. Our method is used to try to recreate the specific circumstances in which the bugs will occur.

1.2 Outline

The remainder of this thesis is organised as follows. In Chapter 2 we introduce the background knowledge required to understand our work. Chapter 3 briefly discusses previous work related to this thesis. In Chapter 4 we discuss how we approach the forced interleaving and monitoring of interleavings. Our results are shown and discussed in Chapter 5, along with an explanation of our testing methods. Finally, we present our concluding remarks in Chapter 6 together with future work.

Chapter 2

Background

In this section we present the basic notions needed to understand the remainder of this work. We specifically introduce the relevant notions around programming, testing, and debugging multithreaded applications in C.

2.1 Multithreading

Multithreading is a technique that enables a program to use multiple concurrent threads, thus potentially enabling multiple tasks to be executed in parallel. Each of these tasks is then executed by a *thread*. In parallel systems, these threads often run on different processors or different cores, thus enabling true parallel program execution. However, they are all part of the same process. Threads are usually spawned by a main thread. All these spawned threads share the same memory space, can share variables with the main thread and with each other, and often communicate using shared variables. After creation, threads can also declare and use "local variables", which are private to the threads themselves.

2.2 Data Race

For safe communication between threads, proper *synchronization* is required. Thread synchronization ensures that shared memory cannot be accessed by two threads at the same time. An example of a synchronization mechanism is a *lock*, which allows only one of the threads to access the shared memory at one point in time. Take an example with two threads, A and B. If a lock is acquired by thread A, then thread B has to wait until thread A releases that lock before it can acquire that lock. If synchronization is not correctly enforced by the programmer, communication through shared variables can result in data race conditions. Such data races occur when two threads access the same shared-memory resource, concurrently, and at least one of the actions they perform is a write operation. Data races can cause a program to output incorrect results, or even crash altogether. Furthermore, data races can introduce code vulnerabilities that malicious attackers can exploit [9].

A drawback of these synchronization operations is that they reduce the concurrency of the program. When using locks, two threads cannot perform an operation on the shared variables anymore, and they are executed in sequence.. As such, with performance as an objective of parallelism and multithreading, it is preferable to minimize synchronization. However, this is difficult to do in practice, and programmers tend to either add too many or too few synchronization points.

2.3 Atomics

An operation is atomic if it in indivisible, and thus can execute to completion uninterrupted. In a concurrent context, an atomic operation (in short, an *atomic*) executed by one thread cannot be interrupted by another operation from other threads. For example, take the `a += 1;` C instruction: when executed, it gets divided into three operations read `a`, add `1`, and write `a`. With a *nonatomic* addition, these three instructions can be interleaved with instructions/operations from other threads, potentially leading to incorrect results due to data races. An *atomic* add ensures all three instructions execute in order to completion, hence the multithreaded execution is race-free.

Atomics also respect the *happens-before* relation. This relation means that if one of multiple events happens before the other(s), the results must reflect that.

To create atomic operations, some form of synchronization is needed. In C and C++, the `<stdatomic.h>` header from the C standard library implements atomic objects and functions, ensuring there are no data races when these are used in a multithreaded context. For simplicity, we call a library that implements atomics an *atomics library*.

2.4 Compilers and optimizations

> "*The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language.*" [10]

In other words, a compiler is a program that translates some human-written source code into something executable by a computer. Thus, compilers also play a role in the behaviour of multithreaded programs. They ultimately decide what the computer has to execute, and what the scheduler has to decide goes first. Compilers can apply optimizations to the code where they deem necessary, and these optimizations modify the generated machine code drastically. GCC provides the user the option to set a level of optimization from no extra optimizations to multiple sets of optimizations with the use of the `-OX` flag [11]. Take for example the following code snippet:

```
atomic_int a = 10;

int main() {

    a += 10;
    return 0;
}
```

We compile this small code snippet with a GCC compiler (version 11.3.0). This will yield the following assembly code when no optimization flags are used (`-O0`):

```
a:
        .long   10
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-8], 1
        mov     eax, DWORD PTR [rbp-8]
        mov     edx, eax
        mov     eax, edx
        lock xadd       DWORD PTR a[rip], eax
        add     eax, edx
        mov     DWORD PTR [rbp-4], eax
        mov     eax, 0
        pop     rbp
        ret
```

And the following assembly code is the result of using optimization flag `-O3`:

```
main:
        lock add        DWORD PTR a[rip], 1
        xor     eax, eax
        ret
a:
        .long   10
```

There are many more instructions when no additional optimizations are applied. This means that there are fewer combinations of interleavings for the instructions when there are multiple threads executing code with additional optimizations.

2.5 SuperTest

Solid Sands has developed a test suite for the validation of compilers, called SuperTest. Currently, it has some tests for multithreaded programs and instructions, but it does not yet test for multithreading properties such as data races. As such, this method could also be incorporated into SuperTest. SuperTest has many individual tests, and each test can be performed on different compilers. Therefore, our method needs to be made with performance as a priority. With performance in mind, we implemented the monitoring of the thread interleavings using very fast instructions. This is done with the intent of being able to do many iterations in seconds.

2.6 x86 and ARM

Most of the tests are run on an AMD Ryzen 7 5800H CPU, based on an x86_64 architecture with a Zen 3 microarchitecture. The CPU consists of 8 cores with two threads per core, making a total of 16 threads. SuperTest aims to test many different compilers while also being able to run regardless of operating system and CPU architecture. Therefore, our method needs to work effectively with multiple compilers and CPU architectures. For this reason, we have also tested our program on an ARM Raspberry Pi CPU. Although this CPU only contains one core with four threads, it still supports our method. To get a good comparison between the results on the x86 machine with 16 threads and the ARM machine, we opted to test only for a maximum of four threads.

ARM processors make use of a *reduced instruction set computer* (RISC) microprocessor, while x86 CPUs make use of a *complex instruction set computer* (CISC) microprocessor. RISC is designed to reduce the complexity of each instruction by having simpler but more instructions, whereas CISC utilizes more complex instructions to reduce the number of instructions. This has an impact on the machine code the compiler generates and therefore on the way a multithreaded program may interleave. RISC systems generate more instructions, therefore there are more interleaving combinations. On the other hand, since the execution speed of these simpler instructions is faster, there is less of a gap between instructions which reduces the chance of interleavings occurring.

2.7 Compare and Swap

In an ideal world, locks are not needed. Locks are slow, so we prefer to use lock-free operations. One such operation is the *compare-and-swap* (CAS) operation. The goal of a CAS is that a (shared) variable only gets updated if it still has the same value as when it fetched the value of that variable. For example, if two threads `A` and `B` try to modify a variable `x` which has a value of 5. `Thread_A` modifies the value to 10 first, which means that when `thread_B` tries to execute the CAS it fails, because `x` does not equal 5 anymore. `Thread_B` then keeps attempting the CAS until it succeeds. Unfortunately, this means that `thread_B` can get stuck if it re-fetches the value every time while other threads keep updating the value. However, it still guarantees that there will be global progress to the total value of the shared variable. A CAS is an atomic operation. If a CAS operation fails, it has to repeat the process of reading the memory location of the variable again. As this happens very quickly and very often, this can hurt the performance of the program. To combat this, modern hardware and CAS algorithms have a small delay before retrying the CAS loop. [12].

2.8 ABA problem

CAS algorithms also introduce problems. Since it is a lock-free operation, it can introduce concurrency bugs. A well-known concurrency problem is the *ABA problem*. The ABA problem occurs when multiple threads read and write shared memory concurrently (i.e., the threads are interleaving). Take the following example where two threads A and B attempt to modify a shared variable `x`:

1. Threads A and B read the original value from `x`.
2. A precedes B, so A writes a new value to x.
3. A now writes a value equivalent to `x`'s original value to `x`.
4. B reads the value of x which is now the same as the original value. This means that thread B can proceed and modify `x`.

Even though thread A's modification looks bogus, in some cases this "redundant" modification can lead to unintended or incorrect behaviour. A list that has an element removed and then added will cause issues with pointers, as the original pointer to the removed element will often point to the new element [13]. In our work, we test our method using addition and subtraction, thus the newly written value to shared memory can never be the same as the old value. Therefore, we do not encounter ABA problems.

Chapter 3

Related work

In this chapter, we present existing literature related to the focus of our work: multithreading and its primitives. Specifically, we focus on bug-detection approaches in parallel programs. We discuss the approach of various papers and their relation to our work, the differences compared to our research, and their limitations.

A known method for finding bugs is *fuzz testing*, also known as *fuzzing*. Fuzzing has also been used to test multithreaded programs for data races, often by stressing the system to force the scheduler to make decisions on the order of threads, attempting to create uncommon interleavings. However, it does so by utilizing code instrumentation, a technique that adds code to an existing program to track its behaviour. For this reason, fuzzing is beyond the purpose of this work. Similarly, model checking can be used for executing each thread schedule. However, this can lead to the state-explosion problem. Our work focuses on testing small bits of code for a large number of times, making model checking not a viable solution.

3.1 Testing multithreaded programs

3.1.1 Coverage-based testing

Coverage-based testing is a form of software testing that uses a metric called *coverage* to determine the extent to which the source code is executed by test cases. In the context of multithreading, coverage metrics can be more arbitrarily defined. Therefore, we are not just looking at lines of code covered by tests, but we also have to look at things such as interleavings, memory dependencies, and synchronization operations.

