UPPSALA
UNIVERSITET

UPTEC IT 22031
Examensarbete 30 hp
November 2022

Compiler Testing of C11 Atomics for Arm and RISC-V

Hampus Adolfsson

Civilingenjörsprogrammet i informationsteknologi

UPPSALA
UNIVERSITET

Compiler Testing of C11 Atomics for Arm and RISC-V

Hampus Adolfsson

Abstract

The C11 standard introduced atomic types and operations, with an accompanying memory model, to enable the use of shared variables in concurrent programs. In this thesis, I demonstrate how compilers can be tested, in a way that is deterministic and covers the entire set of atomic operations, to ensure they correctly implement C11 atomics and the C11 memory model.

I use a large set of short concurrent programs (”litmus tests”), generated from a model written in a specification language and based on a formalized C11 memory model. Each test program is compiled and run with a model checker, to determine the possible outcomes; any program with an outcome that is possible after compilation but not allowed by C11 is a failed test case. As an alternative to model checking, I also test a nondeterministic, hardware-based method for running tests, but I find that this method is too inaccurate to be useful.

I test IAR and gcc compilers for Arm and RISC-V; all of these compilers pass all tests. Out of three compilers with purposefully inserted bugs, all are correctly identified as faulty. This testing process thus shows some promise, but further evaluation is needed.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala/Visby

Handledare: Björn Bergman Ämnesgranskare: Bengt Jonsson

Examinator: Lars-Åke Nordén

Popular scientific summary in Swedish

Den komponent i en dator som exekverar kod kallas för en *processor*. Prestandan hos processorn, alltså hur snabbt den kan exekvera kod, är en av de viktigaste delarna för datorns prestanda överlag. Traditionellt sett har man ökat processorers prestanda genom att ge dem fler och mindre transistorer. Detta lönar sig dock inte längre lika mycket; istället tillverkar man processorer med flera *kärnor*, där alla kärnor exekverar kod samtidigt, delvis oberoende av varandra. Kod som körs på de olika kärnorna kan kommunicera med varandra genom att skriva och läsa data från ett gemensamt minne, men reglerna för hur sådan kommunikation fungerar är ofta mycket komplicerade.

För att ett program ska kunna dra nytta av de flera kärnorna i en processor krävs att programmet har flera kodavsnitt som kan köras på varsin processorkärna, och som samarbetar för att uppnå sitt mål. I praktiken skrivs sådana program (precis som andra datorprogram) ofta i ett programmeringsspråk, som är enkelt att formulera algoritmer i men som processorn inte förstår sig på. Programmet körs sedan genom en *kompilator*, som konverterar programmet till maskinkod.

I den här avhandligen behandlar jag kompilatorer för programmeringsspråket C. I C finns s.k. atomiska operationer som ska användas för att skriva och läsa gemensamt minne i program som körs på flera kärnor. Dessa atomiska operationer följer en uppsättning regler som gäller oberoende av vilken processor programmet ska köras på. När en kompilator konverterar ett C-program till maskinkod måste den beakta de regler som gäller för den specifika processorn och dess minne, och generera maskinkod som inte bryter mot C:s processoroberoende regler för atomiska operationer. Mitt mål med denna anvhandling är att beskriva hur man kan testa en kompilator i syfte att verifiera att den alltid genererar maskinkod som uppfyller C:s regler för atomiska operationer.

Det mest centrala i min testprocedur är en samling testprogram. Dessa är korta C-program som ska kunna köras på flerkärninga processorer och använder atomiska operationer för att kommunicera mellan kärnorna. Om man vet vilka resultat ett sådant program får ge utan att bryta mot C:s regler för atomiska operationer kan det användas för att testa att kompilatorn är korrekt.

Testprogrammen behöver inte skrivas av en person utan genereras automatiskt. Jag använder ett verktyg som utifrån en beskrivning av hur ett testprogram ser ut och ett antal regler för vad som gör ett testprogram intressant, kan generera alla intressanta testprogram upp till någon storleksgräns. För varje testfall räknas också ut vilka utfall som går att få när programmet körs, alltså de utfall som tillåts av C.

En kompilators korrekthet testas genom att för varje testprogram först kompilera det, och sedan undersöka vilka utfall som går att nå genom att köra det kompilerade programmet. De nåbara utfallen jämförs med de tillåtna utfallen som räknats ut när testet genererades; om det kompilerade programmet kan nå något utfall som inte är bland de tillåtna utfallen har kompilatorn gjort en felaktig kompilering av programmet. Att undersöka vilka utfall som ett kompilerat program kan nå är dock inte helt trivialt och jag jämför därför två olika

metoder för att göra det, en matematisk (algoritmisk) metod och en som kör programmet på hårdvara.

Jag testar först olika populära kompilatorer. Alla kompilatorer klarar alla testfall, men det är svårt att dra några slutsatser från detta eftersom vi inte vet om dessa kompilatorer är korrekta eller inte. Därför testar jag också tre kompilatorer i vilka jag medvetet har stoppat in buggar. Med min testprocess hittar jag alla tre av dessa fel.

Jag jämför också de två olika metoderna för att undersöka kompilerade program. Den matematiska metoden är deterministisk och i de flesta fall snabbare än den hårdvarubaserade metoden. Den hårdvarubaserade metoden är enklare att applicera på olika processorer men är alldeles för opålitlig för att vara användbar.

Contents

1 Introduction

In recent decades, multicore processors have grown increasingly prevalent in almost all areas of computing. This has come with an increased need to write concurrent programs, and a demand for programming languages with support for such programs. The C11 standard for the C programming language addressed this demand by introducing, among other things, *atomic data types*, which can be accessed concurrently by multiple threads or cores using *atomic operations*. The semantics of atomic operations is described by the C11 memory model, and it is up to compilers to implement the memory model for their target architecture. Verifying that they do so correctly is difficult, both because of the complexity of the memory model, and because of the inherent nondeterministism of concurrent programs.

In this thesis I present and describe a process for automated testing of a compiler's implementation of the C11 memory model, focusing on IAR and gcc compilers for the Arm and RISC-V architectures. I exclusively test low-level atomic operations (atomic reads, writes, read-modify-writes and fences), and thus do not test locks, thread creation or any of the other high-level concurrency features introduced with C11.

First, I present a method for generating a comprehensive suite of compiler tests targeting C11 atomics. An axiomatic C11 model by Lahav et al. [LVK+17] — rewritten in the Alloy modelling language by Lustig et al. [LWPG17] — is used as the source-of-truth on what behavior C11 allows and forbids. I use this model to generate short concurrent C programs that may exhibit some interesting forbidden behavior (such tests are often called *litmus tests*). It is up to the compiler to forbid this behavior; if the behavior is possible after compilation with some compiler, then that compiler has failed the test.

Second, I present two competing methods for exploring the behavior of generated test cases after compilation. Since tests are concurrent and thus run nondeterministically, it is nontrivial to determine whether some behavior is possible. The first method uses `HERD` [AMT14], a model checker and simulator. `HERD` is able to find *all* possible executions of a test, but can be slow for some tests and suffers from limited instruction support. The second method, *stress testing*, runs tests directly on hardware many times over, with the expectation that all or most possible behavior will eventually be exhibited. This method is less accurate, but is easier to apply to new architectures and does not suffer from `HERD`'s poor instruction support.

I evaluate my testing process in three ways. First, I apply the process to try to find bugs in gcc and IAR compilers, some of which are modified to insert bugs (some from the literature and some made up). Testing deliberately buggy compilers lets me make sure that the process is able to find realistic compiler bugs. The non-modified compilers tested all pass every test. The modified (i.e. deliberately buggy) compilers all fail some tests, and are thus correctly identified as faulty. This shows that the process can be used to find bugs, but it would have to be tested with more known or deliberately buggy compilers before one could estimate how accurate it is.

Second, I examine whether the test generation method is able to find classic litmus tests defined by Alglave et al. [Lit]; these tests represent common programming patterns [AMT14] and are present in many litmus test suites. I find that my test generation method finds at least one variant of each of the classic tests, with the exception of a few tests that are not applicable to C11.

Third, I assess the accuracy of the stress testing test running method, by examining how much of the behavior allowed by C11 is visible when running tests on hardware. I find that stress testing is only able to find a third of all outcomes found by model-checking with HERD. Moreover, when used to test two of the deliberately buggy compilers mentioned above, stress testing fails to find faults in either compiler. I conclude that my stress testing implementation is too inaccurate to be useful.

Section 2 below introduces some background on compilation, concurrency and memory models. Section 3 then lays out what is and is not the objective of this thesis, and argues for why this work is important.

Then follows Section 4 on the C11 memory model and on memory model theory, which is aimed at helping the reader fully understand the rest of the thesis.

An overview and discussion is given in Section 5 of the process and methods I use to test compilers, and of how I evaluate those methods. Details of how I implement the process to generate tests, and run tests on compilers, are then given in Sections 6 and 7 respectively.

Evaluation results are presented and discussed in Section 8, after which the conclusions of the thesis are summarized in Section 9. Finally, previous work relevant to this thesis is discussed in Section 10.

2 Background

This section introduces and provides context around some concepts that are important to this thesis.

