Detecting Relaxed Memory Concurrency Bugs in C and C++ Compilers

Luke James Geeson

Doctor of Philosophy

Programming Principles, Logic, and Verification Group

Department of Computer Science

University College London (UCL)

March 29, 2026

I, Luke James Geeson, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

This thesis develops an automated testing framework and new testing techniques for discovering concurrency-related bugs in C and C++ compilers. Firstly, we present a technique that compares source and compiled program behaviours using source and architecture models. The Téléchat tool implements this technique and has found a number of new concurrency bugs. Secondly, we present the *Mix testing* technique, that tests the interoperability of compiler mappings. The `Atomic-mixer` tool implements mix testing, has found a number of new *mixing bugs*, and has been used to develop an *Atomics Application Binary interface* with Arm's engineers. Lastly, we deploy Téléchat in automated testing, exposing the limits of current tools and models whilst exploring the state of concurrency compilation correctness in the LLVM and GCC compiler toolchains.

Impact Statement

We summarise how the thesis has been, and could be, put to use inside and outside of academia.

Conference Publications and Workshops: Our work has been published in two international conferences. These are the IEEE/ACM *Symposium on Code Generation and Optimisation (CGO 2024)* and the ACM *Conference on Systems, Programming, Languages and Applications (OOPSLA 2024)*. We also presented at *The Future of Weak Memory Workshop* (POPL 2024), and the *Kent Concurrency Workshop 2024*.

We outline several lines of inquiry for future work in Chapter 5 that could motivate work on, for instance, scalable testing using models.

Testing Techniques, Tools, and a New Kind of Bug: We present two techniques that find bugs in C/C++ compilers. The first technique compares outcomes of executing concurrent programs under source and architecture models, finding bugs when unexpected outcomes arise. This technique is implemented in the Téléchat framework. Our approach is simpler than prior work, and is amenable to testing as understood by engineers who are not necessarily experts in concurrency. The second technique, known as *mix testing*, finds bugs by mixing implementations of atomics. Mix testing is implemented in the `atomic-mixer` tool. We identify a new class of bugs, coined *mixing bugs*, that arise only when mixing mappings from source operations to assembly sequences. John Wickerson kindly published a blog post on the topic:

`https://johnwickerson.wordpress.com/2024/06/28/mix-testing-revealing-a-new-class-of-compiler-bugs/`

Mix testing is a general idea that could be applied in domains outside of concurrency. Other groups are working on the interoperability problem as summarised in Chapter 6.

Better Oracles: Testing under source and architecture models increases the chances of observing bugs. By using models of the C/C++ language standards and processor architectures, we gain oracles that closely follow official specifications. Of course testing only shows the presence of bugs, but the task of finding bugs up to fixed bounds is now more reliable.

Testing compilers under models of source and architecture specifications brings together the fields of testing and formal modelling. We expect that testing under specifications will become more important as the demands on modern processors increases their complexity. Indeed, RISC-V maintainers [156], Arm [6], and NVIDIA [103] have already developed formal models and used them for testing. We contribute compiler testing under processor (CPU) architecture models and expect there is work to be done on GPU compilation testing in the future.

Another Angle on Testing: Mix testing exposes an uncomfortable truth. Mix testing implies we cannot test atomics mappings in isolation, but rather by strategically testing in the presence of exponentially many choices of mappings, both now and as architectures evolve. This requires a new bug class definition, practical methods to test interoperability, and specifications to bound the test surface. To quote an informal remark from an academic: The OOPSLA work is "*One of those rare papers that opens your mind to a new bug class and therefore the need for a new definition of what compilation correctness even means in this setting*".

`https://x.com/tobycmurray/status/1806679173454504251`

We expect there is work to be done on further reducing the search space, to make it feasible to deploy mix testing as part of continuous integration.

Tool Adoption: Our work was adopted and deployed by Arm's compiler teams. The Téléchat tool tests Arm's open-source and commercial compilers.

Given we support models of most mainstream processors, one could straightforwardly deploy Téléchat in testing for other architectures too.

Bugs Discovered: We found 9 bugs in the LLVM and GCC compilers, which have either been fixed, or are triaged for fixing by Arm's engineers. This compares favourably with the numbers of bugs found by prior work.

`https://lukegeeson.com/blog/2023-10-17-Telechat-Bug-Board`

After assessing the limitations of today's test generators in Chapter 5 we conclude that more work is needed on concurrency test generation to catch some corner cases in code generation we found manually.