For example, Yu et al. [14] attempt to define coverage metrics for multithreaded programs by a set of interleavings. They proposed a tool called Maple for coverage-based testing of multithreaded programs. This tool aims to expose untested thread interleavings that could cause rare concurrency bugs. It does so by tracking and memoizing explored interleavings, and actively controlling the thread schedule to execute unexplored interleavings. However, Maple does not guarantee to find more interleavings than random and systematic testing tools, nor is it able to always find and test all interleavings. They argue that even though systematic testing suffers from scalability problems, it can provide guarantees in finding concurrency bugs. For our work, we focus on the generation of different interleavings in some run and we do not focus on the generation of specific unexplored and rare interleavings.

3.1.2 Deterministic approach

One of the largest difficulties in multithreaded testing lies in the non-determinism of multithreaded programs. Eliminating this means ensuring all possible interleavings are exhausted when testing multithreaded programs, making the program deterministic. For example, in previous work for Solid Sands, Rick Watertor et al. [15] have tried to assess the correctness of instructions concerning atomics and fences. They do so by using the GNU Debugger to execute each instruction one by one, going through all interleavings manually. Whilst this is a very interesting approach, the method does not detect bugs that are triggered by threads executing concurrently. Moreover, their method loses the purpose of multithreading, as the processes are so independent of each other that you could run processes in different GDB instances. Our work draws heavy inspiration from this work and can be regarded as a follow-up to this paper. Compared to their work, the goal of our research is the same, but our approach is differ-

ent. We aim to build a significantly faster method that can detect concurrency bugs when threads are executing at the same time, by forcing threads to operate at the same time. This unfortunately leads to our method not being deterministic, but compared to the work by Watertor et al. our method can be applied to real test suites more easily.

T. Liu et al. present `Dthreads` [16], which is a deterministic multithreading library aimed at replacing the `pthreads` library. Rare bugs which they call "Heisenbugs" only happen under specific interleavings. Their goal is to remove the hassle of debugging such bugs by removing the biggest problem of multithreaded testing, non-determinism. By making their tests deterministic, the bugs can be reproduced consistently. The `pthreads` library does not offer any determinism in processes such as synchronization and locks, whereas `Dthreads` does not allow interleavings at so-called synchronization points. However, to make this deterministic each thread cannot perform updates on shared memory. Instead, they each have some 'private' memory that gets synchronized with the other threads at the synchronization points, which always happens in the same order to make it deterministic. However, `Dthreads` does not support programs that use *ad hoc* synchronization, which is a common practice even though ad hoc synchronization might lead to bugs [17]. Our work is intended to test any multithreading primitive, which includes primitives used for ad hoc synchronization.

3.1.3 Atomicity bugs

CTrigger is a testing framework used to study hard-to-expose atomicity violation bugs [18]. Their work focuses on studying unserializable interleavings, which means there is no sequential equivalent for these interleavings. By ranking the interleavings on their probability and then reproducing the low-probability interleavings, they effectively expose bugs. They rank the interleavings based on the gap between instructions, with the gap being the execution time between two local instructions. They argue that the larger this gap is, the greater the chance for a different thread's instruction to interleave. Although their statistics indicate this is an effective method, it is anything but guaranteed that a specific interleaving is more likely to occur when the gap is bigger. On top of that, this requires very precise control of the timing and execution speed of the threads. Thus, such methods are not able to be run thousands of times within a very short period, contrary to our work.

3.1.4 Other tools

In this section, we discuss some existing tools for detecting concurrency bugs that do not fall into the other categories, and why our research is different from these tools. One such tool to detect, measure, and annotate data races is *ThreadSanitizer* [19]. This tool, presented by Serebryany et al., observes the execution of a program as a sequence of events, with an emphasis on the *memory access* and the *synchronization* events. These events are the most important for a multithreaded program to work properly. However, while this tool works well and is used by companies such as Google, this tool has some drawbacks. Although there may be some vulnerability in a program for a data race, it may not detect the data race if it does not occur. Furthermore, if the program is badly written and the programmer uses more synchronization and locks than necessary, ThreadSanitizer might not pick up data races even though they do happen. In our work, we are more focused on a much smaller scale. Furthermore, our method applies to multiple compilers and can be used for many different tests built into a test suite.

Another method to test multithreaded programs does so via controlling "thread speeds", presented by Chen et al. [20]. In this work, they present the interleaving space as a *speed space*. By manipulating the speed at which each thread operates they can systematically explore different interleavings, allowing them to create uncommon interleavings. This helps them expose hard-to-find bugs in multithreaded programs, which only occur when a certain interleaving is explored. There are some limitations with this work, however. There is no guarantee that all interleavings are exhausted, and all concurrency bugs are found. Moreover, they add significant overhead to the program as a result of instrumenting existing frameworks. Our work does not induce such overhead, and while our method does not control thread speeds, we aim to control the timing of the execution of multiple threads to be the same.

In summary, our work is different from testing multithreaded programs because it creates a concurrent setting that increases the likelihood of threads interleaving, therefore increasing the chances of concurrency bugs occurring. Whereas most of the literature focuses on bug detection by detecting rare bugs, our work also attempts to increase the bug occurrence rate. It is interesting to see whether it is

possible to force threads to operate concurrently and induce rare concurrency bugs that way. Our work draws inspiration from other works such as previous work done at Solid Sands by Watertor et al. and *ThreadSanitizer* [10, 19]. We further proceed to explain our methods and evaluation in the next chapter.

Chapter 4

Methodology

In this chapter, we present our methodology for building a method to force threads to operate as concurrently as possible, by creating circumstances where data races are most likely to happen. This method can be used to assess standard-compliance of compilers and libraries.

4.1 Method overview

The goal is to build a method to assess correctness of library function implementations without using any form of instrumentation, thus on a source code level. In order to be able to assess correctness of such functions work as intended, we need to test these functions on their behaviour in a highly concurrent environment. As multithreaded programs are non-deterministic, our method needs to be at least able to consistently force threads to operate concurrently. Furthermore, if the library's implementation is correct, any atomic operation performed when exposed to a highly concurrent environment should always result in the expected outcome. The method also needs to be able to monitor whether threads are operating concurrently, to improve consistency and performance. Lastly, it should take no longer than a few seconds to generate a sufficient amount of thread interleavings.

We can break this down into four points:

1. Consistently force threads to operate concurrently.
2. Monitor whether threads are interleaving, as this increases the chance of potential errors.
3. Assess whether the outcome of the current execution is the same as the expected outcome.
4. Generate many thread interleavings with a maximum runtime of a few seconds.

For the first point, our approach is to have all threads start executing every iteration of the algorithm at the same time, such that the scheduler is at least forced to make decisions with regard to the order of the threads. As a result, we end up with many different thread orders.

As for the second point, we approach the monitoring of the thread orders by building it directly before and after the instruction we want to test. The monitoring instructions of the interleavings are done by assigning variables corresponding to the order in which the thread functions are executed. In order to prevent the monitoring from adding much overhead to the program which could cause the threads to get delayed, suspended or hampered by other forms of timing disruption, we have designed the monitoring instructions to be very lightweight. As they are just an atomic addition and a variable assignment, it keeps the added overhead to a minimum.

The third point is dependent on the instruction the method needs to test, therefore, the expected value needs to be adjusted on a test-case basis. For example, the result of an integer addition is simply `RESULT = BASE + N_THREADS * INCR`, where `BASE` is the initial value of a variable, `N_THREADS` is the number of threads and `INCR` is the value we want to increase the base value with. In case of the example below in listing 4.1 `BASE` is 10, `N_THREADS` is 2, and `INCR` is 5, which brings the expected total to 20. However, for something like an atomic `XOR` function, this is a bit more tricky. With our method, the end-user has to determine what the expected value is and program that themselves.

Lastly, the program needs to run until sufficient thread interleavings have been found. As we cannot generate thread interleavings 100% of the time, it is inefficient to set a specific amount of iterations the algorithm has to execute before terminating, as that can lead to too few thread interleavings generated to confidently say that a test has passed. Instead, the algorithm runs until it has found a specific number

of thread interleavings. This increases the consistency of the program, as data races occur at a higher rate when these thread interleavings take place. A schematic overview of the algorithm is presented in Figure 4.1

```
#include <stdatomic.h>

    atomic_int x = 10;          // Global atomic integer.
    int i = 10;                 // Global integer.

    thread_func_A() {
        x += 5;                     // Add 5 to x, an atomic object.
        atomic_fetch_add(&i, 5);    // Add 5 to i using an atomic add function.
    }

    thread_func_B() {
        x += 5;                     // Add 5 to x, an atomic object.
        atomic_fetch_add(&i, 5);    // Add 5 to i using an atomic add function.
    }
```

Listing 4.1: Simplified example of potential test cases for our method. We have two shared global variables, an atomic integer `x`, and an integer `i`. We have two threads increase both of these shared variables by 5, thus we expect the end result for both of these variables to be 20.