2.1 Glossary

Term	Explanation
Embedded System	A computer system that is embedded in some larger system, such as a modern car or an industrial robot. Typically these are small, power-efficient and low-cost boards.
Instruction Set Architecture (ISA)	An abstract specification of some processor, sometimes referred to as just *architecture*. Examples: x86, ARMv7-A, MIPS
Atomic operation	An operation that is completed (as if) in a single step, with no visible intermediate state. The operation is either fully executed, or not executed at all.

2.2 IAR Systems

IAR Systems is a Swedish company supplying tools and services for the development of embedded systems. These tools include an IDE, a debugger, compilers and more. IAR compilers are available for a range of architectures, and this thesis uses those for the ARMv7 and RISC-V architectures. IAR compilers generate bare-metal programs that run without any operating system, and so a few other IAR tools are also used to help flash programs onto hardware.

2.3 Concurrent programs

A *concurrent program* refers to any program that has several active components or tasks that are executed or progress independently [AdBO10]. Conversely, a *sequential* program executes all its tasks in sequence, without any interleaving or overlapping. Without proper synchronization, the outcome of a concurrent program is often non-deterministic, meaning it varies depending on the order in which certain concurrent events happen or tasks are completed.

2.3.1 Parallelism

Concurrency is often conflated with *parallelism*, but they are distinct concepts. I will use the term parallel to refer to a program executing on multiple CPU cores at the same time (other forms of parallelism exist, but are not relevant here). In this sense, parallelism is a means of achieving concurrency, but a concurrent program is not necessarily parallel.

For some time, multi-core systems have grown more and more prevalent. With the decline of Dennard scaling [EBA+11] (which roughly implies that a processor's power use is proportional to its area), it has become less feasible to reduce a processor's power draw or increase its frequency simply by using smaller

transistors. In response, many computer architects have turned to scaling the number of cores in a processor, or the number of processors on a chip. Two examples of this from the embedded world are the Cortex-A9 with up to 4 cores [Armc], and the Arm big.LITTLE architecture [Armb] which puts a low-performance and power-efficient CPU on the same chip as a high-performance and power-hungry CPU.

This trend creates a need for programs to be parallelized in order to take advantage of the capabilities of the hardware. Thus, there needs to be programming languages with support for parallel algorithms, as well as compilers that can compile programs for multi-core hardware.

2.3.2 Interrupts

Interrupts are another common source of concurrency [WCM+16,SCGC19], and are perhaps even more common in the context of embedded systems. Interrupts "interrupt" the processor at the request of some external device, and cause an interleaving of regular code and interrupt-handling code.

2.4 Memory consistency models

A concurrent program will often have some shared memory that is used by two or more threads of execution, e.g. for communication [Han77]. How this shared memory behaves is governed by the memory consistency model — hereafter referred to as just *memory model* — of the underlying system. Hill et al. define a memory model as "a specification of the allowed behavior of multi-threaded programs executing with shared memory" [HWS11]. Informally, a memory model describes the order in which stores to shared memory may happen and become visible to the different threads of execution.

Thread 0	**Thread 1**
Initially, x0 = x1 = 0	

`x1 = 1`
`if (x0 == 0)`
`  print("T0 won")`
	`x0 = 1`
	`if (x1 == 0)`
	`  print("T1 won")`

Listing 1: Pseudo-code for a program requiring sequential consistency

Probably the simplest possible memory model is sequential consistency (SC), first described by Lamport [Lam79]. Under SC, a concurrent program executes as if the memory accesses from all threads were performed one-by-one in some total order (i.e., in an interleaving fashion), and the effect of each access was immediately visible to all threads. This is a fairly intuitive model. Consider the example program in Listing 1. Under SC, it is obvious that both threads can not "win", since each thread must block the other thread before it can win itself. SC

is considered a *strong* memory model, since it makes strong guarantees about the behavior of memory.

However, there are weaker memory models, that are not as strict as SC, and where memory behavior is not as intuitive. For example, total store order (TSO) mostly behaves like SC, but allows stores to not be immediately visible to other threads [HWS11]. If we run the program in Listing 1 under TSO, it would — perhaps counter-intuitively — be possible for both threads to win, since the store to x0 may be invisible to thread 0 when it reads x0, and the store to x1 may, in the same execution, be invisible to thread 1 when it reads x1.

To avoid such problems, most memory models allow programs to insert *memory fences* (sometimes called barriers) to strengthen the guarantees made by the memory model for some part of the program. A memory fence prevents memory operations from being reordered across the fence, and some fences can force all previous stores to become visible before the thread continues. Listing 2 shows how the previous example can be amended to work under TSO, by inserting memory fences that ensure the stores are visible to the other thread before continuing.

Thread 0	**Thread 1**
Initially, x0 = x1 = 0	
`x1 = 1`	`x0 = 1`
`mfence`	`mfence`
`if (x0 == 0)`	`if (x1 == 0)`
`  print("T0 won")`	`  print("T1 won")`

Listing 2: The example from Listing 1, but fixed for TSO

2.4.1 Hardware and software memory models

In many contexts, the term "memory model" alone is used to refer the memory model of the computer architecture, which is specified by the ISA and implemented by the hardware. This model determines how, for example, the effects of store instructions become visible to other cores in the CPU. Most real hardware does not implement SC; to improve performance or reduce hardware complexity computer architects weaken the memory model. This can allow the use of — for example — store buffers or out-of-order execution [HWS11]. As an example, Arm has a relatively weak memory model [HWS11,MSS12], but this allows Arm CPUs to implement lots of hardware optimizations [MSS12].

Similarly, a programming language may have a memory model that defines how accesses to shared memory behave in that language. The memory model of the programming language allows the semantics of concurrent programs to be clearly defined and to be the same regardless of what hardware the program is run on [HWS11]. Here, too, weaker memory models can promote performance,

by allowing more aggressive compiler optimizations and more efficient use of weak hardware memory models [AG96]. As with any part of a language specification, it is the job of the compiler to make sure that when code is compiled for some specific hardware, the guarantees specified by the memory model of the programming language are upheld.

For both hardware and software, there is a specification (for the ISA and language, respectively) and many implementations (in the form of hardware and compilers, respectively). It is not always practical or feasible to implement a memory model exactly as specified, and implementations may forbid certain behavior that is allowed by the specification (but they may not allow behavior that is forbidden by the memory model).

2.5 The C11 memory model

Prior to the ratification of the C11 language standard in 2011, the C language did not have much builtin support for concurrent programming, nor did it have a real memory model. This made writing concurrent programs — especially *platform-independent* concurrent programs — both tedious and difficult. To rectify this, the C11 standard introduced a standardized memory model, and also extended the standard library with atomic data types, locks and more [ISO10]. In this thesis I discuss these atomic data types and the atomic operations that can be performed on them (both referred to as "C11 atomics" or "the C11 atomics library"). The C11 atomics library includes atomic loads and stores, as well as some atomic read-modify-write operations, such as compare-exchange and fetch-and-add. The C11 memory model and atomics library are described further in Section 4.1.

3 Objective and motivation

As shown earlier in the thesis, memory models are often complex and non-intuitive, both on the software- and hardware level. Nevertheless, it is essential that the memory model is accurately obeyed and implemented by both software and hardware, in order for concurrent programs to execute as intended by the programmer. Previously, I mentioned that compilers have an important role to play in this; when a concurrent C11 program is compiled, the compiler has to take into account both the C11 memory model and the memory model of the target architecture, and make sure that the semantics of the program is completely preserved after compilation. This is, of course, not trivial, especially for weak architectures like Arm and RISC-V.

This thesis is aimed at developing techniques for verifying the correctness of compilers with respect to the C11 memory model, focusing on atomic variables. Specifically, I present and detail a process for testing a compiler's implementation of C11 atomics. This process involves generating and running a set of test cases, and are intended to be able to show with high confidence that a compiler passes the tests. An important part of this is maximizing and quantifying that

confidence. Two secondary desired properties of this process are traceability, so that it is possible to analyze failed test runs after the fact, and reproducibility, so that test runs are consistent.

My focus is specifically on the IAR C compilers for ARMv7 and RISC-V, as well as their `gcc` equivalents. Both IAR and `gcc` compilers are available for a range of architectures, and so a tertiary ambition is for this process to be easily applicable to various architectures.

3.1 Delimitations

C11 introduced several features to help with parallel programs: a threading library, mutexes, futures and atomic variables. This thesis focuses only on testing atomic variables (e.g. `_Atomic int`) and the functions used on them, as well as fences. Moreover, I do not test the `consume` memory order. Its specification is fairly complex, and its implementation is currently very impractical [Imp15]; in C++17 and later, the use of `memory_order_consume` is discouraged until its semantics is reworked [ISO17].

I do not test the correctness of compiler optimizations, but rather focus only on testing that the compiler is correct at *some* optimization level. The correctness of common compiler optimizations under C11 is an active area of research (see for example Morisset et al. [MPZN13] and Vafeiadis et al. [VBC+15]). It is a complex topic on its own that requires a significantly different approach, and is out of scope for this thesis.

As stated above, I only test compilers for ARMv7 and RISC-V. Additionally, since I have not had access to any multicore RISC-V hardware, I only run hardware tests on Arm.

4 Theory

This section introduces some theory around memory models in general and the C11 memory model in particular. This theory is necessary to understand the rest of the thesis. The expert reader may skip (or skim) this section.