Software Specification: We worked with Arm's engineers to publish an official *Atomics Application Binary Interface* (ABI) specification.

`https://github.com/ARM-software/abi-aa/releases/tag/2024Q3`

When we reported [50] a bug in the implementation of atomic structs, the question arose of what other architecture ABIs say about atomics. Since there were no official CPU atomics ABIs beyond what we contribute in this thesis, one could develop ABIs for each CPU architecture in the future.

Curating Expertise: We fostered a cross-group team of compiler engineers who now maintain the C/C++ atomics ABI for the Armv8 architecture, fix concurrency bugs, and answer queries regarding the compilation of atomics. We document our work in the form of internal presentations, wiki pages, artifacts, and talks at the *Arm Global Engineering Conference* and the *Arm Research Summit*.

Acknowledgements

In this section, I name those who supported me throughout this PhD but admit this list is far from exhaustive.

I'd like to thank my supervisors James Brotherston, Earl Barr, and Lee Smith for training me in the art of research, while grounding my work in its real-world implications. I'd like to thank David Clark and Peter Sewell for assessing this thesis. I thank Earl Barr, James Brotherston, and Peter O'Hearn for assessing previous vivas at UCL. I'd like to thank my co-authors James Brotherston, Alastair F. Donaldson, Wilco Dijkstra, Lee Smith, Tyler Sorensen, and John Wickerson for their advice on communicating my research. I'd like to thank numerous academics for the illuminating discussions including Mark Batty, Tobias Grosser, and Hernán Ponce de León. I'd like to thank members of the Programming Principles, Logic, and Verification Group at UCL including Byron Cook, David Pym, Maria Schett, and Alexandra Silva.

I'd like to thank my sponsor and host Arm who provided the resources that made this work possible. I'd like to thank Sam Ellis, Nick Gasson, Al Grant, Richard Grisenthwaite, Tomas Matheson, Peter Smith, Ties Stuij, Hugo Vincent, Shale Xiong, and many others at Arm. I thank Arm's Compiler Teams and Arm Architecture & Technology Group for their feedback and assistance. This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/V519625/1]. The views of the authors expressed in this thesis are not endorsed by Arm or any other company. Additionally I'd like to thank Alastair Reid, Luc Maranget, Gustavo Petri, Kyrylo Tchakov, Dan Lustig, Gonzalo Brito, Simon Cooksey, and Genevieve Breed.

Lastly I'd like to thank my friends, family, and others for their unwavering support throughout. These include Jakki, Mark, Emily, Geoff, Marley, Miren, Matt J, Rishi, Morgan, Claire, Alex, Liam, Matt E, Maria, Kevin, Sam, Bianca, Eva, Nick, Manoj, and James.

Contributions and Declarations

We summarise our contributions and provide required information for the UCL Research Paper Declaration Forms. We detail copyright declarations and attributions of authorship for published work.

Academic Contributions

Peer-Reviewed Publications (as Primary Author)

Chapter 3: Luke Geeson and Lee Smith. 2024. Compiler Testing with Relaxed Memory Models. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. pre-print: `https://lukegeeson.com/assets/publications/cgo24/paper.pdf`. IEEE Computer Society, (Mar. 2024), 334–348. ISBN: 979-8-3503-9509-9. doi:`10.1109/CGO57630.2024.10444836`

Chapter 4: Luke Geeson, James Brotherston, Wilco Dijkstra, Alastair F. Donaldson, Lee Smith, Tyler Sorensen, and John Wickerson. 2024. Mix Testing: Specifying and Testing ABI Compatibility of C/C++ Atomics Implementations. In *2024 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* number OOPSLA2. Vol. 8. pre-print: `https://lukegeeson.com/assets/publications/oopsla24/paper.pdf`. Association for Computing Machinery, 442–467. doi:`10.1145/3689727`

Reproducible Artifacts and Experiments

CGO Artifact: The artifact received the available, evaluated, and reproducible badges from the artifact evaluation committee, steps to reproduce the results

are in Appendix A. [SW] Luke Geeson and Lee Smith, CGO Artefact for Compiler Testing With Relaxed Memory Models Dec. 2023. doi:10.5281/zenodo.10204529, URL: https://zenodo.org/records/10411403