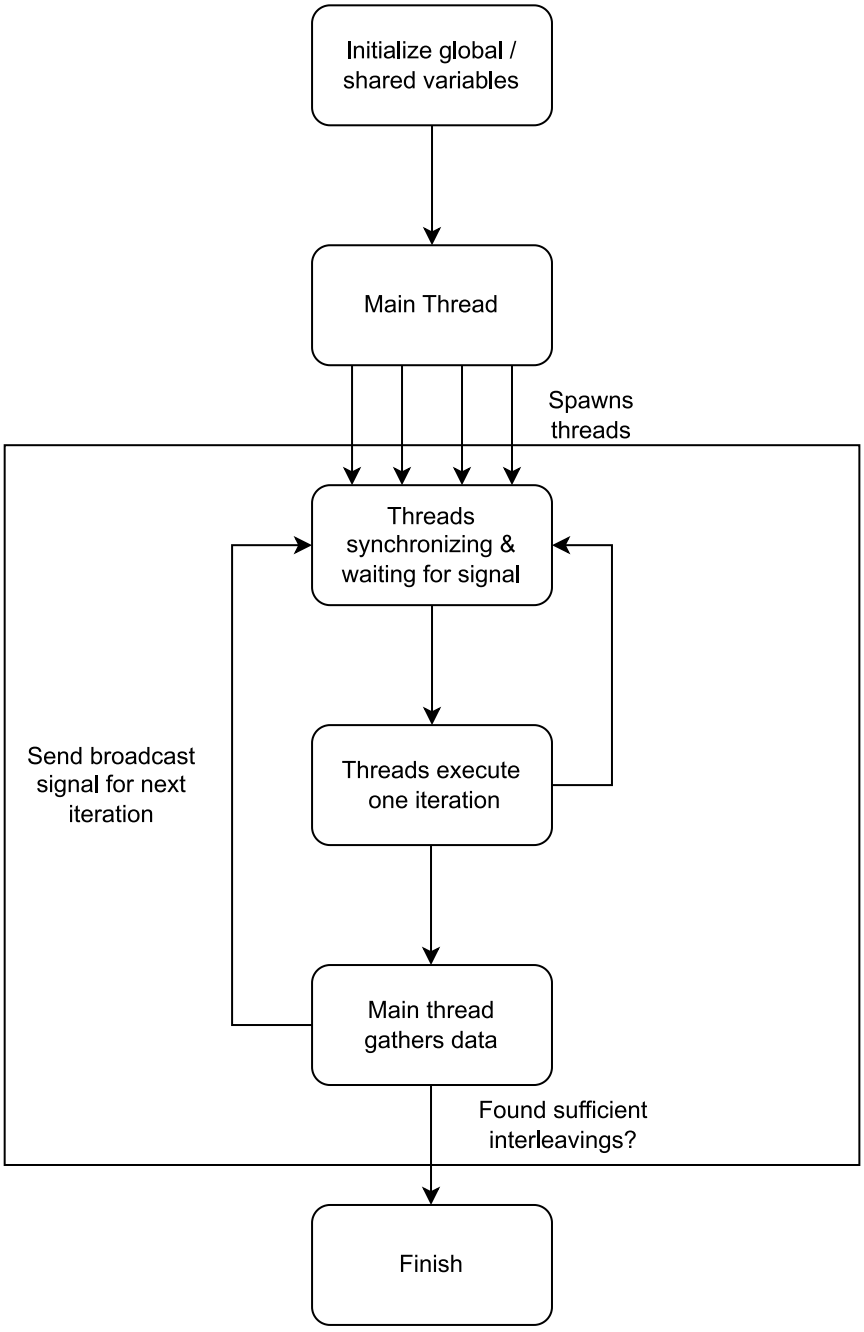

Figure 4.1: This flowchart demonstrates schematically and globally how the algorithm works. First, it initializes all necessary shared variables. Then, the main thread spawns any number of threads between 2 and 4. Once the threads are spawned with their corresponding thread functions, they all move into a waiting state. When all threads are in the waiting state, the main thread sends a broadcast that all threads can execute an iteration. Once the iteration is done, the threads return to their waiting state again. The main thread gathers all necessary results from that one iteration and resets all necessary variables back to their original state. After the main thread has finished doing this for an iteration, it sends a signal to all threads again that they can execute an iteration again. This is repeated until the desired number of interleavings has been found. Once the algorithm is finished, it outputs the results.

4.2 Monitoring

As mentioned in Section 4.1, we need a method to monitor if threads are operating in a narrow window to measure concurrency. Since we only need to test one instruction at a time, we are able to build the monitoring tools directly around it. An example of this is demonstrated in listing 4.2.

```
int x = 10;          // Global variable

thread_function() {
    monitoring_instructions;

    x += 5;          // Instruction to assess

    monitoring_instructions;
}
```

Listing 4.2: Pseudocode example of thread concurrency monitoring.

One way to measure concurrency is to have multiple threads adjust shared variables "at the same time" or at least close to, and then per iteration measure if we can see differing thread orders for the modification of the shared variables. For this purpose, we create a 2D array to represent the thread order in an `8 x N` matrix, where N is the number of threads. To demonstrate this, we have created the following example below:

```
int a[8][N_THREADS]; // 2D array of thread order indicators.
int x = 10;

thread_function(int thread_id) {

    for N iterations {
        wait_for_main_thread_signal;

        a[0][thread_id] = current_thread_order;
        a[1][thread_id] = current_thread_order;
        a[2][thread_id] = current_thread_order;
        a[3][thread_id] = current_thread_order;

        x += 5;          // Instruction to assess

        a[4][thread_id] = current_thread_order;
        a[5][thread_id] = current_thread_order;
        a[6][thread_id] = current_thread_order;
        a[7][thread_id] = current_thread_order;
    }
}
```

Listing 4.3: (Pseudo)code of the thread functions of the tool. Every iteration we synchronize the functions by letting them all wait for the main thread to send a signal to the thread functions that they can execute an iteration.

4.2.1 Atomic Fetch Add

For the `current_thread_order` demonstrated in listing 4.3, we use the function `atomic_fetch_add` [21]. This function is defined as demonstrated in listing 4.4.

```
atomic_fetch_add( volatile A* obj, M arg );
atomic_fetch_add_explicit( volatile A* obj, M arg, memory_order order );
```

Listing 4.4: C `atomic_fetch_add` as defined in header `<stdatomic.h>`. Atomically replaces the value pointed by `obj` with the result of addition of `arg` to the old value of `obj`, and returns the value `obj` held previously [21].

Because this function returns the value `obj` previously held, we can use this function to see in which order all threads execute. For example, if we set shared variable `indicator` to 1 initially, we can use `atomic_fetch_add` to increment `indicator` by 1 for every thread that does one iteration of the method. The first thread that increments `indicator` by 1 with `atomic_fetch_add` will receive 1 as return value. The second thread modifying `indicator` will then have 2 returned, and so forth. It then stores the return value into the 2D thread order matrix corresponding to the column of the thread currently executing.

If thread 1 is first in the execution order, then the first column of the matrix will be 1 for every value that is first updated by thread 1. Similarly, if thread 2 is third in the execution order, then the second column of the matrix will be 3 for every value that is updated third by thread 2.

Order of threads: t1, t2, t3, t4

```
a0: 1  2  3  4
a1: 1  2  3  4
a2: 1  2  3  4
a3: 1  2  3  4
```

Order of threads: t4, t2, t1, t3

```
a0: 3  2  4  1
a1: 3  2  4  1
a2: 3  2  4  1
a3: 3  2  4  1
```

Figure 4.2: Example of the internal representation of the 2D matrix with the thread orders. Each row represents the order in which a shared variable is adjusted

4.3 Thread Synchronization

Every iteration of the method the thread functions are synchronized using the `pthread_cond_wait` command. It depends on a conditional variable and a mutex. The main thread sends a signal using the `pthread_cond_broadcast` command to each of the thread functions. This releases all threads currently blocked by `pthread_cond_wait` at the same time, effectively releasing every thread simultaneously. Each thread now starts executing. After the thread functions finish executing one iteration, all threads start waiting again. The main thread then collects all necessary information from that execution. Once the main thread is done, the next iteration starts and it broadcasts a signal that the thread functions can start executing again. Utilizing this method, we can force the threads to operate concurrently more consistently, even though the executions are still non-deterministic. This is done to force concurrency in every single iteration of the loop and to prevent the threads from getting out of sync.

4.4 Thread interleavings

Since we are only interested in cases where data races are likely, our method looks for cases where the thread order differs between variables. For example, if the thread execution order with four threads is $1, 2, 3, 4$ for one variable, but $1, 3, 2, 4$ for another variable, then it is more likely that the threads will have interleaved. If the threads have a higher chance of interleaving, then the chance of a data race occurring also increases. For this reason, our method is only interested in these cases, where we observe different thread orders for different variables. This is demonstrated in Figure 4.3.

Monitoring before instruction to assess

```
a0: 1  2  3  4
a1: 1  2  3  4
a2: 1  3  2  4
a3: 1  2  3  4
```

Monitoring after instruction to assess

```
a4: 1  2  3  4
a5: 1  2  3  4
a6: 1  2  3  4
a7: 1  2  3  4
```

Figure 4.3: Example of an execution with different thread orders for the same iteration of the method. The thread order for variable `a2` is different from the rest. For all but one variable, the thread order is $1, 2, 3, 4$. The odd one out is `a2`, whose thread order is $1, 3, 2, 4$.

4.5 Thread execution order

We experiment with different orders for the thread monitoring instructions. Prior to and after the test subject, we execute four operations that track the execution order of the different threads. Our method has two execution order modes:

- Same execution order - in this mode, every monitoring operation by each thread is performed in the **same** order; 1, 2, 3, 4 for operations before the test subject, and 5, 6, 7, 8 after the test subject.

- Different execution order - in this mode, every monitoring operation by each thread is performed in a **different** order;
  - `thread_1`: 1, 2, 3, 4 - test subject - 5, 6, 7, 8
  - `thread_2`: 4, 3, 2, 1 - test subject - 8, 7, 6, 5
  - `thread_3`: 1, 3, 2, 4 - test subject - 5, 7, 6, 8
  - `thread_4`: 2, 4, 1, 3 - test subject - 6, 8, 5, 7

This is done with experimental purposes. We see whether it has any impact on the consistency of interleaving generation, the amount of errors, the percentage of interleaved errors etc.