4.1 The C11 memory model

This section contains a brief and informal description of the C11 memory model, omitting locks since they are not dealt with in this thesis. For a more rigorous and complete description, see Boehm and Adve [BA08], or — for a formal description — Batty et al. [BOS+11].

Around the same time C11 was introduced, C++11 introduced an almost identical memory model into C++. The two are similar enough that they are often discussed as the same, the "C/C++11 memory model". Some of the literature referenced in this section refer to it only as the C++11 memory model, but it applies to C11 as well. Besides some formatting and language syntax,

the specification of atomic operations for the two standards are almost word-for-word identical [ISO10, ISO12].

C11 introduces *atomic* data types, which are qualified with the `_Atomic` keyword (i.e. `_Atomic int` gives an atomic integer type). These are types on which one can perform atomic operations. The operations available on atomic types are read operations, write operations and read-modify-write operations (e.g. fetch-and-add).

Besides the guarantees of atomicity, C11 also lends atomic operations some synchronization guarantees, dictating the order in which atomic operations may be executed and their effects may become visible. By default, atomic operations are sequentially consistent. They behave as if:

- There is a total order in which all atomic operations are executed, that respects the order in which the operations are sequenced (i.e. the order in which they appear in the code).
- Once an atomic operation is executed, its effects become instantly visible to all threads.

With SC atomics, we can correctly implement the store-buffering test from earlier in C11, as shown in Listing 3.

Thread 0	**Thread 1**
Initially, `_Atomic int x0 = 0, x1 = 0;`	
`atomic_store(x1, 1);`	`atomic_store(x0, 1);`
`if (atomic_load(x0) == 0)`	`if (atomic_load(x1) == 0)`
`puts("T0 won")`	`puts("T1 won")`

Listing 3: The store-buffering test from Listing 1, implemented in C11

However, as described by Boehm and Adve [BA08] enforcing SC can significantly impact a program's performance, especially on architectures where it requires inserting hardware memory barriers. For this reason, C11 allows the programmer to tag an atomic operation with an explicit *memory ordering*, in order to weaken the guarantees made about the operation. For example, tagging an operation with `memory_order_relaxed` removes all ordering guarantees from it. A store tagged with `memory_order_release` prevents reordering memory operations after it. A load tagged with `memory_order_acquire` prevents reordering memory operations before it. None of these memory orderings require the sometimes expensive total order of execution that SC does. Manually specifying memory orderings can be quite difficult, and using an ordering that is too weak can break one's program in subtle ways.

An important feature of the C11 memory model is that a program that contains *any* data race is considered to have undefined behavior (C11 defines a data race as two concurrent accesses to the same location where at least one is

a store, and at least one is non-atomic). This means that shared variables are not allowed to be accessed non-atomically, unless such accesses are protected by some synchronization primitive (e.g. a mutex lock) [ND16]. It also allows to "extend" some of the synchronization properties of atomic operations to non-atomic operations around them: a block of non-atomic operations surrounded by synchronizing sequentially consistent atomic operations can be thought of as sequentially consistent itself, since no other thread is allowed to observe an intermediate result of those non-atomic operations. Indeed, for a race-free program using sequentially consistent atomics, C11 guarantees SC for *all* memory operations, not just atomic ones [BA08].

To exemplify a few features of the C11 memory model, consider the program shown in Listing 4. The first thread stores a message to `val`, and then signals the other thread via an atomic flag. The release ordering used to update the flag ensures that the store to `val` is not reordered after the store to the flag. The second thread reads the value of the flag, and when its value is 1, we can be sure that this thread also sees `val = 42`. The acquire ordering additionally ensures that the read from `val` is performed *after* the flag is read. Thus, transitively the store to `val` always happens before it is read by the other thread. This program would have been just as correct had the two atomic operations been sequentially consistent (i.e. without giving them explicit memory orderings). In this case, however, there is no need for the extra synchronization that would provide. Thus, the explicit memory orderings let us relax the memory model and — hopefully — lets the compiler produce more performant code.

Thread 0	**Thread 1**
Initially, `_Atomic _Bool flag = 0; int val = 0;`	

`val = 42;`
`atomic_store_explicit(flag, 1,`
`  memory_order_release);`
	`while (atomic_load_explicit(flag,`
	`  memory_order_acquire) == 0);`
	`int received_val = val; // will always read 42`

Listing 4: The message-passing test, implemented in C11

C11 also allows inserting fences. These support the same memory orderings as regular atomic operations, but do not perform any memory operations.

4.2 Binary relations

As we will see below, binary relations are often practical when describing memory models. A binary relation is a set of ordered pairs of elements from some set A (this is a slightly narrow definition, but it suffices here). For every pair (x, y) in the relation, we say that x is related to y. For example, for $A = \{1, 2, 3\}$ we can define the relation *greater-than* — which relates each number to those smaller than it — as $gt = \{(3, 2), (3, 1), (2, 1)\}$.

The following are some relevant properties of binary relations:

- An *acyclic* relation is one that does not contain any cycles.
- An *irreflexive* relation is one in which no element is related to itself.
- A *transitive* relation R is one in which $(a, b) \in R \wedge (b, c) \in R \implies (a, c) \in R$.
- The composition $R; S$ of two relations R and S is given by $(a, b) \in R; S \iff \exists x : (a, x) \in R \wedge (x, b) \in S$.

As an example, the greater-than relation described above is acyclic, irreflexive and transitive.

4.3 Axiomatic formal models

Memory models are often specified in prose text written in natural language, such as in Section 4.1 above, in the C11 standard [ISO10] or in the AArch64 documentation [Lea]. In order to study them and remove the ambiguity that often comes with natural language descriptions, memory models are *formalized*. This thesis is concerned specifically with *axiomatic* formal models.

Axiomatic memory models represent programs as a set of memory events $\mathbb{E}$ (e.g. loads and stores), and then define a number of binary relations over $\mathbb{E}$ ordering the events. For example, the sequenced-before (`sb`) relation orders events from the same thread by the order in which they are sequenced (roughly, the order in which they appear in the code); for some $e_1, e_2 \in \mathbb{E}$, we can write $(e_1, e_2) \in$ `sb` to say that e_1 is sequenced before e_2. `sb` is determined entirely by the program, and does not change between runs of the same program. Other relations *do* vary between runs. Examples of such relations are `mo` (modification order), which orders all store events to the same location (e.g. variable, memory address), and `rf` (reads-from), which relates each store to any loads that took their value from that store. Note that the names and exact meanings of these relations vary from model to model, these are simply a few of those used for the C11 model. Together, an instance of all model relations describe an execution of the program.

The memory model is given its properties by specifying axioms over program executions (i.e. over the model relations), that must hold in order for an execution to be valid. As an example, a trivial axiom under all memory models is that `sb` (or its equivalent) must be irreflexive: it would not make sense for an event to be sequenced before itself. Another example used in some models is that given the modification order mo_l for each location l, $mo_l \cup sb$ must be acyclic. This states that writes to the same location must be performed in the order they appear in the code.

For a more in-depth explanation of axiomatic models, see Alglave et al. [AMT14].

The boolean nature of axiomatic models makes them well-suited to use with SAT solvers, something several of the tools used in this thesis take advantage of.

5 Methods

This section outlines the methods used for each step of the testing process, to generate tests and test compilers, and provides motivation for why the methods were chosen. The implementation of the methods are detailed and discussed in later sections.

A common approach to verifying the correctness of a compiler is to produce a formal proof. Doing so can be a daunting task, and requires sound formal models of both the language being compiled and the target architecture. For C11 atomics, there are formally proven compilation schemes for x86-TSO [BOS+11] and Power/Arm [SMO+12].

Another approach — the approach I take in this thesis — is to rely on testing. Testing can be used in cases where it would be too difficult to produce a formal proof, or as a complement to formal proofs; Trippel et al. used testing to find a mistake in a formally proven compilation scheme from C11 atomics to POWER [TML+17]. Moreover, while formal proofs are only able to prove that some mappings between C11 and some instruction set is correct, testing is able verify that a compiler correctly *implements* those mappings.

Testing brings with it its own set of difficulties, especially for concurrent language features. Writing and executing concurrent tests involves many of the same challenges that come with concurrent programming; reasoning about concurrent code is much more difficult than about serial code, and subtle bugs — whether introduced in the code code or by the compiler — are often difficult to detect and pin down. A concurrent program or test that runs correctly *once* is far from guaranteed to be free from errors.

To test a compiler, I use the following testing process:

1. Create a suite of parallel C11 test cases, where the allowed outcomes of each program are known.

2. Compile the tests with the compiler that is being evaluated.

3. Examine the reachable outcomes of each compiled program, and compare to the known outcomes allowed by C11.

If a compiled program is able to reach an outcome that is not in the list of allowed outcomes, it has been compiled incorrectly. Steps 1 and 3 can be considered in some separation, since the compiler acts as a stable interface between them. This testing procedure is similar to one published by Trippel et al. [TML+17], but differs in its purpose and implementation. Their methods are discussed further in Section 10.1.