OOPSLA Artifact: The artifact received the available and functional badges from the artifact evaluation committee, steps to reproduce the results are in Appendix B. [SW] Geeson, Luke and Brotherston, James and Dijkstra, Wilco and Donaldson, Alastair F. and Smith, Lee and Sorensen, Tyler and Wickerson, John, OOPSLA Artifact for Mix Testing: Specifying and Testing ABI Compatibility Of C/C++ Atomics Implementations Dec. 2024. doi:10.5281/zenodo.12671272, URL: https://zenodo.org/records/12671272

Atomics ABI: A copy of the ABI as it was published in Q3 of 2024 is in Appendix C. Luke Geeson and Wilco Dijkstra. 2024. C/C++ Atomics Application Binary Interface Standard for the Arm® 64-bit Architecture. https://github.com/ARM-software/abi-aa/releases/tag/2024Q3. (2024)

Copyright Declarations

CGO Paper: Luke Geeson and Lee Smith are the authors of the article, IEEE permits authors to re-use all or portions of the work in other works (this thesis). IEEE owns the copyright on the article.

CGO Artifact: Téléchat consists of an External Module for use in connection with the herdtools [12] Software. Luke Geeson is the author of the Téléchat code, University College London owns the copyright. Téléchat is Licensed under CeCILL-B.

OOPSLA Paper: Luke Geeson, James Brotherston, Wilco Dijkstra, Alastair F. Donaldson, Lee Smith, Tyler Sorensen, and John Wickerson are the authors of the article. The article is licensed under Creative Commons Attribution-ShareAlike 4.0 International Licence. ACM is granted a permissive licence to publish the article. University College London owns the copyright on the material authored by Luke.

OOPSLA Artifact: The `atomic-mixer` tool consists of an External Module for use in connection with the Téléchat Software. Luke Geeson is the author of the `atomic-mixer` code, University College London owns the copyright. The `atomic-mixer` tool is Licensed under CeCILL-B.

Atomics ABI: The ABI is licensed under a Creative Commons Attribution-ShareAlike 4.0 International Licence and grant of Patent Licence. Luke Geeson authors his contributions, and University College London owns the copyright on those contributions.

Author Contribution

For all published works, where the work has multiple authors, the contributions are split as follows (using the ANSI/NISO CRediT taxonomy standard [119]):

- Luke Geeson: Conceptualisation, Data Curation, Formal Analysis, Investigation, Methodology, Software, Validation, Visualisation, Writing – original draft, and Writing – review & editing.
- James Brotherston: Supervision and Writing – review & editing.
- Wilco Dijkstra: Writing – review & editing.
- Alastair F. Donaldson: Writing – review & editing.
- Lee Smith: Supervision and Writing – review & editing.
- Tyler Sorensen: Writing – review & editing.
- John Wickerson: Writing - review & editing.

Contents

List of Figures

List of Tables

List of Abbreviations

SC Sequential Consistency . . . 24

TSO Total Store Order . . . 25

Arm ARM Arm Architecture Reference Manual . . . 27

MP Message Passing . . . 37

`X%Pn_x` symbolic `X` registers for each thread `Pn` containing location `x` . . . 37

outcome outcome of execution . . . 39

UB Undefined behaviour . . . 40

model Memory Consistency Model . . . 43

po program order . . . 43

rf reads-from . . . 43

co coherence . . . 43

fr from-read . . . 43

relaxed models Relaxed Memory Models . . . 45

AArch64 model Arm AArch64 memory model . . . 45

`atomic_int` `_Atomic`-qualified integer types . . . 47

`load` `atomic_load_explicit` . . . 47

`store` `atomic_store_explicit` . . . 47

`fence` `atomic_thread_fence` . . . 47

`sc` `memory_order_seq_cst` . . . 47

`acq` `memory_order_acquire` . . . 47

`rel` `memory_order_release` . . . 47

`acq_rel` `memory_order_acq_rel` . . . 47

`rlx` `memory_order_relaxed` . . . 47

Chapter 1

Introduction

According to a February 2024 White House report [157], *"up to 70 percent of security vulnerabilities [...] are due to memory safety issues"*. The cited source of these issues [152] are unsafe C and C++ programs. Unsafe languages leave memory management to the user, where any slip up can introduce *bugs*.

This thesis is concerned with testing C/C++ compilers for bugs. We test LLVM and GCC for bugs introduced during the translation of *concurrent* programs. Concurrent programs are temporal in nature, consisting of multiple threads that observe memory accesses made by other threads, but not necessarily in program order. As a compiler engineer at Arm, I was sitting at my desk one day, wondering why we rarely tested the compilation of concurrent C/C++, beyond rudimentary checks that the assembly code generated by compilers remains unchanged between compiler revisions.