Chapter 5

Evaluation

In this chapter, we present our approach for the evaluation of our method and the results of the evaluations.

5.1 Experimental setup

All of our experiments are run on two machines: AMD and R-Pi. Most of the tests are run on the AMD machine, which features an AMD Ryzen 7 5800H CPU, based on an x86_64 architecture with a Zen 3 microarchitecture. Some of the tests are also run on R-Pi, a Raspberry Pi 4 Model B Rev 1.5, with a Cortex-A72 CPU based on an aarch64/armv8 architecture. We use three different compilers for our test: `GCC`, `musl-gcc`, and `clang`. We chose these compilers because they are commonly used, readily available open-source compilers.

For all but the experiments in Section **??** we run the program on four threads until the program has generated 100000 interleaved runs. To show the method is consistent, we run the program five times for each test case. The test is compiled four different times with some optimization flags: `-O0`, `-O1`, `-O2`, and `-O3`. We find it important to understand what impact the optimization flags have on the program's ability to generate interleaved runs and the impact they have on the amount of data-race errors.

5.2 Test cases

For the testing and validation of our method, we focus on the following test cases:

- Non-atomic operations, such as regular additions and subtractions. This experiment will serve as the baseline to prove our method can reliably detect concurrency bugs and force interleavings.
- Atomic operations, such as `atomic_add` and `atomic_sub`. This experiment will be done to prove our method does not pick up false positives, as we expect there to be no concurrency bugs in this library.
- A hard-to-detect atomic library bug. This experiment will show that our method can detect rare concurrency bugs, whereas a naive test would not be able to.

We have to prove that for non-atomic operations our method picks up errors, as well as for the hard-to-detect atomic library bug. If our method can reliably prove that there is an error with these implementations, we can utilize the method to reliably prove the atomic library is working as intended if we never pick up any error. In this context, an error can mean that operations are non-atomic, resulting in unexpected behaviour or there is a bug somewhere in a library's function causing wrong results.

As described in section 4.4, our method also detects thread interleavings. After we observe that threads are interleaving, we measure if the result of the operation on `x` yields the expected result. The program keeps track of eight metrics that we use to analyze a run:

- **Total runs** - We run the program until the given amount of iterations has been reached. As not every run of the program will lead to interleaved runs, this number varies and is larger than the total interleaved runs.
- **Total interleaved runs** - An **interleaved run** indicates that our method detected that threads have been interleaving for this iteration of the program. This does not mean that our method

detects all runs with interleavings, as the method is only able to detect interleavings based on our measurement variables. This number should always match the given number of desired interleaved runs. In our test cases, this is always `100000`.

- **Total errors** - In every iteration, all threads perform an arithmetic operation, resulting in a number. If this number does not match the expected result, we call this an **error**. Total errors indicate the total amount of errors in a run.
- **Percentage of interleaved runs** - The number of total interleaved runs divided by the number of total runs.
- **Percentage of errors** - The number of errors divided by the number of total runs.
- **Percentage of interleaved errors relative to total runs** - An *interleaved error* means that the program detected an error and in this specific run the threads have interleaved. We keep track of the amount of interleaved errors and divide this number by the number of total runs to get this metric.
- **Percentage of interleaved errors relative to total errors** - The number of interleaved errors divided by the number of total errors. This is the most important metric we use, as it indicates the impact the interleavings have on the amount of errors. If this percentage is high, we can argue that the interleavings have impacted the error rate, and prove that our method succeeds in forcing interleavings and consistent data race generation.
- **Percentage of interleaved errors relative to total interleaved runs** - The number of interleaved errors divided by the number of interleaved runs. We use this metric to see how many interleaved runs have resulted in an error.

5.2.1 Non-atomic operations

We use *non-atomic* operations to demonstrate that our method can consistently force threads to operate concurrently, therefore increasing the amount of data races resulting in more concurrency errors. Subsequently, we apply our approach to assess atomic operations, aiming to identify any potential bugs that may arise during parallel execution. A non-atomic operation is not guaranteed to be executed as an indivisible or uninterruptible unit. Non-atomic operations can be interrupted or interleaved by other operations, leading to potential race conditions. We use these operations in our method to demonstrate that our method can reliably force data races. In this simple experiment, we use the addition (`+`) and subtraction (`-`) operators. The simplicity of these operations allows for easy measurement of the outcome of the run. If a variable `x` starts at 0, and the four threads each add `1` to `x`, the expected total of `x` at the end of a run is 4. If the actual outcome of the program differs from 4, we know that a data race happened. In reality, we assign a thread indicator to every thread (ranging from 1 to the number of threads) and let each thread add its squared thread ID to `x`. For example, `thread_2` will add four to the total of `x`. Our program works with a maximum of four threads, leading to squared thread IDs of 1, 4, 9, and 16. These are all unique numbers, which allows us to see which thread did not execute correctly if `x` does not equal the expected amount.

Figure 5.1 demonstrates the results of a single test run.

```
Total runs:                                                    461446
Total interleaved runs:                                        100000
Total runs with errors:                                        114836
Percentage of interleaved runs:                                 21.67
Percentage of errors:                                           24.89
Percentage of interleaved errors (interleaved errors / total runs):  10.95
Percentage of interleaved errors / total errors:                43.98
Percentage of interleaved errors / total_interleavings:         50.51
```

Figure 5.1: This figure shows the results of a test run with four threads and `100000` interleaved runs. The same execution order mode is used. It took the program 461446 runs to get to 100000 interleaved runs. This test is compiled with the `-O0` optimization flag.

The results of our experiments where threads execute the monitoring instructions in same execution order mode are demonstrated in tables 5.1, 5.2, 5.3, and 5.4.

Test	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	416038	101651	24.04	24.43	12.00	49.12	49.93
2	454042	105544	22.02	23.25	11.49	49.41	52.15
3	472451	134193	21.17	28.40	10.29	36.21	48.60
4	509969	130894	19.61	25.67	9.24	36.02	47.14
5	469658	136089	21.29	28.98	10.15	35.02	47.66
Average	464831.6	121274.2	21.426	26.546	10.234	41.356	49.496

Table 5.1: The results of five runs of the program on an x86 architecture are presented in this table. Run with same execution order mode. Each test was run until 100000 interleaved runs were generated and each test was run on four threads. The tests are compiled with no optimizations, i.e. the `-O0` optimization flag is used.

Test	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	620231	17359	16.12	2.80	1.36	48.71	8.46
2	592879	18611	16.87	3.14	1.44	45.73	8.51
3	515578	18615	19.40	3.61	1.73	47.96	8.93
4	598216	17328	16.72	2.90	1.42	48.96	8.48
5	638111	16214	15.67	2.54	1.46	57.46	9.32
Average	631403	17625.4	16.756	2.998	14.822	49.164	8.94

Table 5.2: The results of five runs of the program on an x86 architecture are presented in this table. Run with same execution order mode. Each test was run until 100000 interleaved runs were generated and each test was run on four threads. The tests are compiled with the `-O1` optimization flag.

Test	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	510860	19517	19.57	3.82	1.63	42.76	8.34
2	581522	21160	17.20	3.64	1.69	46.40	9.82
3	622358	17399	16.07	2.80	1.38	49.34	8.59
4	557574	18775	17.93	3.37	1.62	48.01	9.01
5	629512	17119	15.89	2.72	1.44	52.97	9.07
Average	580965.2	18714	17.332	3.066	1.552	47.896	8.766

Table 5.3: The results of five runs of the program on an x86 architecture are presented in this table. Run with same execution order mode. Each test was run until 100000 interleaved runs were generated and each test was run on four threads. The tests are compiled with the `-O2` optimization flag.

Test	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	623379	16855	16.04	2.70	1.47	54.29	9.15
2	578114	18379	17.30	3.18	1.69	53.17	9.77
3	701016	18731	14.27	2.67	1.35	50.58	9.48
4	612090	18120	16.34	2.96	1.40	47.35	8.58
5	575797	17971	17.37	3.12	1.64	52.48	9.43
Average	618679.2	18011.2	16.46	2.92	1.51	51.77	9.10

Table 5.4: The results of five runs of the program on an x86 architecture are presented in this table. Run with same execution order mode. Each test was run until 100000 interleaved runs were generated and each test was run on four threads. The tests are compiled with the `-O3` optimization flag.

The results of our experiments where threads execute the monitoring instructions in different execution order mode are demonstrated in tables 5.5, 5.6, 5.7, and 5.8.

test	Runs	Errors	% interleaved runs	% errors	% interleaved errors / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	159565	39447	62.67	24.72	24.08	97.40	38.42
2	163077	40933	61.32	25.10	24.29	96.76	39.61
3	140168	45575	71.34	33.23	32.88	98.96	46.09
4	196481	40934	50.90	20.83	19.60	94.10	38.52
5	160872	44613	62.16	27.73	26.40	94.62	42.21
Average	164032.6	42300.4	61.68	26.32	25.45	96.37	40.97

Table 5.5: The results of five runs of the program on an x86 architecture are presented in this table. Run with different execution order mode. Each test was run until 100000 interleaved runs were generated and each test was run on four threads. The tests are compiled with no optimizations, i.e. the `-O0` flag is used.

Test	Runs	Errors	% interleaved runs	% errors	% interleaved errors / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	204370	5249	48.93	2.57	2.56	99.60	5.23
2	211605	5681	47.26	2.68	2.67	99.37	5.64
3	197535	6283	50.62	3.18	3.16	99.33	6.24
4	217239	6351	46.03	2.92	2.91	99.54	6.32
5	214119	6407	46.70	2.99	2.97	99.19	6.36
Average	208973.6	5994.2	47.91	2.87	2.85	99.41	5.96

Table 5.6: The results of five runs of the program on an x86 architecture are presented in this table. Run with different execution order mode. Each test was run until 100000 interleaved runs were generated and each test was run on four threads. The tests are compiled with the `-O1` optimization flag.