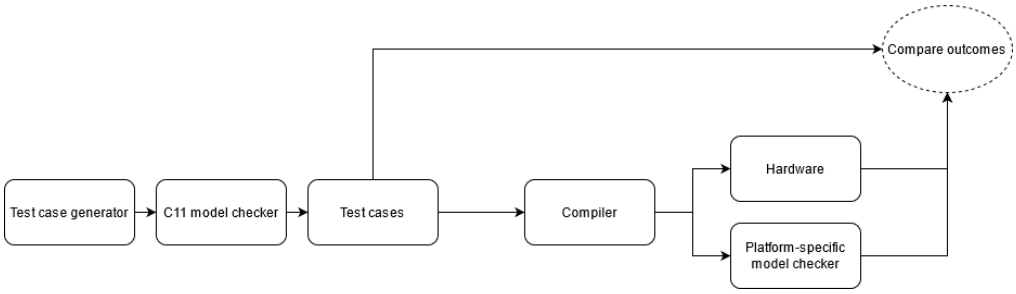

Figure 1: An overview of the testing process

I use IAR compilers and `gcc` to run tests on, focusing on the Arm and RISC-V architectures.

5.1 Generating test cases

The test programs used are *litmus tests*. These are small, parallel programs that demonstrate specific features of a memory model, and are meant to exhibit different behaviors under different memory models [LWPG17]; they are frequently used to test and study (the implementation of) memory models [LWPG17, AMSS12,ABD+15,BA08,FGP+16]. The mutual exclusion program described in Section 2.4 is an example of a litmus test. Litmus tests typically only contain a few memory operations per thread; the number of possible executions grows exponentially with the number of operations (the 'state space explosion' problem), making it difficult to reason about larger tests. Instead, a very large number of small tests tests are often used. This aligns with the 'small-scope hypothesis' [ADK03, OPP+12], which posits that testing a system thoroughly within a small scope can be just as or more effective than testing it incomprehensively within a larger scope.

For some hardware platforms, there are publicly available collections of litmus tests (e.g. for Arm [AMSS, ARMa]). However, no such collections seem to exist for C11; most research applying litmus tests to C11 only use a handful of test cases, as a means of highlighting specific features of the model rather than for rigorous testing. Moreover, it is non-trivial to rewrite test cases written in assembly to C11, since most ISAs do not have C11-style memory orderings.

To solve this, I use an automatically generated suite of test cases. I base my work on Lustig et al. and their `litmustestgen` tool, which models C11 litmus tests and uses a solver to find all litmus tests that are sufficiently interesting. The model considers all litmus tests up to some size bound, and should thus provide good coverage, but only generates tests that are as small and simple as possible, and should thus avoid generating redundant tests. Since `litmustestgen` does not determine the allowed outcomes of each test, I use `HERD` — an exhaustive memory model checker — to do this.

A few other methods for litmus test generation are discussed in Section 10.2, but they are not easily applicable to C11 and as such are not worth discussing

here.

5.2 Running compiled test cases

After creating tests and compiling them, the next step is exploring what outcomes are reachable by the resulting executables. The two methods often employed for this are:

- Running the test through a model checker or similar (e.g. `HERD` [AMT14]) that can exhaustively enumerate the reachable outcomes.
- "Stress testing" by running the test repeatedly on real (or simulated) hardware and recording which outcomes occur.

I prefer the model-checking approach, since it is exhaustive. This is the method I rely on the most, using the `HERD` memory model simulator, which enumerates all allowed outcomes for a test under a given memory model. The TriCheck tool mentioned earlier also uses a model checking approach. However, rather than verifying tests against the ISA memory model, TriCheck verifies them against a description of some microarchitectural implementation of that memory model, with the goal of verifying that a compiler and microarchitecture collectively uphold the C11 memory model. This opens the possibility for microarchitectural models that are stricter than the ISA memory model to conceal errors introduced by the compiler. Since I *only* aim to test the compiler, I use the more appropriate (and simpler) `HERD` tool, to check the tests directly against the ISA memory model.

The stress testing approach is less reliable than model-checking, since it can only prove that any given execution is allowed, not that it is forbidden. However, stress testing has some practical advantages over model checking:

- It can easily be applied to almost any platform, without requiring a model checker or a (sound) formalization of the ISA's memory model. `HERD`, for example, only supports five architectures.
- It supports the full instruction set, and so can run virtually any test, such as those involving interrupts. For RISC-V processors that do not support the atomics portion of the ISA, the IAR compiler deals with interrupts; such implementations cannot be tested with `HERD`.
- It scales better with the size of the tests. This becomes relevant when running tests for Arm that contain several RMW operations, as we will see later. Moreover, if one wanted to also run larger, more realistic programs, model checkers would be altogether too slow to use.

When given enough time, stress testing has been shown to be effective at finding even rare executions [AMSS11, AMT14, AMSS12, HVM+04]. Some amount of research has been done on how to maximize the number of outcomes found by stress testing, specifically on how to schedule and synchronize threads [MMSM20],

and how to cleverly stress the hardware (although only on GPUs) [ABD+15]. Besides this, I could try experimenting with different CPUs (number of cores, clock speed), or running tests while artificially flooding the coherence protocol.

I implement both stress testing and model-based testing, and compare the two methods to find out if stress testing can reach a satisfactory degree of confidence.

5.3 Evaluation plan for the testing process

I evaluate the testing process in three ways:

- Whether the process is able to identify compilers with known bugs in their compilation of C11 atomics, as faulty. I am not aware of any such bugs in popular compilers, so instead I artificially create bugs by modifying compilers to use known incorrect compiler mappings. This serves as a crude metric of the coverage of the test suite, and demonstrates the viability of the process as a whole.

  For Arm, I use the leading-sync and trailing-sync mappings, first described by Batty et al. [BMO+12] and supposedly proven correct for Arm and Power (the two architectures are similar enough that the proof applies to both). However, a mistake was later found in the proof [MTL+16]. At the same time, the trailing-sync mapping was later proven *incorrect* for both architectures [MTL+16] using counter-examples found with the TriCheck tool mentioned earlier [TML+17]. Counter-examples were also found for the leading-sync mapping by other researchers [LVK+17], but they only confirmed that these invalidated the mapping to Power, and did not investigate whether the counter-examples also applied to Arm.

  For RISC-V I use a made-up bug, since I have not found any in the literature. Trippel et al. have published two compiler mappings with similarly subtle bugs as the Arm ones mentioned above (in fact, they concluded that there are no valid C11-atomics compiler mappings for RISC-V) [TML+17]; however, their findings have been addressed with recent changes to the RISC-V specification [WAR19], and so their mappings are no longer buggy.

- To what degree and with what frequency stress testing is able to produce the different reachable outcomes for each test case. If it is not able to reliably find rare outcomes, it will not be effective at finding bugs. On the other hand, if it *is* effective at finding all outcomes, that could justify using stress testing instead of an exhaustive verification tool. I use the `HERD` tool [AMT14] mentioned earlier as the source-of-truth on what outcomes are possible for each compiled test, and compare that to the outcomes reachable with stress testing.

- Whether the test generator is able to find classic litmus tests from the literature. Alglave et al. define several families of litmus tests [Lit], classified according to what memory operations they contain, and how the operations interact. Many of there litmus tests frequently occur in real-world

code [AMT14]. I evaluate whether the test generator finds tests in these families, excluding a few families that are only interesting for Arm and POWER.

6 Synthesizing test cases

As mentioned in Section 5, I use the Alloy-based `litmustestgen` tool published by Lustig et al. [LWPG17] to generate test cases. The tool uses a model of a basic C11 program containing only memory operations, and applies an axiomatic C11 model by Batty et al. [BDW16] to the program model. Running this model generates all valid programs and executions in this subset of C11 (up to some bound on the number of memory operations).

In order to generate only those test cases that are most likely to expose errors in a memory model implementation, the authors apply a "minimality criterion" to the model. This criterion is defined around two *instruction relaxations*: one removes a memory operation, and the other demotes its memory order (reduces it to a weaker order). An execution is defined to be minimal if and only if the following constraints both hold:

- The execution is forbidden by the C11 memory model.
- No instruction relaxation can be applied to any of the memory operations without allowing the execution.

The minimality criterion ensures that each generated test contains a forbidden execution, and that it contains just enough memory operations and synchronization to forbid that execution. Lustig et al. mention a third instruction relaxation, which decomposes a RMW operation into an atomic read and an atomic write. However, they do not implement this relaxation for C11, most likely because the C11 model they use makes it too difficult.

I make a few restrictions to the search space of the test generation model in order to remove tests that I consider uninteresting. This saves a significant amount of time when executing the generated test suites (and probably also when generating them). First, I do not generate tests that have only a single thread. The compilation of single-threaded programs is probably best tested using other methods, such as the one described in Section 10.3. Second, I choose not to generate test cases that contain nonatomic variables or operations. Because I do not generate tests that contain locks, or any other form of conditional execution, every operation in a test is concurrent with all other operations in any other threads. Recall that C11 forbids data races; if two operations concurrently access the same variable, they must either both be reads or both be atomic. For these tests, there are then three scenarios in which a variable *does not* have a data race:

- All accesses to it are from the same thread.
- All accesses to it are reads.

- All accesses to it are atomic.

The first two cases are not interesting, since the value of the variable will not vary between executions, and thus cannot be used to detect faulty executions. If we leave those out, we are left generating test cases with only atomic variables and operations.