Like many simple questions, it turns out the answer is not straightforward. As we shall see, modelling the semantics of concurrency is hard as processors can exhibit many unintuitive behaviours as a result of implementing *relaxed memory models*. Worse still, just *observing* concurrent behaviour can be hard as certain behaviours may only arise if specific conditions are met. It is unsurprising that engineers, who are not necessarily concurrency experts, have largely left the task of concurrency compilation testing to researchers.

In this chapter we motivate some problems (§1.1), provide the context of this thesis (§1.2), and summarise our contributions (§1.3).

1.1 Motivating The Problems

We focus on two problems. Firstly, we lack techniques that test concurrency compilation from source to assembly and produce the same results every time. Secondly, prior work had not considered the case of *mixing* atomic compiler mappings, which can be done by the linker.

Our desire to test compilation from source to assembly arises from the need to test Arm Compilers. Our use case is the compilation of concurrent C/C++ to Armv8 assembly, where both the source and assembly can exhibit multiple behaviours. All such behaviours must be observed if we are to catch unexpected bugs. Such observations must be *repeatable* if we are to detect bugs, or regressions, as they arise in the many compiler revisions made each day. Our desire for a technique that gives the same results every time acknowledges this fact, with a secondary benefit that such a technique may be deployed in industry regression testing. Further, compilers for mobile applications optimise programs at both compile-time and link-time to minimise the memory and runtime footprint. Our desire to test mixing arises from the observation that bugs may arise when combining programs using mappings with programs using different mappings (that are otherwise self-consistent) at link time.

This thesis presents testing techniques that find concurrency bugs in C/C++ compilers. We do so by leveraging relaxed memory models as parts of compiler testing oracles. By using models to test source and compiled program behaviour we gain techniques that can repeatedly detect bugs as they arise. We implement these techniques in tools that find numerous bugs in the LLVM and GCC compilers. These bugs have already been fixed or are triaged for fixing. We test compilation from C/C++ to multiple architectures, showing that our technique is not specific to Armv8, but is parametric in the models available. We test the interoperability of mappings, finding new mixing bugs, and we worked with Arm's engineers to develop a specification that, if implemented, outlaws such bugs from arising in the future. Lastly we deploy these techniques in industry, finding gaps where models or tools can be improved.

1.2 Some History on The Matter

Our work sits at the intersection of concurrency (§1.2.1), specification (§1.2.2), and compiler correctness (§1.2.3). Each field has a rich history, and we do not aim to cover it all, but instead focus on related works.

1.2.1 Concurrency and Relaxed Memory Models

In 1979, Lamport articulated a well-known early model [91] of concurrency that informs this thesis. The *Sequential Consistency (SC)* model describes a programmer's view in the sense that developers have an intuitive understanding of concurrency as an interleaving of multiple programs that access shared memory. The SC model is defined in terms of relations between threads that access shared memory. We illustrate the SC model in Figure 1.1.

Under SC, threads read and write to a single shared memory and the sequence of accesses is called an *execution*. A program exhibits many SC executions that are made by interleaving accesses in the program order specified by each thread. Program order is the syntactic sequencing of instructions in the program listing for each thread. In other words, the SC model constrains executions to the interleavings that obey the stated program order. Executions define the *behaviour* of the concurrent program in question under a model such as SC. Formally, executions are modelled as graphs that are consistent with the partial orders specified by the model [1]. There is a wealth of work on both memory *consistency* models and SC [3, 78]. Both the C/C++ memory model and processor memory models build on the SC model.

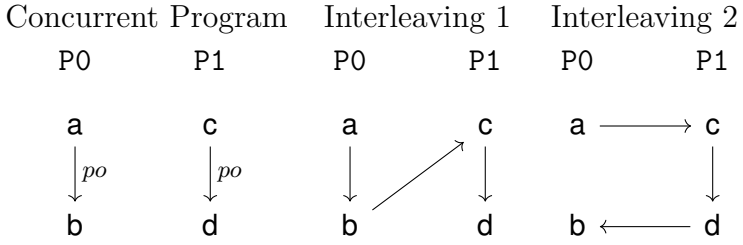

Figure 1.1: Pseudocode consisting of two threads `P0` and `P1`, each with memory accesses `a;b` and `c;d` in program order (*po*), respectively.