Test	Runs	Errors	% interleaved runs	% errors	% interleaved errors / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	200569	8245	49.86	4.11	4.09	99.43	8.2
2	189200	6474	52.85	3.42	3.40	99.41	6.44
3	216894	7157	46.11	3.30	3.28	99.39	7.11
4	188086	7618	53.17	4.05	4.02	99.23	7.56
5	194145	6021	51.51	3.10	3.09	99.50	5.99
Average	197778.8	7103	50.7	3.60	3.58	99.39	7.06

Table 5.7: The results of five runs of the program on an x86 architecture are presented in this table. Run with different execution order mode. Each test was run until 100000 interleaved runs were generated and each test was run on four threads. The tests are compiled with the `-O2` optimization flag.

Test	Runs	Errors	% interleaved runs	% errors	% interleaved errors / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	181736	7614	55.02	4.19	4.18	99.83	7.60
2	190824	7267	52.40	3.81	3.80	99.66	7.24
3	195581	7118	51.13	3.64	3.61	99.28	7.07
4	212534	7259	47.05	3.42	3.40	99.63	7.23
5	214503	6487	46.62	3.02	3.00	99.32	6.44
Average	199035.6	7149	50.44	3.62	3.59	99.54	7.12

Table 5.8: The results of five runs of the program on an x86 architecture are presented in this table. Run with different execution order mode. Each test was run until 100000 interleaved runs were generated and each test was run on four threads. The tests are compiled using the `-O3` optimization flag.

From these results, we can conclude that there is a correlation between thread interleavings and the errors that occur while testing our method on an x86 architecture. Whenever we observe an error, we see that there is a very high chance that the threads also have interleaved, as indicated by the percentage of interleaved errors compared to the total errors. For the same execution order mode, using no optimizations results in an average interleaved error rate of 41.36%. This increases to close to 50% when using optimization flags.

For the different execution order mode, using no optimizations results in an average interleaved error rate of 96.37%, and when using optimization flags the percentage increases to over 99%. Naturally, we observe that the threads are "interleaving" more when we apply a different execution order, because all four threads will modify a different monitoring variable last. This artificially inflates the interleaving rate, but we notice consistency wise both methods do not under perform relative to one another. The percentage of errors are similar for the two modes and the percentage of interleaved errors relative to interleavings is almost as high as the same execution order mode.

We see that our method can generate interleavings with reasonable consistency, force data races consistently as well as observe that data races happen consistently. However, we also observe many runs that have interleaved with no errors. This is because not every interleaving leads to a data race.

5.2.2 Atomic operations

As the goal of this project is to verify concurrency and atomic C libraries on their intended behaviour, we test our method with the C standard library `<stdatomic.h>` header. We use the same testing methods as the tests in section 5.2.1. The test runs until 100000 interleaved runs have been generated. However, in this case, we expect the amount of errors to be zero. It is very unlikely that there is a bug with a tried and tested standard C library that has been used for over a decade at the point of writing this thesis. Therefore, we use this experiment to test if our method detects false positives. On the very odd chance that we do get a true positive, we can use our method to detect very rare concurrency bugs. The results of this experiment are presented in Table 5.9.

Test	Runs	Errors	% interleaved runs	% errors	% interleaved errors / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	141509	0	70.67	0.00	0.00	0.00	0.00
2	140794	0	71.03	0.00	0.00	0.00	0.00
3	155765	0	64.20	0.00	0.00	0.00	0.00
4	153632	0	65.09	0.00	0.00	0.00	0.00
5	157821	0	63.36	0.00	0.00	0.00	0.00
Average	149904.2	0	66.87	0.00	0.00	0.00	0.00

Table 5.9: Test on the `atomic_fetch_add` function from the C standard library `<stdatomic.h>` header.

From the results in Table 5.9, we can conclude that our method did not find any concurrency bugs in the standard atomic library. Furthermore, due to the non-determinism of the program, we can never with absolute certainty say that our program does not detect false positives. Nevertheless, it is demonstrable that the likelihood of detecting false positives is extremely low. Because of this, we run this test another ten times, with one million interleavings per run until a total of ten million interleavings are generated. All of the results gained from those experiments returned the same results as presented in Table 5.9. Adjusting the optimization levels has no impact on the number of errors generated, which remains 0 across all experiments that use atomic instructions from the standard C `atomic.h` library.

5.2.3 Atomic library bug

The final test is designed to show that our method can expose rare data race errors better than a naive concurrency test. For this purpose, we introduce a bug that is more likely to trigger when treads are executing concurrently. We modify an existing compare-and-swap (`a_cas`) function from the musl-gcc library [22], by changing the spinlock from a do-while loop to a for-loop. By doing so, we create a hard limit for the CAS function so it does not wait indefinitely for the other threads to finish before it modifies a shared variable. We then compare the test results of our method on the aforementioned library function to a naive program to demonstrate that a simple test fails to detect any errors, while our method succeeds in doing so.

The CAS function we modify is originally designed for an `aarch64` architecture, which our Raspberry Pi runs on. We found that modifying the x86 CAS (shown in Listing 5.1) function was too hard, as it either required removing a lock or introducing hard-coded bugs. This is because x86 is a CISC based architecture, meaning its instructions are more monolithic and are therefore harder to alter than multiple relatively simpler instructions.

```
static inline int a_fetch_add(volatile int *x, int v)
{
    __asm__( "lock ; xadd %0, %1" : "=r"(v), "=m"(*x) : "0"(v) : "memory" );
    return v;
}
```

Listing 5.1: x86 compare and swap function from the musl-gcc library.

In listings 5.2 and 5.3 is demonstrated what the compare and swap function `a_cas` looks like, and how the compare and swap function is altered for our test. We replace the do-while loop with a for-loop with a low range for the loop counter, ranging from 0 and 4 for a total of 5 iterations. From our testing, we saw that any limit ≥ 5 for the loop counter makes the CAS function race free. This is a result of our decision to work with a maximum of four threads because of the hardware limitations of the R-pi, as it only has one core with four threads. This means that if all four threads have the possibility to wait more than four times for other threads, all threads will be able to modify the shared variable without issue. Therefore, with a CAS iteration limit ≥ 5, our method will find no errors.

The process starts in the `a_fetch_add` function, where the old value of the shared variable is stored into `old`, whereafter it calls the `a_cas` function. The `a_cas` function then loads the current value of what `*p` points to. If this is not the same as the old value of the shared variable (`t`), the function sets up a memory barrier and returns 1, after which the while loop in the `a_fetch_add` function continues. If the value of `old` is the same as `t`, then the function can safely modify the shared variable. Finally, the CAS function returns 0 and the function is completed.

```
#define a_cas a_cas
static inline int a_cas(volatile int *p,
    int t, int s)
{
    int old;
    do {
        old = a_ll(p);
        if (old != t) {
            a_barrier();
            return 1;
        }
    } while(!a_sc(p, s));
    a_sc(p, s);
    return 0;
}

static inline int a_fetch_add(volatile
    int *x, int v)
{
    int old;
    do old = *x;
    while (a_cas(x, old, old+v));
    return old;
}
```

Listing 5.2: Original CAS

```
#define a_cas a_cas
static inline int a_cas(volatile int *p,
    int t, int s)
{
    int old;
    for (int i = 0; i < 5 && !a_sc(p, s);
         i++) {
        old = a_ll(p);
        if (old != t) {
            a_barrier();
            return 1;
        }
    }
    return 0;
}

static inline int a_fetch_add(volatile
    int *x, int v)
{
    int old;
    do old = *x;
    while (a_cas(x, old, old+v));
    return old;
}
```

Listing 5.3: New CAS

Tests with our method

In Tables 5.10, 5.11, 5.12, and 5.13 we present the results of our experiments with our method. We let our method execute the `a_fetch_add` (listings 5.2, 5.3) function until 100.000 interleavings have been generated.

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	10527033	10527033	0.98	100.00	0.98	0.98	100.00
2	6204252	5123	1.89	0.09	0.00	1.14	0.05
3	8532817	0	1.35	0.00	0.00	nan	0.00
4	6642733.2	0	2.006	0.00	0.00	nan	0.00
5	2017257	0	4.96	0.00	0.00	nan	0.00

Table 5.10: This table presents the averages of the results of our experiments, where we test the modified CAS algorithm with our method. These tests are run on the R-Pi, compiled with the `-O0` optimization flag.

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	2095691.6	2095691.6	5.028	100.00	5.028	5.028	100.00
2	2863354.6	46212.4	3.84	1.686	0.304	18.904	8.516
3	2538973.2	109.6	4.066	0.006	0.000	36.192	0.04
4	2491285.2	0.8	4.154	0.00	0.00	10.00	0.00
5	4408911.2	0	2.516	0.00	0.00	0	0.00

Table 5.11: This table presents the averages of the results of our experiments, where we test the modified CAS algorithm with our method. These tests are run on the R-Pi, compiled with the `-O1` optimization flag.

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	3528579.2	3528579.2	3.158	100.00	3.158	3.158	100.00
2	2694038	36690.6	4.546	1.588	0.296	19.528	6.792
3	3865392.8	45.2	3.372	0.002	0.000	56.684	0.024
4	2253943.6	1.6	4.572	0.00	0.00	72.22	0.00
5	3851595.8	0	3.322	0.00	0.00	nan	0.00

Table 5.12: This table presents the averages of the results of our experiments, where we test the modified CAS algorithm with our method. These tests are run on the R-Pi, compiled with the `-O2` optimization flag.