I also change how the model handles variable initialization. C11 forbids indeterminate reads (reads from uninitialized variables). When running litmus tests, all variables are typically initialized before the test is run. However, `litmustestgen` does not model variable initialization; it treats any read from a variable's initial value as an indeterminate read, and thus disallows executions containing such reads. As a result, it fails to generate the classic store-buffer and message-passing tests seen earlier, among others. To improve this, I allow indeterminate reads and take any such read to instead be a read from the variable's initial value. By modifying the `fr` relation to include indeterminate reads, there is an implicit initialization event that appears first in the modification order for each variable. I have verified that for a bound of 5 events, the tests generated with this modification is a strict superset of those generated without it.

Finally, in order to save time both when generating tests and when running them, I exclude any test that contains two or more RMW operations. I mention above that the model does not implement the third instruction relaxation (decomposition of RMWs). This means that generated tests containing RMWs are not necessarily minimal, and that it is slightly more safe to remove them, compared to tests without RMWs. However, my main reason for excluding tests with several RMWs is that running such tests for Arm with `HERD` is extremely slow; both compilers tested implement RMWs on Arm using loops, and `HERD` does not handle loops well. While a test without RMWs takes at most a few seconds to run, a test with two RMWs can take several minutes, one with three RMWs around 15 minutes and one with four RMWs several hours. For more complete results one could — and probably should — use the complete test set, especially for architectures that do not suffer from slow `HERD` execution times.

6.1 Instantiating tests

The Alloy model described above generates a set of abstract test cases, each consisting of several threads of memory events. Each memory event specifies a type (read, write or RMW), a variable and a memory order. An example is shown in Figure 2.

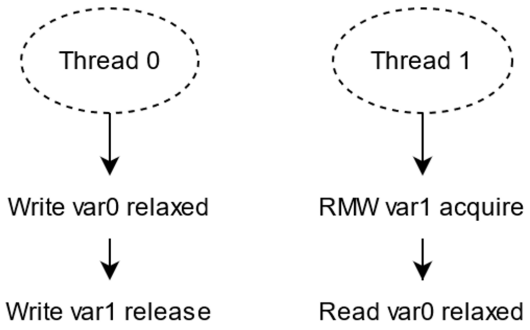

Figure 2: An example abstract test case generated by `litmustestgen`, consisting of two threads with two memory operations each.

Note that these events do not specify the values to use for write or RMW operations; in order to produce a runnable test from an abstract test case, I first need to assign values to the operations that need it. The assignment has to be done in such a way that each distinct execution is mapped to a unique outcome, so that it is possible to differentiate between allowed and forbidden executions. `litmustestgen` uses a model by Batty et al. [BDW16], in which the reads-from and modification order relations uniquely identify an execution: all other relations are either derived from these two, or are the same for every execution. Thus, we need to ensure that we can identify these two relations from the test outcome.

As long as all writes store unique values, the reads-from relation is easily identifiable from the outcome, since the source of a read can be inferred from its value. The same is true of the modification order as long as there is a maximum of two writes per variable, since the final value of each variable is then enough to infer the modification order. For variables with more than three writes, one can add extra reads to the test, as documented by the `diy` tool [Gen]. However, since merely 18 of the 3025 tests generated would need it, I opt not to amend those tests. Note that RMWs do not count: all C11 RMWs return the value they read, and thus their place in the modification order is always unambiguous.

Ensuring the uniqueness of every regular write is simple (all it takes is assigning each write a unique value). Ensuring the uniqueness of RMWs is more difficult, and requires choosing the values of both writes and RMWs such that each RMW, no matter what value it reads, produces a new unique value. Table 1 specifies the how the values are determined for each RMW type. Note that the table includes values for tests with more than one RMW operation, but as discussed earlier I do not use any such test in this thesis.

Note also that the abstract test case in Figure 2 does not indicate a specific RMW function to use, since all RMW functions are treated the same by the memory model (with the exception of `atomic_compare_exchange`, which I

RMW function	Initial value	Write values	RMW values
atomic_exchange	$I = 0$	$W_0 = 1$ $W_n = W_{n-1} + 1$	$X_0 = \|W\| + 1$ $X_n = X_{n-1} + 1$
atomic_fetch_add	$I = 0$	$W_0 = 1$ $W_n = W_{n-1} + 1$	$X_0 = \|W\| + 1$ $X_n = 2 \cdot X_{n-1} + 1$
atomic_fetch_sub	$I = 0$	$W_0 = 1$ $W_n = W_{n-1} + 1$	$X_0 = \|W\| + 1$ $X_n = 2 \cdot X_{n-1} + 1$
atomic_fetch_or	$I = 0$	$W_0 = 1$ $W_n = W_{n-1} + 1$	$X_0 = \|W\| << 1$ $X_n = X_{n-1} << 1$
atomic_fetch_xor	$I = 0$	$W_0 = 1$ $W_n = W_{n-1} + 1$	$X_0 = \|W\| << 1$ $X_n = X_{n-1} << 1$
atomic_fetch_and	$I = 2^{\|X\|} - 1$	$W_n = 2^{\|X\|+n+1} - 1$	$X_n = (2^{\|W\|+\|X\|} - 1) \oplus 2^n$

Table 1: A specification of the set of values used for writes (W) and RMW operations (X). The order in which the values are used does not matter.

choose not to handle in this thesis). In order to cover all RMW functions, I create six concrete tests for each abstract test case with a RMW operation, one for each of the different RMW functions. This also poses an interesting question for tests with more than one RMW operation, which I happen to avoid by excluding such (for other reasons, as described above). For such tests, one might think to create one concrete test for every combination of RMW functions, but this would give an absurd number of tests. A better alternative might be to create one test per RMW functions, where all RMW operations use the same function; it seems reasonable that a RMW function with insufficient synchronization would be as or more likely to display a forbidden outcome interacting with itself, as it would interacting with another RMW function.

6.2 Finalizing tests

Once the values are determined, I generate equivalent C11 code for each thread of memory events, and store it to a file so that it can be compiled later. Listing 5 shows the code generated for the abstract test case shown earlier and for one of the RMW functions.

```
_Atomic int var0 = 0, var1 = 0;
int n0, n1;

void thread0() {
  atomic_store_explicit(&var0, 1, memory_order_relaxed);
  atomic_store_explicit(&var1, 1, memory_order_release);
}
void thread1() {
  int n0 = atomic_fetch_add(&var1, 2, memory_order_acquire);
  int n1 = atomic_load_explicit(&var0, memory_order_relaxed);
}
```

Listing 5: A finalized test case for the events in Figure 2, using the `atomic_fetch_add` RMW function.

Note that for the read and RMW operations, I write the return value to a unique nonatomic variable so that it can be retrieved after the test has been run. These extra write operations do not change the allowed outcomes. All axioms of the C11 memory model are either irreflexivity axioms on some composition of relations, or acyclicity axioms. The extra writes I introduce affect *only* the `sb` relation, and because `sb` is transitive, any cycle containing the extra write could be formed without it, and any composition of `sb` would not have its reflexivity affected by the extra write.

Besides the code, for each test I also store the name and initial values of all variables it uses. Finally, for each test I identify and store the set of allowed outcomes under C11; this is the set that I then compare against the set of reachable outcomes after compilation. To identify the allowed outcomes I run the C11 code through `HERD`, a memory model simulator which is described in more detail in Section 7.1. I use one of the default C11 models provided with `HERD`, which is based on the original C11 formalization by Batty et al. [BDW16, Ove].

7 Running test cases

As mentioned earlier, I run each generated test case by creating a C program containing its code, compiling the program, and then identifying the reachable outcomes of the compiled program. In order to determine the reachable outcomes of the compiled program I use both a simulator (`HERD`) and real hardware.

7.1 HERD

The `HERD` tool is a memory model simulator published by Alglave et al. [AMT14]. As input, `HERD` takes a test file containing variable (or memory) initializations and some code to run in different threads, as well as a an axiomatic memory model specification. Then, it uses a SAT solver to find all executions of the

test file (and thus all outcomes) that are valid under the memory model. An example test input file can be seen in Listing 6. `HERD` supports C11, Arm assembly, RISC-V assembly and a few other languages. I use the default Arm and RISC-V memory models that come with `HERD`.

```
ARM mp_relacq
{
% initialize stack & frame pointers
0:SP=0;0:R11=0;1:SP=1000;1:R11=1000;
% initialize variables
100628=0;100624=0;100632=0;100636=0;
}
P0                | P1;
movw R3, #35088   |          movw        R3, #35088;
movt R3, #1       | movt        R3, #1;
mov R2, #1        | ldr R2, [R3, #4];
str R2, [R3]      | dmb sy;
dmb sy            | str R2, [R3, #8];
str R2, [R3, #4] | mov R1, #2;
                  | str R1, [R3, #12];
                  | cmp R2, #1;
                  | movweq R3, #35088;
                  | movteq R3, #1;
                  | ldreq R2, [R3];
                  | streq R2, [R3, #12];
% addresses of variables
locations [100636;100632;100624;100628;]
```

Listing 6: A `HERD` input file for the message passing test described earlier, compiled for Arm using gcc.