A *relaxed* model is one where executions do not necessarily respect program order [146]. SC is too restrictive to model the behaviour of modern multiprocessors, whose executions do not necessarily respect program order [2]. Such executions can arise due to processor pipelining, out-of-order execution, cache hierarchies, and many other reasons. We work with relaxed models.

There are two dominant formalisms for specifying relaxed models: operational and axiomatic semantics. Operational semantics define state transitions over abstractions of machine or language components, including caches and buffers [54, 22, 127]. Axiomatic models abstract the system entirely and describe relations over executions. We use axiomatic models in this work.

In the last three decades, there was an explosion of architecture models. These include models of IBM Power and Armv7 [7, 105, 135], Intel x86-TSO [143, 136, 105], Armv8 [22, 127, 6], MIPS and RISC-V [22] and many more. Each model relaxes particular constraints as permitted by a given architecture. Models are defined in terms of relations over the effects of executing assembly instructions. For example, the Intel x86 *Total Store Order (TSO)* model [143] relaxes store-to-load ordering, such that store instructions may be observed after subsequent loads issued by the same thread.

Figure 1.2 shows the languages and compilers we test in the centre. A C/C++ program is parsed and translated into LLVM or GCC's internal

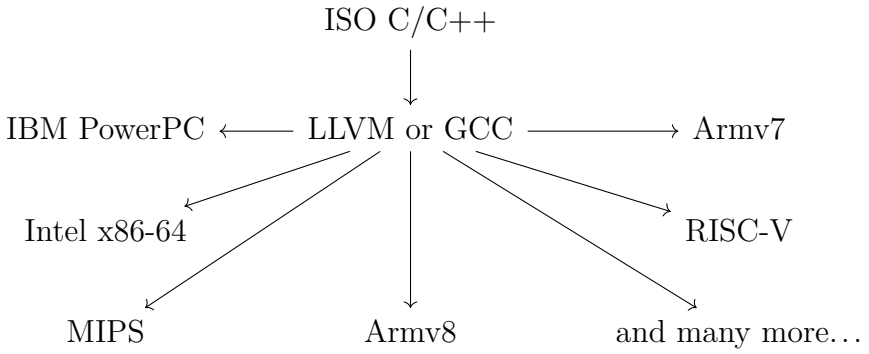

Figure 1.2: The languages and compilers we work with. Nodes are the languages and arrows are the steps where bugs may arise.

representation, where mappings and optimisations are applied, then code generators for each architecture produce the final assembly. At each stage bugs may be introduced, and compiled programs may exhibit executions under the architecture model, that do not have comparable executions in their source program under the source model. Such executions may lead to an erroneous final state, assuming the source model forbids that behaviour. We use models of assembly languages to compute compiled program behaviour.

The C and C++ models are also relaxed. The C/C++ models have evolved over time, as numerous authors have contributed to them including Boehm and Adve [36], Batty [32, 31], and others [155, 90, 28, 85, 97]. The ISO C/C++ standards [80] define their models in terms of constraints on relations over memory access events. There are also intermediate models that act as stepping stones in soundness proofs between C and architecture models, for instance the LLVM memory model [39], the intermediate [125] memory model (IMM), and its axiomatic counterpart WeakestMO [40]. We use variants of the C/C++ models [90] to compute source program behaviours.

The compiler must implement the intent of the ISO C/C++ model in terms of the available architecture instructions. These *mappings from C/C++ statements to assembly sequences (mappings)* must respect the constraints on executions given by the C/C++ model. To assist in this endeavour, new C/C++ models provide *compilation schemes* that present idealised mappings from C/C++ atomic operations to assembly sequences. Schemes such as those in Table 1.1 are unfortunately quickly outdated, since compilers implement *multiple* mappings that are routinely revised.

Atomic Operation	Assembly Sequence
`load(loc,sc)`	`LDAR W2, [loc]`
`store(loc,val,sc)`	`MOV W2, #val` `STLR W2, [loc]`

Table 1.1: Mappings from C/C++ to Armv8, from Lahav et al. [90].

We now turn to the second pillar of work upon which this thesis depends.

1.2.2 Architecture and Language Specifications

Memory models began as part of language specifications. As such there is a lot of prior work, and we do not attempt to be exhaustive. We focus on efforts to make models executable as test oracles, the tools that compute the behaviour of small programs called *litmus tests*, and tools for reasoning about larger programs. To set the scene we first look at some work that reasons about large programs before focusing in on the work closely related to this thesis.