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	7026004.4	7026004.4	1.756	100.00	1.756	1.756	100.00
2	5017489	2309.2	2.312	0.058	0.002	5.018	0.076
3	6738556.4	0.6	1.406	0.00	0.00	13.33	0.00
4	6823077.2	0	1.924	0.00	0.00	nan	0.00
5	3240734.4	0	4.162	0.00	0.00	nan	0.00

Table 5.13: This table presents the averages of the results of our experiments, where we test the modified CAS algorithm with our method. These tests are run on the R-Pi, compiled with the `-O3` optimization flag..

From these results, we see a couple of interesting things:

- With only 1 CAS iteration, the errors already happen regardless of whether interleavings have been generated, therefore the percentage of interleaved runs is lower.
- When compiling with no optimizations (`-O0`), our method does not detect data race bugs when the CAS function iterates more than twice.
- When compiling with optimization flags `-O1` and `-O2`, we see that our method detects bugs quite consistently anywhere up to four CAS iterations.
- When compiling with optimization flag `-O3`, we see that our method only detects bugs until 3 CAS iterations.
- As we are limited by a maximum of 4 threads by the R-pi, any amount of CAS iterations greater than 4 results in no errors. The threads will have to wait a maximum of 4 times to modify the

shared variable. As a result, we see that for 5 CAS iterations there are no errors detected regardless of optimization level.

Naive test

The goal of the naive test is to demonstrate that our method is an improvement over a simple naive testing method. The naive test is less likely to detect data races in the aforementioned bug-induced `a_cas` function. We have implemented a naive method ourselves, because we found there is lack of state-of-the-art black-box testing methods for multithreaded programs in the current literature that follow our performance and portability requirements.

Much like our method, the naive test utilizes several threads, up to a maximum of four. Every thread runs the same for-loop for 1.000.000 iterations, where the `a_fetch_add` (listings 5.2, 5.3) function is called to add 1 to a shared variable `total_count`, which starts at 0. So, if we run this test with four threads, we expect the shared variable to be 4.000.000 if there are no concurrency bugs. We opted for a higher number of executions here, since this simple function cannot loop until X amount of interleavings or concurrent executions have been generated. We see from the experiments in the previous section that the number of runs in those experiments could easily exceed multiple millions. To make the number of executions comparable, we chose to let each thread run a million times in the naive experiments.

Result (CAS: 1)	Result (CAS: 2)	Result (CAS: 3)	Result (CAS: 4)	Result (CAS: 5)
1480468	3999994	4000000	4000000	4000000
1481467	3999997	4000000	4000000	4000000
1479773	3999995	4000000	4000000	4000000
1492221	3999998	4000000	4000000	4000000
1541700	3999998	4000000	4000000	4000000

Table 5.14: This table shows the results of 5 naive tests for our bug-induced function, 4 threads, 1.000.000 executions per thread, compiled with `-O0`. The expected value is $4 \times 1.000.000 = 4.000.000$, and the table shows the actual values gained from the tests.

Result (CAS: 1)	Result (CAS: 2)	Result (CAS: 3)	Result (CAS: 4)	Result (CAS: 5)
1999103	3999999	4000000	4000000	4000000
1998993	4000000	4000000	4000000	4000000
1996102	4000000	4000000	4000000	4000000
1995997	3999999	4000000	4000000	4000000
1995977	4000000	4000000	4000000	4000000

Table 5.15: This table shows the results of 5 naive tests for our bug-induced function, 4 threads, 1.000.000 executions per thread, compiled with `-O1`. The expected value is $4 \times 1.000.000 = 4.000.000$, and the table shows the actual values gained from the tests.

Result (CAS: 1)	Result (CAS: 2)	Result (CAS: 3)	Result (CAS: 4)	Result (CAS: 5)
1994600	3999999	4000000	4000000	4000000
1997615	4000000	4000000	4000000	4000000
1998959	3999999	4000000	4000000	4000000
1999297	3999999	4000000	4000000	4000000
1998874	4000000	4000000	4000000	4000000

Table 5.16: This table shows the results of 5 naive tests for our bug-induced function, 4 threads, 1.000.000 executions per thread, compiled with `-O2`. The expected value is $4 \times 1.000.000 = 4.000.000$, and the table shows the actual values gained from the tests.

Result (CAS: 1)	Result (CAS: 2)	Result (CAS: 3)	Result (CAS: 4)	Result (CAS: 5)
1996356	3999999	4000000	4000000	4000000
1996642	4000000	4000000	4000000	4000000
1996497	4000000	4000000	4000000	4000000
1996325	3999999	4000000	4000000	4000000
1996536	3999999	4000000	4000000	4000000

Table 5.17: This table shows the results of 5 naive tests for our bug-induced function, 4 threads, 1.000.000 executions per thread, compiled with `-O3`. The expected value is $4 \times 1.000.000 = 4.000.000$, and the table shows the actual values gained from the tests.

These results show that the naive tests only result in wrong outcomes when the for-loop limit is ≤ 2. We see that for 1 CAS iteration even the naive function detects data race bugs 100% of the time. For 2 CAS iterations, it already becomes much harder for the naive function to detect any bugs, sometimes even resulting in the expected value of four million. For more than 2 CAS iterations, the naive algorithm does not show any bugs, resulting in the expected value 100% of the time.

Compared to the naive tests, our method performs significantly better. Regardless of optimization flags, our method picks up bugs 100% of the time with 2 CAS iterations. The naive tests are only bug-free with no optimizations and 2 CAS iterations. Furthermore, our method can detect bugs in the modified CAS algorithm for a maximum of 4 iterations.

Chapter 6

Conclusion

6.1 Summary and findings

Testing multithreaded programs and primitives is hard due to their non-deterministic nature. With the OS scheduler deciding the order of execution of the instructions in different threads, many interleavings are feasible and, often, cannot be tested effectively with regular procedures. Therefore, solely relying on traditional tests is not suited for multithreaded testing. In previous work, various attempts have been made to reduce non-determinism by discovering and testing rare interleavings that cannot be recreated under normal circumstances. Unfortunately, such methods are still not guaranteed to be able to recreate all interleavings. Attempting to do so will quickly lead to a state-explosion problem. Instead, in our work, we focus on testing instructions such as multithreaded primitives in C on their susceptibility to data race bugs when operating in a concurrent environment. To this end, we presented a novel way to generate many thread interleavings with a focus on runs where we detected thread interference. Our methodology and evaluation indicate our method is able to identify multithreading vulnerabilities. We therefore proceed to summarize the answers we have found to our research questions.

RQ1: On a source code level, how can we monitor if threads are interleaving?

To monitor if threads are interleaving, we use lightweight instructions that track the execution order of threads by their respective thread ID. We found that using shared variables reflecting thread execution order for monitoring was most suited. Upon encountering differences in thread execution order for a specific shared variable, we can derive that threads are operating concurrently. We then further analyzed these cases and found that, in these cases, there was a higher probability of data-race errors compared to runs where no thread interference was found.

RQ2: How can we systematically test C library functions on their vulnerability to data races in a multithreaded environment?

To test C instructions on their vulnerabilities to data races, we designed a method to force instructions to run concurrently for multiple threads. We did so by starting the executions of each thread simultaneously, letting each thread execute, analyzing the outcome of the run, and synchronizing every thread again. This whole process is done for every single iteration of the tests with good performance. The method can generate many interleaved runs quickly and with high consistency. Even though we cannot systematically go through every possible interleaving, we can go through sufficiently many very quickly. Although our method does not guarantee that some instructions are bug-free when the tests result in no errors, the likelihood increases that they are.

RQ3: How can our method be used to detect multithreading vulnerabilities in a C library?

We validated our method by purposefully introducing a bug into an existing atomic library, and comparing the results of tests using our method to results from our naive tests. We found that, to a certain extent, the naive tests were able to detect some concurrency bugs. However, in the cases where the naive tests were not able to detect bugs, our method found errors on multiple occasions. Our method was able to detect bugs until it there was no possibility for data races to happen anymore.

6.2 Contributions

In this work, we made the following contributions:

- We developed a portable method to quickly generate many *different* interleavings of multiple threads running concurrently.
- We demonstrated how our method runs on different systems, with different architectures and different compilers.
- We showed that different optimization levels have a large impact on the test results.
- We demonstrated that our method is an improvement over a naive test, as it can detect additional bugs.
- We showed how our method can be used to increase the confidence that tested multithreading primitives are race-free.

6.3 Limitations and threats to validity

Our method is designed to test compilers and libraries, such as `stdatomic.h` in C, on their correctness in multithreaded programs. However, for the monitoring of the method, we utilize some functions from the same `stdatomic` library - like `atomic_fetch_add`. Thus, for our work, we have to assume that at least some functions do behave properly, otherwise it would be impossible to design this method. Therefore, this method's correctness is based on the assumption that `atomic_fetch_add` from the `stdatomic` library and `pthread_cond_wait` from the POSIX thread library are fully correct.

Another limitation - which appeared due to time and hardware constraints - was the hardware we could test on. We tested our method on an AMD and ARM CPU, but we had no opportunities to test our method on an Intel chip. Currently, Intel still has the largest market share of CPUs over other companies such as AMD [23], therefore testing on that hardware would have been interesting. Furthermore, we have only quickly tested whether our method *works* for multiple compilers. Our most comprehensive analysis focused on `gcc`. It is possible that other compilers could show different results in more complex tests. To ensure our findings hold for most compilers, additional testing needs to be done with those other compilers.