7.1.1 From compiler to HERD

Converting a C11 test into an assembly equivalent that is runnable with HERD is somewhat complicated. Since `HERD` runs assembly code, not binary files, I first need to compile the code for each thread and extract the assembly code. I create a file containing an empty `main` function, and a function for each thread containing its code. I then compile and link this file with the compiler to test, disassemble the output, and extract the assembly corresponding to each thread.

Next, I create the memory initializations at the top of the `.litmus` file. For each variable, I look up its address in the symbol table of the executable, and initialize the memory location at the top of the test file. I also add each address to the locations at the bottom of the file, so that `HERD` knows to output the values at these addresses for each outcome.

The runtime of `HERD` scales superlinearly with the number of memory operations a test contains, and in some cases compilers output unnecessary memory

operations that make tests unfeasible to run. First, gcc compilers make heavy use of the stack when optimizations are turned off. While I generally avoid optimizations to make it easier to map between assembly and the original C code, for gcc I turn on debug-friendly optimizations (`-Og`) and turn off whole-program optimizations (`-fno-whole-program`), which resolves this issue. Second, the Iar Arm compiler loads variable addresses from a static section of the executable, rather than loading them as immediates. I convert those loads into immediate loads before passing the assembly to `HERD`.

`HERD` has a somewhat limited instruction support, and compilers sometimes output instructions that `HERD` does not support. For some instructions, I replace them with equivalent supported instructions. For instructions without a supported equivalent, I have extended `HERD` to support them.

7.1.2 Running HERD tests

After generating a `.litmus` file, I run it with `HERD`. `HERD` then generates all possible outcomes, that is, a list of all possible final values at the memory addresses specified at the end of the file. The addresses are converted back to variable names using the symbol table of the compiled executable.

7.2 Stress testing on hardware

As discussed in Section 5, running tests on hardware is sometimes more practical and allows testing parts of the ISA that model checking tools do not implement.

Each test case is compiled with a test harness that runs the test multiple times while tallying the outcomes. The code for each thread is placed in a loop running as many iterations as desired, and each loop is run in a separate thread. Every iteration, these threads synchronize so that all threads run the same iteration at roughly the same time. For each variable in the test, the harness allocates an array of size equal to the number of test runs. Then, for all memory accesses, the iteration number is used to index into these arrays, so that each test iteration accesses a unique set of memory locations. When all test iterations have been run, the arrays are scanned to determine the outcome of each run (by determining the final value of all memory locations). The test harness is run bare-metal on a Cortex-A9 processor, with each thread running on its own dedicated physical core. The final tally of outcomes is transmitted from the chip using a debug probe and semihosted I/O.

7.2.1 Optimizing for variability of outcomes

Running tests on hardware has no guarantee of finding all reachable outcomes, and certain outcomes that depend on rare hardware behavior may be *very* rarely produced. To increase the number of outcomes found, the most obvious solution is to increase the number of test runs. However, there are a number of other factors that affect the outcomes found.

Possibly the most important factor is the type of barrier used to synchronize the threads. The barrier is responsible for the majority of test runtime [MMSM20]. A good barrier is fast so as to save runtime, and not too exact so as to allow some variation in executions. Melissaris et al. [MMSM20] compare various barrier types methods used by Alglave et al. [AMSS11, Run], and find that the timebase and userfence barriers perform the best in almost all tests and by almost all metrics. Thus, these are the two I implement and evaluate. Melissaris et al. also present a method of running litmus tests without barriers, which significantly improves performance. However, their method does not seem to be able to observe the modification order of test runs; the authors only explain how to observe the reads-from relation, and the method is only applied to tests where the modification order is insignificant. Since it would only be applicable to a subset of all tests, I do not implement the test method of Mellisaris et al.

Another important factor is the array access pattern between test instances. As described, each test iteration is allotted a set of unique locations in the arrays representing the test variables. The most obvious allotment is sequential, so iteration i of a test that accesses some variable x, accesses the i-th element of the x array. However, experiments have shown a much greater variability of outcomes when the allotment is randomized, so that the same iteration i would instead access some random (but unique) element of the x array [AMSS11]. Thus I use a randomized allotment.

There are a few more factors that affect the outcome variability. These include the scheduling and affinity of the test threads (not applicable to bare-metal programs), explicit prefetching between test iterations, and external stress put upon the coherence protocol. These factors are not very well studied, and it is either not clear how to best apply them, or how large of an effect they would have. For this reason, I choose not to implement these methods, and instead leave them as future work.

8 Results & discussion

In this section, the results are described and discussed according to the evaluation criteria laid out in Section 5.3. First, I evaluate the generated test suite. Then, I examine the accuracy of the tool when applied to both gcc and IAR compilers, including some deliberately buggy compilers. Last, I compare the two test-running methods, stress testing and model checking.

To run tests in this section, I use a tool I built in ruby implementing the methods I describe in the thesis. Thus, this tool is capable of both generating test suites and running them with either of the methods described earlier. In this section, I refer to this tool simply as "the tool".

8.1 Tests generated

Generating tests is very slow, and generating tests with a bound larger than six memory events seems unfeasible. This is an acceptable limit; of the litmus

tests families defined by Alglave et al. [Lit], only one is larger than six events. Moreover, Mador-Haim et al. [MAM10] have shown that litmus tests of at most size six is sufficient to prove equivalence between TSO-like models, and while this is not directly applicable to the C11 model it at least shows six events is not *too bad* of a bound.

Table 2 shows the number of tests generated for each test size. Unsurprisingly, the number of tests increases exponentially with the test size.

Test family	**Tests found**
4	90
5	472
6	2463
Total	3025

Table 2: A tally of the number of tests found in each of the families defined by Alglave et al.

Section 5.3 mentions the litmus test families defined by Aglave et al. [Lit], which commonly occur in other litmus test suites and some of which correspond to commonly used programming patterns. Table 3 shows the number of tests belonging to each family in the suite of tests generated for this thesis. All families have at least one test belonging to them; some have several, typically in the form of variants of the same test but with various fences inserted. These numbers differ from other litmus test suites, such as those generated by Alglave et al. for Arm [SSA$^+$] and the Linux kernel [AMM$^+$18], which have tens of tests — or more — for each family. However, the vast majority of those tests are nonminimal, and some do not even have any forbidden outcomes. These suites may be designed to study some model or model implementation in depth, rather than to simply test its correctness or compliance with some specification. Then, it would be interesting to include tests that are expected to pass. Also, since most other suites do not include RMWs, they would be very small if they only included minimal tests (for C11, there are only 121 minimal tests below seven operations that do not contain RMWs). Considering it is fairly cheap to run a litmus test, it's possible that some nonminimal tests are included simply for good measure.

8.2 Finding compiler bugs

As described earlier, I test both the IAR and gcc compilers for Arm and RISC-V, using the `HERD`-based testing method. The results are shown in Table 4. As is shown in the second third colum, no bugs were found in any of the unmodified compilers.

Section 5.3 described how the leading-sync and trailing-sync compiler mappings from C11 atomics to Arm were thought to have been proven correct, but were later shown to be incorrect in some cases [MTL$^+$16, LVK$^+$17]. As far as I am aware, no popular compiler has ever used this compiler mapping; most

Test family	Tests found
MP	8
S	4
SB	3
R	3
2+2W	3
LB	4
WRC	6
WWC	3
RWC	4
WRW+WR	3
WRR+2W	8
WRW+2W	4
IRIW	2
IRWIW	2
IRRWIW	2
ISA2	1
W+RWC	1
3.SB	1
3.LB	1
3.2W	2
Z6.0	1
Z6.1	2
Z6.2	1
Z6.3	1
Z6.4	2
Z6.5	2

Table 3: A tally of the number of tests found in each of the families defined by Alglave et al.

compilers use a variant of the trailing-sync mapping, with a stronger mapping for acquire-loads. I use two modified versions of the IAR Arm compiler, using the exact leading-sync and trailing-sync mappings respectively, and test the compilers with the tool developed for this thesis. The trailing-sync compiler fails 6 tests, two of which are the IRIW and WRC counter-examples already found by Manerkar et al. [MTL$^+$16]. The leading-sync compiler fails 6 tests, of which neither are the two counter-examples found by Lahav et al. [LVK$^+$17].

I also run tests against a modified version of the IAR RISC-V compiler, again with a weakened compiler mapping. Normally, the compiler maps a SC-load to a load (`lw`) followed by a `fence rw,r` instruction. My version changes the fence to `fence w,r`, thus weakening it. With this change, 48 tests fail.

It is impossible to know for certain whether the reason the tool does not find any bugs in the unmodified compilers is that there are no bugs in them, or that the tool is not good enough to find them. However, the fact that all modified

Architecture	Compiler	Failed tests	Unobservable outcomes
Arm	IAR	0	6.7%
Arm	gcc	0	6.7%
Arm	IAR (leading-sync)	6	2%
Arm	IAR (trailing-sync)	97	2%
RISC-V	IAR	0	6.7%
RISC-V	gcc	0	6.7%
RISC-V	IAR (modified)	48	4.4%

Table 4: The results from applying the testing tool to various compilers.

compilers are correctly identified as faulty should lend some credibility to the tool. The two Arm bugs, especially, are very subtle, and demonstrate that the tool is able to find even bugs that only present themselves very rarely. Recall that, as mentioned in Section 5.3, TriCheck [TML+17] only found test failures for the trailing-sync mapping and not the leading-sync mapping. According to the authors, this was because their test case generator did not support fences.