Reid [131] presented `Archex`, a tool for running and reasoning about the executable *pseudocode* (of the whole architecture, not just the memory model) developed by Arm's architects over the years. Armstrong et al. [22] incorporated this work into the Sail [23] project, which has a tool for reasoning about pseudocode for Armv8, RISC-V, and CHERI-MIPS amongst others. Specifying architectures required monumental efforts, but led to many publications including on synthesising correct Verilog programs [132], formalising virtual memory [147], and heap temporal safety of capability architectures [53].

There are similar tools for reasoning about large C and C++ programs. Memarian et al. [114] presented the `Cerberus` tool, which enables reasoning about C and has been used to study pointer provenance [113], a memory object model [75], and for verifying hypervisors [128]. The `CBMC` [45] tool is a model-checker for C that has been extended to compute the partial orders (DPOR) of concurrent C programs [10]. Kokologiannakis and Vafeiadis have made numerous improvements to the DPOR algorithm in the `GenMC` [89] tool, which has been used to verify concurrency algorithms and libraries [88].

Work on specifying memory models happened in parallel. The C11 and C++11 language standards introduced atomic operations and with them a prose description of the C/C++ memory model. Likewise, the Arm Architecture Reference Manual (Arm ARM) defines a prose model of the Arm AArch64 architecture in terms of relations over the effects of executing assembly instructions. The relations defined by each model differ, but both models define axiomatic relations in some form. We use *executable* variants of these models.

Many works focused on making these models executable as test oracles. The `memevents` [7] tool explored program behaviour under the x86-TSO and Armv7 models as defined in the axiomatic style. Likewise `ppcmem` [136] and `ppcmem2` tools (that later evolved into `RMEM` [137]) enable the exploration of IBM Power and x86-CC program executions under operational models. The `RMEM` [22, 137, 127] tool provided the means to explore program behaviours under models of Armv8. In 2012, the `cppmem` tool [30] and its successor [34] enabled exploration of concurrent C/C++. There are of course earlier tools, such as `TSOtool` [76] and `Nemos` [163], but we focus on `cppmem` and `RMEM` as they are close predecessors of tools we use in this thesis.

The `RMEM` and `cppmem` tools use *litmus tests* to explore model behaviour. Litmus tests are small concurrent programs with a fixed initial state and a predicate over the final states. Under axiomatic models, the threads of a litmus test are run in parallel from their fixed initial state to produce a set of executions. Those executions lead to one or more *outcomes* and the predicate over the final state is used to query whether certain outcomes are observed. Litmus tests are thus a blend of syntax and semantics that is used to test whether a certain execution, and hence behaviour, is allowed. In this thesis we test compilation from C/C++ litmus tests to assembly litmus tests. We use tools to compute the behaviour of litmus tests.

The `Isla` [21], `Dartagnan` [126], and `herd` [17] tools simulate litmus test behaviour under models encoded using the Cat [5] language. The Cat language puts a syntax to memory models specified in the relational algebra style [17]. Lau [94] extended `Cerberus` to simulate concurrent C/C++ under Cat [5] models and Batty et al. [28] extended the `herd` tool to model the concurrency semantics of OpenCL. We use `herd` to simulate both C/C++ and assembly litmus tests under executable models. We do so because it hosts models of both C/C++, and the processor architectures that are targeted by compilers. Our work can however be adapted to use tools that accept litmus tests and Cat memory models, including `Dartagnan`, `CerberusBMC`, and `herd`.

In order to simulate behaviour one must have tests. The `diy` [15] tool generates litmus tests from specifications. Litmus tests are specified as cyclic executions over the relations used by a memory model. The `Memalloy` [158] improved on `diy` by framing the presence of cyclic executions as a constraint problem and using a solver to generate tests. The `Litmustestgen` and `Kater` [87] tools built on this idea to generate suites of litmus tests. We use the `diy` tool to generate C/C++ litmus tests.

The works outlined in this section treat simulators as oracles of program behaviour. We will return to the oracle problem in the context of compiler testing soon, but first we describe the third pillar that supports this thesis.