Even though we found there is a correlation between the interleavings and data races, indicated by the high percentage of interleavings with errors, we have no direct proof that there is a causal link between the two. Deeper analysis indicated to us that when threads are interleaving in our method, the likelihood of data race errors increases. However, we have not been able to prove that this is the case. All of our data is empirical.

Finally, our method can only increase the level of confidence that certain functions - such as multithreading primitives -are data-race bug-free. Because our method is still intrinsically non-deterministic, some bugs may never be discovered. As such, our method is by design as limited as other randomly generated tests.

6.4 Future work

For future work, an interesting research direction is to look deeper into the optimization levels. From our results, we see that optimizations have a large impact on the outcome of our tests. For now, we can only speculate why that might be, and further insight can be gained by analyzing and comparing the generated bytecode for every optimization level. Such research can reveal if certain optimizations make it easier for threads to start interleaving, or more prone to data race bugs.

Furthermore, our work can be extended beyond instructions: it can allow for larger chunks of code to be tested. If the method can somehow be improved to work for programs, it can be used to increase confidence in the correctness of such programs.

Currently, our method is able to detect data vulnerabilities, but not determine whether they are caused by a faulty compiler or library. Future work should also explore a method to extend this work by determining the root cause of a bug. This would greatly improve the use case of our method, as it will speed up the whole testing process significantly.

If our method can be successfully implemented into SuperTest in the future, other compilers and libraries could be tested better on their correctness in multithreaded programs.

Bibliography

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[2] P. Kopta *et al.*, "Parallel application benchmarks and performance evaluation of the intel xeon 7500 family processors," *Procedia Computer Science*, vol. 4, pp. 372–381, 2011.

[3] N. Rudin, D. Hoeller, P. Reist, and M. Hutter, "Learning to walk in minutes using massively parallel deep reinforcement learning," in *Conference on Robot Learning*, PMLR, 2022, pp. 91–100.

[4] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014.

[5] *C11 standard draft*, `https://www.open-std.org/JTC1/SC22/WG14/www/projects#9899`.

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

[7] J. Hillebrand, P. Reichenpfader, I. Mandic, H. Siegl, and C. Peer, "Establishing confidence in the usage of software tools in context of iso 26262," in *Computer Safety, Reliability, and Security: 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings 30*, Springer, 2011, pp. 257–269.

[8] Y. Li, S. Ding, Q. Zhang, and D. Italiano, "Debug information validation for optimized code," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1052–1065.

[9] D. Alam, M. Zaman, T. Farah, R. Rahman, and M. S. Hosain, "Study of the dirty copy on write, a linux kernel memory allocation vulnerability," in *2017 International Conference on Consumer Electronics and Devices (ICCED)*, IEEE, 2017, pp. 40–45.

[10] W. M. Waite and G. Goos, *Compiler construction*. Springer Science & Business Media, 2012.

[11] *Options That Control Optimization*, `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[12] D. Dice, D. Hendler, and I. Mirsky, "Lightweight contention management for efficient compare-and-swap operations," in *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings 19*, Springer, 2013, pp. 595–606.

[13] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the aba problem in descriptor-based lock-free designs," in *2010 13th IEEE international symposium on object/component/service-oriented real-time distributed computing*, IEEE, 2010, pp. 185–192.

[14] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 485–502.

[15] R. Watertor, "Assessing the standard-compliance for multi-threading primitives in c compilers,"

[16] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: Efficient deterministic multithreading," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 327–336.

[17] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, "Ad hoc synchronization considered harmful," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[18] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, 2009, pp. 25–36.

[19] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71.

[20] 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*, 2018, pp. 15–25.

[21] *`Atomic_fetch_add` documentation as per `cppreference.com`*, `https://en.cppreference.com/w/c/atomic/atomic_fetch_add`.

[22] *"musl is an implementation of the C standard library built on top of the Linux system call API"'*, `https://musl.libc.org/`.

[23] V. Tiwari, "Amd's comeback in the cpu market share with the launch of ryzen," *Int. J. Res. Circuits Devices Syst*, vol. 3, pp. 17–23, 2022.

Appendix A

Non-crucial information

test	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	6761639	802	1.48	0.01	0.01	99.88	0.80
2	3560548	1995	2.81	0.06	0.06	99.70	1.99
3	4131195	1430	2.42	0.03	0.03	99.79	1.43
4	10996097	568	0.91	0.01	0.01	98.77	0.56
5	5905664	1043	1.69	0.02	0.02	99.71	1.04
Average	8093308.6	1677.6	1.842	0.026	0.026	99.768	11.644

Table A.1: Non-atomic, different instruction order, 4 threads, 100000 interleavings, -O0:

test	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	3239679	10321	3.09	0.32	0.32	99.88	10.31
2	608076	9813	16.45	1.61	1.61	99.99	9.81
3	717983	5002	13.93	0.70	0.70	99.76	4.99
4	966623	2232	10.35	0.23	0.22	97.27	2.17
5	7670369	9759	1.30	0.13	0.13	99.89	9.75
Average	4551466	5405.4	9.424	0.598	0.596	99.758	7.206

Table A.2: Non-atomic, different instruction order, 4 threads, 100000 interleavings, -O1:

test	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	2412296	15417	4.15	0.64	0.64	99.57	15.35
2	674744	12504	14.82	1.85	1.85	99.74	12.47
3	827399	10257	12.09	1.24	1.24	99.98	10.26
4	1075196	12104	9.30	1.13	1.11	99.03	11.99
5	3700462	13905	2.70	0.38	0.37	99.15	13.79
Average	2740219.4	10837.4	8.612	0.808	0.802	99.494	12.572

Table A.3: Non-atomic, different instruction order, 4 threads, 100000 interleavings, -O2:

test	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	4722804	678	2.12	0.01	0.01	100.00	0.68
2	17687723	281	0.57	0.00	0.00	99.64	0.28
3	5814491	594	1.72	0.01	0.01	99.83	0.59
4	4146495	318	2.41	0.01	0.01	100.00	0.32
5	8541333	277	1.17	0.00	0.00	99.64	0.28
Average	7172589.2	429.6	1.798	0.006	0.006	99.822	0.49

Table A.4: Non-atomic, different instruction order, 4 threads, 100000 interleavings, -O3:

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	8898554	8898554	1.12	100.00	1.12	1.12	100.00
1	9387701	9387701	1.07	100.00	1.07	1.07	100.00
1	9966879	9966879	1.00	100.00	1.00	1.00	100.00
1	16457301	16457301	0.61	100.00	0.61	0.61	100.00
1	9479630	9479630	1.05	100.00	1.05	1.05	100.00
Average	10527033	10527033	0.98	100.00	0.98	0.98	100.00

Table A.5: Buggy library test, 4 threads, 100000 interleavings, -O0, CAS iterations: 1

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
2	3406357	2578	2.94	0.08	0.00	1.67	0.04
2	4317427	3639	2.32	0.08	0.00	1.37	0.05
2	11435123	4312	0.87	0.04	0.00	0.63	0.03
2	5839738	5792	1.71	0.10	0.00	1.38	0.08
2	6188359	9296	1.62	0.15	0.00	0.68	0.06
Average	6204252	5123	1.89	0.09	0.00	1.14	0.05

Table A.6: Buggy library test, 4 threads, 100000 interleavings, -O0, CAS iterations: 2

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
3	5066011	0	1.97	0.00	0.00	nan	0.00
3	11605143	0	0.86	0.00	0.00	nan	0.00
3	5318150	0	1.88	0.00	0.00	nan	0.00
3	11306673	0	0.88	0.00	0.00	nan	0.00
3	8534108	0	1.17	0.00	0.00	nan	0.00
Average	8532817	0	1.35	0.00	0.00	nan	0.00

Table A.7: Buggy library test, 4 threads, 100000 interleavings, -O0, CAS iterations: 3

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
4	3592540	0	2.78	0.00	0.00	nan	0.00
4	4622438	0	2.16	0.00	0.00	nan	0.00
4	5273337	0	1.90	0.00	0.00	nan	0.00
4	10507478	0	0.95	0.00	0.00	nan	0.00
4	8096013	0	1.24	0.00	0.00	nan	0.00
Average	6642733.2	0	2.006	0.00	0.00	nan	0.00

Table A.8: Buggy library test, 4 threads, 100000 interleavings, -O0, CAS iterations: 4

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
5	17009687	0	0.59	0.00	0.00	nan	0.00
5	9681541	0	1.03	0.00	0.00	nan	0.00
5	3830843	0	2.61	0.00	0.00	nan	0.00
5	9303190	0	1.07	0.00	0.00	nan	0.00
5	3451271	0	2.90	0.00	0.00	nan	0.00
Average	7932886.4	0	1.84	0.00	0.00	nan	0.00

Table A.9: Buggy library test, 4 threads, 100000 interleavings, -O0, CAS iterations: 5

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	2161507	2161507	4.63	100.00	4.63	4.63	100.00
1	1530030	1530030	6.54	100.00	6.54	6.54	100.00
1	3774814	3774814	2.65	100.00	2.65	2.65	100.00
1	1352458	1352458	7.39	100.00	7.39	7.39	100.00
1	1659649	1659649	6.03	100.00	6.03	6.03	100.00
Average	2095691.6	2095691.6	5.028	100.00	5.028	5.028	100.00

Table A.10: Buggy library test, 4 threads, 100000 interleavings, -O1, CAS iterations: 1