Interestingly, the tool does not find either of the two leading-sync counter-examples published by Lahav et al [LVK+17]. One of those counter-examples contains eight atomic operations, and is thus too large for the tool to find when using a test size limit of six operations. The other counter-example only contains six operations, but is non-minimal; the tool does not find the exact same example, but instead finds a smaller equivalent test. Recall that Lahav et al. only show that this test fails on Power, but do not evaluate it on Arm (see Section 5.3); I am able to show that it does *not* fail on Arm. Lahav et al. discuss that a lightweight barrier used when compiling the test to Power is what causes the test to fail. However, Arm uses a full barrier instead and thus does not fail the test.

All tests that fail for the leading-sync scheme are tests that contain an `atomic_exchange` RMW operation. It appears that this operation is slightly weaker than the other RMW operations, possibly because it has no dependency between the loaded value and the stored value. A slight change to the implementation of the `atomic_exchange` operation might thus be enough to make the leading-sync mapping valid for Arm.

The second column of Table 4 shows the portion of allowed C11 outcomes that are not observable after compilation. A low number indicates that the compiler and hardware implement much of the relaxed behavior allowed by the C11 model, which in turn is a sign of good performance of the compiled programs. We can note that the modified (buggy) compilers have significantly fewer unobservable outcomes compared to the regular ones. If the C11 model is changed so that the leading- and trailing-sync mappings become valid, as has been proposed [LVK+17], they might be able provide better performance than the mappings used today. Note, however, that the tests used in this thesis are not very well suited to this type of analysis. The advantage of generating minimal tests is that they only require a little less synchronization for a forbidden

outcome to become visible, but they may (and probably do) require a lot of extra synchronization before an allowed outcome becomes unobservable. If one wanted to study the unobservable outcomes of some compiler, one would be better off generating a suite of non-minimal tests. One could invert the minimality criterion to generate "maximal" tests, for which no memory ordering could be promoted without hiding some allowed outcome. An improvement to my process would be to generate both a minimal and a maximal test suite, to detect both forbidden outcomes (for conformance testing) and unobservable outcomes (as a performance metric). Then, compiler developers could dial in a performant mapping by trying to reduce the proportion of unobservable outcomes while making sure none of the tests fail.

8.3 Comparing model checker to stress testing

In this section, I compare the `HERD` simulator described in Section 7.1 to the stress testing hardware-based test running method described in Section 7.2. All tests are run using gcc, for Arm only (as stated in Section 3.1, I have not had access to suitable RISC-V hardware).

Table 5 compares the results of the two methods, broken down per test size. It is apparent from these results that my stress testing method is not very accurate; only about a third of all outcomes are visible with it. This inaccuracy translates to a poor ability to find bugs. For the tests run in Section 8.2, the stress testing method did not give a single test failure for either of the two buggy Arm compilers, and thus incorrectly classified them as bug-free. The advantages mentioned earlier are clearly not enough to justify using this method where model-checking is possible.

One should be careful to take these results to mean that stress testing is *never* effective. As mentioned in Section 2.4.1, and as demonstrated in practice by Alglave et al. [AMT14], some behavior that is allowed by the ISA may not be visible on all processors. It is possible that the processor I use implements a significantly stronger version of the Arm memory model, and that this is part of why the outcome statistics are poor. The problem could be diminished by running tests on several *different* processors; a larger set of processors would give a wider range of visible behaviors that more accurately represents the ISA model. Recall also that in Section 7.2.1 I mention a few stress testing methods that I do not implement, because i am unsure of their effectiveness. By experimenting with these methods and by using multiple processors, it is possible that stress testing could reach a tolerable level of accuracy, but more research would obviously have to be done.

9 Conclusion

The testing process presented in this thesis is aimed at testing C compilers' implementations of the C11 atomics library and memory model. The process is based on a suite of C11 litmus tests that are compiled to some target platform;

Memory operations (Number of tests)	C11 outcomes Average[1]	HERD outcomes Average visible[2]	HERD outcomes % visible	Stress testing outcomes Average visible[2]	Stress testing outcomes % visible
4 (90)	2.7	2.7	100	1.0	33
5 (472)	5.1	5.0	97	2.1	40
6 (2463)	11.2	10.4	93	2.6	33
3–6 (3025)	10.0	9.3	93	2.5	34

[1]The average number, per test, of outcomes allowed by C11.
[2]The average number, per test, of outcomes visible with `HERD` and stress testing, respectively.

Table 5: A comparison of the outcomes visible via `HERD` and via stress testing.

the compiled programs are then evaluated to find their possible outcomes. For a compiler to be considered correct, no compiled test program must have an outcome that is disallowed by the C11 standard.

I show that the process is able to detect three out of the three known bugs tested. One of them is a bug that TriCheck (a similar testing tool) failed to find. I find that the published counter-example to the leading-sync mapping for Power does not apply to Arm, and my results indicate that the only invalid part of the leading-sync Arm mapping is the `atomic_exchange` RMW operation. If so, the mapping could be fixed simply by changing the `atomic_exchange` implementation, rather than changing the C11 model as has been proposed by Lahav et al. [LVK+17]. I find no bugs in any of the compilers I tested.

The C11 litmus test suite I use is generated using a model-based approach which, unlike other common test generation methods, does not require any guidance from the user. Unlike the suite used by Trippel et al. [TML+17], my suite encompasses both fences and RMW operations. The suite includes several instances of tests from common litmus test families, although not as many as most other test suites. The tests generated this way are well-suited to testing conformance of some memory model implementation to a specification, but are likely less useful for other applications of litmus testing.

I evaluate two methods for exploring the behavior of compiled tests: one using model checking, and one using hardware. The model checking method is deterministic and in *most* cases fast enough to be viable, but is very slow at handling RMW operations for Arm and sometimes suffers from poor instruction support. On the other hand, I find the hardware-based method to be mostly unsuccessful; even though it is more portable than model checking, it is too inaccurate to justify its use. However, there are some possible avenues for improvement (such as using multiple processor models) which could be explored in the future. Given its ability to find compiler bugs (as demonstrated with the modified compilers), this testing process could be useful for automated conformance testing of compilers. Even though I focus on Arm and RISC-V in this thesis, nothing about these methods are specific to those architectures. Power would be an easy next target considering it is well supported by `HERD`, but

it would be interesting to apply these methods to an architecture that is less well-studied.

Before too much faith is placed in this method of testing, however, further evaluation would have to be done to determine exactly how accurate it is. Moreover, since this process is not able to exhaustively test the correctness of compiler optimizations, it would need to be combined with other tools or methods in order to completely test a compiler.

10 Related work

This section describes other work that is related to this thesis, or may be interesting to the reader. Related work that has already been discussed in the thesis is not mentioned here.

10.1 Full-stack memory model verification

The TriCheck tool published by Trippel et al. [TML+17] tests a compilation scheme and hardware implementation together, to verify that a compilation scheme, an ISA memory model and an implementation of the ISA together adhere to the C11 memory model.

The process involves translating automatically generated C11 litmus tests into an assembly language, and verifying their behavior against a model of some hardware. This process is unable to distinguish between test errors introduced by the compiler and by the hardware model; my `HERD` method verifies tests directly against the ISA memory model, and is thus more appropriate when *only* testing a compiler.

Trippel et al. use a fairly simple method for test generation; common litmus tests are used as templates, and C11 variants of them are brute-forced by enumerating all possible memory orderings for the atomic operations. This almost certainly creates lots of redundancy — tests that are too similar to others be interesting, or that behave exactly the same as other tests. Moreover, Trippel et al. do not generate tests with RMWs or fences.

Trippel et al. do not use a compiler to compile the tests, but rely on a predefined mapping between C11 atomics and hardware instructions. This makes their methods less suited for continuous automated testing with a real compiler, since one would have to update this mapping with every change to the compiler.

10.2 Litmus test generation

Pseudo-random concurrent programs have been used for a long time to help with the design and testing of multicore processors [OMK+95,AS02]. Whether or not these can be considered litmus tests is probably a matter of opinion. One of the first generators using the term "litmus test" was published by Mador-Haim et al. [MAM10], capable of generating minimal litmus tests that demonstrate differences between two memory models.

Another method was later developed by Alglave et al. for the `diy` tool [AMSS12]. `diy` has been used to generate useful suites of litmus tests for various weak memory models [AMT14, AMSS12, ABD+15]. There is no published research using `diy` for C11, but Alglave et al. has used it to generate a litmus test suite for the linux kernel memory model, which is a model fairly similar to the C11 model. Unlike the `litmustestgen` tool by Lustig et al. (the tool I use in this thesis), `diy` is not fully automated, and requires careful user guidance to produce test suites with good coverage. `diy` produces tests containing cycles of relations that are forbidden by the memory model, and the user is responsible for specifying what relations to construct the cycles from. This works well for memory models that can be specified in terms of just a few acyclicity axioms, but all formalized C11 models have too many axioms — of which too few are acyclicity axioms — to be able to easily apply to `diy`. Moreover, `diy` does not support C11 RMW operations.

Several more methods for test generation have been published. Many methods focus specifically on chip design, and verifying memory model implementations at the hardware level. Qin and Mishra [QM12], Elver and Nagarajan [EN16] and Andrade et al. [AGdS20] all generate tests with the aim of maximizing coverage of a cache coherence protocol.