1.2.3 Compiler Correctness

The origins of compiler correctness are at least as far back as McCarthy and Painter [111], and has been considered by many authors over the years of which we mention only a few. In 1966, McCarthy and Painter [111] presented a proof of correctness for a compiler of arithmetic expressions and memory operations. In 1989, Moore [117] presented a verified compiler, whose correctness is witnessed by observing the final state of the FM8502 processor after executing a compiled program. Later, Leroy [98] presented the CompCert verified C compiler, which is a verified implementation of C built on a notion of semantic preservation between the source and compiled programs. While themes of memory operations, observing architecture state, and semantic preservation will recur in this thesis, we focus on the work that proceeds from CompCert.

Verified compilation aims to show that any behaviour of the compiled program is also permitted by the source program semantics. A *behaviour* is the result of executing a source program s from a given input i (assuming s terminates), defined as a set of assignments to shared and local data. Concurrent programs can exhibit multiple executions, and so it is useful to characterise concurrent program behaviours for the source and target as *sets* ($\mathcal{B}_{SRC}$ or $\mathcal{B}_{ARCH}$, respectively). A compiler is correct if the compiled program behaviours, denoted using the set $\mathcal{B}_{ARCH}(comp(s)(i))$, form a subset

of the permitted source program behaviours, denoted $\mathcal{B}_{SRC}(s(i))$. Of course, source and compiled program behaviours are not directly comparable, since source behaviours affect global and local variables (for instance `{ x=1, r0=2 }`), and architecture behaviours affect machine registers and shared memory locations (`{ x=1, X0=2 }`). We thus require a *state mapping* from compiled to source program state, which is constructed from compiler metadata, represented as a function $f : \mathcal{B}_{ARCH} \rightarrow \mathcal{B}_{SRC}$. Correct compilation is then defined $\forall s. \forall i. f(\mathcal{B}_{ARCH}(comp(s)(i))) \subseteq \mathcal{B}_{SRC}(s(i))$.

In 2019, Ringer [133] surveyed verification tools, highlighting work including CompCert [98] and Vellvm [164]. Ševčík developed a theory of sound optimisations [140] with the idea being that if a compiled program exhibits an execution, then the source program should also exhibit that execution. Ševčík et al. [141] extended CompCert to CompCertTSO [141], which verifies the compilation of concurrent C/C++ programs to x86 assembly programs that follow the x86-TSO model. While we do not verify compilers, it is worth noting that ideas from these works influenced the state of the art in testing and translation validation. We focus on translation validation and testing.

Translation validation (TV) [124] is an alternative method of verification that decouples the compiler from validation. TV complements a compiler with a *validator* that checks a posteriori that the output of the compiler is correct [98]. Kasampalis [86] surveys TV works, identifying how equivalence is achieved by verifying a generated *proof script* [124], symbolic evaluation [120], normalising value graphs [154], or matching path conditions through execution graphs [41]. TV is useful whenever the validator is maintained outside the compiler code base (say as a solver) or when it is critical that one particular program is verified. For instance, Sewell et al. [145] verify the seL4 kernel, and Lopes et al. [101] conduct bounded translation validation using the LLVM intermediate representation (IR). Chakraborty and Vafeiadis 2016 [41] extend Ševčík's work to TV, by matching the executions of concurrent LLVM IR programs before and after optimisation using SMT solvers to enumerate states.

Compiler testing only checks that the expected and actual output match, given an input. The input in this case is a fixed program and initial state. Chen et al. [42] survey compiler testing techniques that vary such inputs, identifying *differential* and *metamorphic* testing. These techniques, spearheaded by `CSmith` [162] and `Orion` [95], have found hundreds of sequential compiler bugs to date. Lidbury et al. [99] and Donaldson et al. [52] test the compilation of parallel GPU/OpenCL kernels, and graphics shaders but do not address concurrency-related bugs [41]. In each case, techniques such as fuzzing [115], slicing [77], and mutation [84, 122] are used to find bugs. *Fuzzers* or *slicers* apply semantic-preserving transformations to modify or reduce a program. Morisset et al. [118] adapt Ševčík's work to testing by matching executions derived from the architecture state of assembly programs, before and after optimisation. Windsor et al. 2021 [160] test the compilation of concurrent C/C++ programs by comparing final outcomes of source and compiled programs.

There is a wealth of work on ensuring source programs are correct [8, 134, 33], or improving performance by, for instance reducing caching [92]. We focus solely on testing the compilation of concurrent C/C++ programs that use atomic operations, rather than fixing or improving programs to begin with.