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
2	3401174	57767	2.94	1.70	0.24	14.09	8.14
2	2104020	43532	4.75	2.07	0.38	18.56	8.08
2	2313728	49244	4.32	2.13	0.34	16.09	7.92
2	3573735	33581	2.80	0.94	0.24	25.83	8.68
2	2954116	46938	3.39	1.59	0.32	19.95	9.37
Average	2863354.6	46212.4	3.84	1.686	0.304	18.904	8.516

Table A.11: Buggy library test, 4 threads, 100000 interleavings, -O1, CAS iterations: 2

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
3	1426185	132	7.01	0.01	0.00	37.12	0.05
3	2727836	58	3.67	0.00	0.00	43.10	0.03
3	1451038	83	6.89	0.01	0.00	18.07	0.02
3	3091528	229	3.23	0.01	0.00	34.50	0.08
3	3958319	46	2.53	0.00	0.00	52.17	0.02
Average	2538973.2	109.6	4.066	0.006	0.000	36.192	0.04

Table A.12: Buggy library test, 4 threads, 100000 interleavings, -O1, CAS iterations: 3

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
4	1508265	1	6.63	0.00	0.00	0.00	0.00
4	4329303	1	2.31	0.00	0.00	0.00	0.00
4	2377213	2	4.21	0.00	0.00	50.00	0.00
4	3052503	0	3.28	0.00	0.00	nan	0.00
4	1199142	0	8.34	0.00	0.00	nan	0.00
Average	2491285.2	0.8	4.154	0.00	0.00	nan	0.00

Table A.13: Buggy library test, 4 threads, 100000 interleavings, -O1, CAS iterations: 4

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
5	4519869	0	2.21	0.00	0.00	nan	0.00
5	4855842	0	2.06	0.00	0.00	nan	0.00
5	5214971	0	1.92	0.00	0.00	nan	0.00
5	3899950	0	2.56	0.00	0.00	nan	0.00
5	3533924	0	2.83	0.00	0.00	nan	0.00
Average	4408911.2	0	2.516	0.00	0.00	nan	0.00

Table A.14: Buggy library test, 4 threads, 100000 interleavings, -O1, CAS iterations: 5

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	5788865	5788865	1.73	100.00	1.73	1.73	100.00
1	2214558	2214558	4.52	100.00	4.52	4.52	100.00
1	3121251	3121251	3.20	100.00	3.20	3.20	100.00
1	2713701	2713701	3.69	100.00	3.69	3.69	100.00
1	3774471	3774471	2.65	100.00	2.65	2.65	100.00
Average	3528579.2	3528579.2	3.158	100.00	3.158	3.158	100.00

Table A.15: Buggy library test, 4 threads, 100000 interleavings, -O2, CAS iterations: 1

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
2	3228912	42207	3.10	1.31	0.19	14.22	6.00
2	4065505	34749	2.46	0.85	0.16	18.34	6.37
2	1587400	29398	6.30	1.85	0.48	25.97	7.64
2	3078905	34857	3.25	1.13	0.27	23.43	8.17
2	1509468	42242	6.62	2.80	0.38	13.69	5.78
Average	2694038	36690.6	4.546	1.588	0.296	19.528	6.792

Table A.16: Buggy library test, 4 threads, 100000 interleavings, -O2, CAS iterations: 2

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
3	4585472	54	2.18	0.00	0.00	77.78	0.04
3	2689986	36	3.72	0.00	0.00	44.44	0.02
3	3403270	20	2.94	0.00	0.00	65.00	0.01
3	7488686	21	1.34	0.00	0.00	66.67	0.01
3	1139550	95	8.78	0.01	0.00	30.53	0.03
Average	3865392.8	45.2	3.372	0.002	0.000	56.684	0.024

Table A.17: Buggy library test, 4 threads, 100000 interleavings, -O2, CAS iterations: 3

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
4	3049244	2	3.28	0.00	0.00	50.00	0.00
4	2020459	3	4.95	0.00	0.00	100.00	0.00
4	1340945	3	7.46	0.00	0.00	66.67	0.00
4	1658564	0	6.03	0.00	0.00	nan	0.00
4	3180506	0	3.14	0.00	0.00	nan	0.00
Average	2253943.6	1.6	4.572	0.00	0.00	nan	0.00

Table A.18: Buggy library test, 4 threads, 100000 interleavings, -O2, CAS iterations: 4

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
5	5863797	0	1.71	0.00	0.00	nan	0.00
5	4874853	0	2.05	0.00	0.00	nan	0.00
5	5329581	0	1.88	0.00	0.00	nan	0.00
5	1886025	0	5.30	0.00	0.00	nan	0.00
5	1303723	0	7.67	0.00	0.00	nan	0.00
Average	3851595.8	0	3.322	0.00	0.00	nan	0.00

Table A.19: Buggy library test, 4 threads, 100000 interleavings, -O2, CAS iterations: 5

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
1	13144203	13144203	0.76	100.00	0.76	0.76	100.00
1	5832336	5832336	1.71	100.00	1.71	1.71	100.00
1	2491897	2491897	4.01	100.00	4.01	4.01	100.00
1	8269220	8269220	1.21	100.00	1.21	1.21	100.00
1	4766466	4766466	2.10	100.00	2.10	2.10	100.00
Average	7026004.4	7026004.4	1.756	100.00	1.756	1.756	100.00

Table A.20: Buggy library test, 4 threads, 100000 interleavings, -O3, CAS iterations: 1

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
2	5623319	662	1.78	0.01	0.00	13.44	0.09
2	8211166	4044	1.22	0.05	0.00	1.36	0.05
2	1954583	2703	5.12	0.14	0.01	5.18	0.14
2	5471098	817	1.83	0.01	0.00	4.53	0.04
2	3827279	3420	2.61	0.09	0.00	1.78	0.06
Average	5017489	2309.2	2.312	0.058	0.002	5.018	0.076

Table A.21: Buggy library test, 4 threads, 100000 interleavings, -O3, CAS iterations: 2

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
3	10199099	1	0.98	0.00	0.00	0.00	0.00
3	10520733	0	0.95	0.00	0.00	nan	0.00
3	5500089	3	1.82	0.00	0.00	66.67	0.00
3	8560666	0	1.17	0.00	0.00	nan	0.00
3	8973745	1	1.11	0.00	0.00	0.00	0.00
Average	6738556.4	0.6	1.406	0.00	0.00	nan	0.00

Table A.22: Buggy library test, 4 threads, 100000 interleavings, -O3, CAS iterations: 3

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
4	3906308	0	2.56	0.00	0.00	nan	0.00
4	17500574	0	0.57	0.00	0.00	nan	0.00
4	4080756	0	2.45	0.00	0.00	nan	0.00
4	4241057	0	2.36	0.00	0.00	nan	0.00
4	14403691	0	0.69	0.00	0.00	nan	0.00
Average	6823077.2	0	1.924	0.00	0.00	nan	0.00

Table A.23: Buggy library test, 4 threads, 100000 interleavings, -O3, CAS iterations: 4

CAS iterations	Runs	Errors	% interleaved runs	% errors	% interleaved runs / total runs	% interleaved errors / total errors	% interleaved errors / total interleavings
5	1887629	0	5.30	0.00	0.00	nan	0.00
5	2435425	0	4.11	0.00	0.00	nan	0.00
5	1931782	0	5.18	0.00	0.00	nan	0.00
5	7951579	0	1.26	0.00	0.00	nan	0.00
5	2017257	0	4.96	0.00	0.00	nan	0.00
Average	3240734.4	0	4.162	0.00	0.00	nan	0.00

Table A.24: Buggy library test, 4 threads, 100000 interleavings, -O3, CAS iterations: 5

Actual (CAS: 1)	Actual (CAS: 2)	Actual (CAS: 3)	Actual (CAS: 4)	Actual (CAS: 5)
1480468	3999994	4000000	4000000	4000000
1481467	3999997	4000000	4000000	4000000
1479773	3999995	4000000	4000000	4000000
1492221	3999998	4000000	4000000	4000000
1541700	3999998	4000000	4000000	4000000

Table A.25: Naive buggy library test, 4 threads, 100000 interleavings, -O0, Expected value - 4.000.000

Actual (CAS: 1)	Actual (CAS: 2)	Actual (CAS: 3)	Actual (CAS: 4)	Actual (CAS: 5)
1999103	3999999	4000000	4000000	4000000
1998993	4000000	4000000	4000000	4000000
1996102	4000000	4000000	4000000	4000000
1995997	3999999	4000000	4000000	4000000
1995977	4000000	4000000	4000000	4000000

Table A.26: Naive buggy library test, 4 threads, 100000 interleavings, -O1, Expected value - 4.000.000

Actual (CAS: 1)	Actual (CAS: 2)	Actual (CAS: 3)	Actual (CAS: 4)	Actual (CAS: 5)
1994600	3999999	4000000	4000000	4000000
1997615	4000000	4000000	4000000	4000000
1998959	3999999	4000000	4000000	4000000
1999297	3999999	4000000	4000000	4000000
1998874	4000000	4000000	4000000	4000000

Table A.27: Naive buggy library test, 4 threads, 100000 interleavings, -O2, Expected value - 4.000.000

Actual (CAS: 1)	Actual (CAS: 2)	Actual (CAS: 3)	Actual (CAS: 4)	Actual (CAS: 5)
1996356	3999999	4000000	4000000	4000000
1996642	4000000	4000000	4000000	4000000
1996497	4000000	4000000	4000000	4000000
1996325	3999999	4000000	4000000	4000000
1996536	3999999	4000000	4000000	4000000

Table A.28: Naive buggy library test, 4 threads, 100000 interleavings, -O3, Expected value - 4.000.000