Some of these methods seem promising but are not easily applicable to C11, since they have no strategy for generating C11-style memory orderings, and (except in the case of `diy`) are too specific to hardware testing.

10.3 Sequential compiler testing

Naturally, compilers need to be tested also on their ability to correctly compile *sequential* code. One effective method for this, that is particularly related to this thesis, is differential testing on randomly generated programs, as done by Yang et al. using csmith [YCER11]. Random (but valid) C programs are generated and then compiled by a set of compilers under a range of optimization levels. If every program does not have the same outcome for every compiler and optimization level, there is a bug in at least one of the compilers. This method is difficult to apply to concurrent programs, the outcomes of which are typically non-deterministic. Moreover, generating valid concurrent programs beyond small litmus tests is a much greater challenge than generating comparable sequential programs.

10.4 Validating optimizations under C11

Some compiler optimizations that are common for sequential programs can not easily be shown to be correct under C11 (since concurrency is allowed). The correctness of these optimizations is an active area of research, see for example Morisset et al. [MPZN13] and Vafeiadis et al. [VBC+15]. Research in this area focuses on larger programs involving more language constructs than just atomic operations (i.e. more "natural" programs), and — where compilers are involved

— starts from the assumption that a compiler is able to generate correct unoptimized concurrent code.

10.5 Adjustments to the C11 memory model

Batty et al. were the first to publish a formalized version of the C11 memory model [BOS+11]. At the same time, they pointed out some problems with the C11 model and proposed some fixes that were later integrated into the standard. Later, Batty et al. proposed a number of simplifications to the behavior of SC atomics [BDW16]. Lahav et al. also proposed changes to SC atomics [LVK+17] that restore the correctness of the trailing-sync and leading-sync compiler mappings discussed earlier, without the need to patch compilers or hardware.

10.6 Noise-based testing

Noise-based testing is a technique sometimes used to test concurrent software, where semi-random delays are inserted into the different threads of a program [FHK+15] in order to force different executions. Generally, this is based on a sequentially consistent view of the software, and the delays are inserted so as to generate a variety of interleavings. When testing non-SC memory models, we are more often than not interested in behavior stemming from operations being performed *at the same time*, and inserting delays directly counteracts this. However, a very small amount of delays may be useful even for running litmus test, but to my knowledge this has not been researched.

References

[ABD+15] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU Concurrency: Weak Behaviours and Programming Assumptions," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 577–591, Mar. 2015.

[AdBO10] K. Apt, F. S. de Boer, and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*. Springer Science & Business Media, Oct. 2010.

[ADK03] A. Andoni, D. Daniliuc, and S. Khurshid, "Evaluating the "Small Scope Hypothesis," Tech. Rep., 2003.

[AG96] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[AGdS20] G. A. G. Andrade, M. Graf, and L. C. V. dos Santos, "Chaining and Biasing: Test Generation Techniques for Shared-Memory Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 3, pp. 728–741, Mar. 2020.

[AMM+18] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, "Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, Mar. 2018, pp. 405–418.

[AMSS] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "ARM Litmus Tests," https://www.cl.cam.ac.uk/~pes20/arm-supplemental/.

[AMSS11] ——, "Litmus: Running Tests against Hardware," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, P. A. Abdulla and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer, 2011, pp. 41–44.

[AMSS12] ——, "Fences in weak memory models (extended version)," *Formal Methods in System Design*, vol. 40, no. 2, pp. 170–205, Apr. 2012.

[AMT14] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 2, pp. 7:1–7:74, Jul. 2014.

[ARMa] "ARM Barrier Litmus Tests and Cookbook."

[Armb] Arm, "Big.LITTLE – Arm," https://www.arm.com/why-arm/technologies/big-little.

[Armc] ——, "Cortex-A9," https://developer.arm.com/ip-products/processors/cortex-a/cortex-a9.

[AS02] A. Adir and G. Shurek, "Generating concurrent test-programs with collisions for multi-processor verification," in *Seventh IEEE International High-Level Design Validation and Test Workshop, 2002.*, Oct. 2002, pp. 77–82.

[BA08] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 68–78, Jun. 2008.

[BDW16] M. Batty, A. F. Donaldson, and J. Wickerson, "Overhauling SC atomics in C11 and OpenCL," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 634–648, Jan. 2016.

[BMO+12] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling C/C++ concurrency: From C++11 to POWER," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 509–520, Jan. 2012.

[BOS+11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, Jan. 2011, pp. 55–66.

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2011, pp. 365–376.

[EN16] M. Elver and V. Nagarajan, "McVerSi: A test generation framework for fast memory consistency verification in simulation," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 618–630.

[FGP+16] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, Jan. 2016, pp. 608–621.

[FHK+15] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar, "Advances in noise-based testing of concurrent software," *Software*

Testing, Verification and Reliability, vol. 25, no. 3, pp. 272–309, 2015.

[Gen] "Generating tests," http://diy.inria.fr/doc/gen.html.

[Han77] P. B. Hansen, *The Architecture of Concurrent Programs*. Prentice-Hall, Inc., 1977.

[HVM+04] S. Hangal, D. Vahia, C. Manovit, J.-J. Lu, and S. Narayanan, "TSOtool: A program for verifying memory systems using the memory consistency model," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, Jun. 2004, pp. 114–123.

[HWS11] M. D. Hill, D. A. Wood, and D. J. Sorin, *Synthesis Lectures on Computer Architecture: Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers., Nov. 2011.

[Imp15] "Towards Implementation and Use of memory_order_consume," Tech. Rep. WG21/P0098R0, Sep. 2015.

[ISO10] ISO/IEC, "Programming languages — C. Committee Draft," Tech. Rep. 9899:201x N1570, Dec. 2010.

[ISO12] ——, "Working Draft, Standard for Programming Language C++," Tech. Rep. N3337, Jan. 2012.

[ISO17] ——, "Working Draft, Standard for Programming Language C++," Tech. Rep. N4713, Nov. 2017.

[Lam79] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[Lea] "Learn the architecture: AArch64 memory model," https://developer.arm.com/documentation/102376/0100/Normal-memory.

[Lit] "Litmus test family names," https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test6.pdf.

[LVK+17] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in C/C++11," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 618–632.

[LWPG17] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated Synthesis of Comprehensive Memory Model Litmus Test Suites," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 661–675, Apr. 2017.

[MAM10] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Generating Litmus Tests for Contrasting Memory Consistency Models," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer, 2010, pp. 273–287.

[MMSM20] T. Melissaris, M. Markakis, K. Shaw, and M. Martonosi, "PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2020, pp. 329–341.

[MPZN13] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler testing via a theory of sound optimisations in the C11/C++11 memory model," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 187–196, Jun. 2013.

[MSS12] L. Maranget, S. Sarkar, and P. Sewell, "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models," p. 50, 2012.

[MTL+16] Y. A. Manerkar, C. Trippel, D. Lustig, M. Pellauer, and M. Martonosi, "Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings," *arXiv:1611.01507 [cs]*, Nov. 2016.

[ND16] B. Norris and B. Demsky, "A Practical Approach for Model Checking C/C++11 Code," *ACM Transactions on Programming Languages and Systems*, vol. 38, no. 3, pp. 10:1–10:51, May 2016.

[OMK+95] B. O'Krafka, S. Mandyam, J. Kreulen, R. Raghavan, A. Saha, and N. Malik, "MPTG: A portable test generator for cache-coherent multiprocessors," in *Proceedings International Phoenix Conference on Computers and Communications*, Mar. 1995, pp. 38–44.

[OPP+12] J. Oetsch, M. Prischink, J. Pührer, M. Schwengerer, and H. Tompits, "On the small-scope hypothesis for testing answer-set programs," in *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*, ser. KR'12. Rome, Italy: AAAI Press, Jun. 2012, pp. 43–53.

[Ove] "Overhauling SC atomics in C11 and OpenCL (Additional companion material)," http://multicore.doc.ic.ac.uk/overhauling/.

[QM12] X. Qin and P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '12. San Jose, CA, USA: EDA Consortium, Mar. 2012, pp. 3–8.

[Run] "Running tests with litmus7," http://diy.inria.fr/doc/litmus.html.

[SCGC19] Y. Sun, S. Cheung, S. Guo, and M. Cheng, "Disclosing and Locating Concurrency Bugs of Interrupt-Driven IoT Programs," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8945–8957, Oct. 2019.

[SMO+12] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams, "Synchronising C/C++ and POWER," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 311–322, Jun. 2012.

[SSA+] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER Multiprocesors," https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/.

[TML+17] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 119–133, Apr. 2017.

[VBC+15] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli, "Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: Association for Computing Machinery, Jan. 2015, pp. 209–220.

[WAR19] A. Waterman, K. Asanovic, and RISC-V Foundation, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA," Tech. Rep. 20191213, Dec. 2019.

[WCM+16] X. Wu, L. Chen, A. Miné, W. Dong, and J. Wang, "Static Analysis of Run-Time Errors in Interrupt-Driven Programs via Sequentialization," *ACM Transactions on Embedded Computing Systems*, vol. 15, Dec. 2016.

[YCER11] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 283–294, Jun. 2011.