The test oracle problem [83, 82] affects all correctness efforts. The *test oracle problem* is the challenge of *"distinguishing the corresponding desired, correct behaviour from potentially incorrect behaviour"* [25] given an input to a system. Using the correct compilation terminology, this means that a technique must take care to compute $\mathcal{B}$ correctly, regardless of the choice of s and i.

The work in §1.2.2 treats simulators as oracles of program behaviour $\mathcal{B}$. For compiler testing we need two such oracles. One for the source, or expected program behaviour, and one for the compiled, or actual program behaviour. As such a *compiler testing oracle* can be defined in two parts. The oracle 1) checks if the behaviour of executing a compiled litmus test is 2) also a behaviour of executing the source litmus test. When this does not hold we observe a concurrency-related compiler bug.

Prior work [118, 41, 160, 159] on compiler testing for concurrency differs in how they instantiate the oracle. Prior work defines test behaviours as either executions [118, 41] or outcomes [160, 159]. As such, the behaviour comparison is either a sub-graph check [118, 41] or an outcome subset check [160, 159]. Additionally, these works differ in that they use either hardware execution [160, 159], or simulation under models [118, 41, 160, 159] (or both) to compute behaviours. We take the program behaviour to be the outcomes. This simplifying assumption reduces the problem of bug finding from one of matching graphs of executions to a simple subset check on the sets of allowed outcomes. We compute program outcomes by simulating a litmus test before and after compilation under relaxed models. This assumption pushes the the complexity of observing all concurrent program behaviour into the simulator, so that testing is reduced to a problem of test generation.

We now have everything we need to outline our contributions.

1.3 Contributions and Thesis Structure

Chapter 2, Foundations: This chapter provides the technical foundations for the rest of the thesis, including concurrent programs, compiler testing, and memory models. We also formalise our approach at the core of this thesis.

Chapter 3, and First Contribution (Observability): The state of the art compiler testing [118, 160] and TV [41] tools for concurrent C/C++ do not test compilation from source to assembly under models of both. One [160] of the techniques depends on hardware execution, and one [41] does not test compilation down to the assembly level. Hardware implementations may omit behaviours allowed by the model. Even if the hardware implements the behaviour, it may not exhibit it at runtime. Testing the IR, rather than the assembly, will miss bugs in target dependent optimisations.

We contribute a compiler testing technique that parameterises testing under source and architecture memory models to increase the likelihood of observing the concurrency-related bugs introduced by the compiler. We develop

the Téléchat compiler testing tool that implements this technique, and use it to find a number of new compiler bugs, deploy automated compiler testing of concurrent C/C++, and conduct a number of novel experiments on the GCC and LLVM compilers.

Chapter 4, and Second Contribution (Interoperability): The question of *compositional* correctness arises when combining binaries produced with multiple compilers. Compositional correctness is ensured with an *application binary interface* (ABI), which specifies interoperable compiler mappings. Processor designers publish many such ABIs, like the ABI for the Arm Architecture, but the ABI of *concurrent* programs is relatively unexplored[1]. At the time of writing, there are no official concurrency ABIs as far as we could tell. The lack of published ABIs may be due to a lack of will to engage with the ABI definition processes, or the desire not to diverge from a particular set of mappings once they are established. Indeed, the state of the art testing techniques assume the compiler is fixed such that the *whole* concurrent program is translated using one set of mappings. Compiled programs, whose assembly is constructed from *multiple* mappings, have not been tested.

We present *mix testing*: a new technique designed to find compiler bugs when the instructions of a C/C++ litmus test are separately compiled and then linked together. We have designed and implemented a tool, `atomic-mixer`, which we have used to: (a) to reproduce an existing compiler bug that arises when mixing mappings (a *mixing bug*), (b) to find previously-unknown mixing bugs in LLVM and GCC, and (c) find one prospective mixing bug in mappings proposed for the Java Virtual Machine. Lastly, we develop, jointly with engineers at Arm, an atomics ABI for Armv8, and use `atomic-mixer` to validate the LLVM and GCC compilers against it.

Chapter 5, and Third Contribution (Limitations): Since memory models are part of larger specifications there are many language features that interact with concurrency. Despite their effectiveness in the validation of concurrent

[1]Sewell [144] maintains a web page of mappings.

programs, today's models and simulators are limited in their practical applicability since the models paired with the `herd` simulator are either unsound or incomplete. Significant work was required to apply such models to the task of finding concurrency-related compiler bugs. We report on our experience deploying these models, emphasising cases where the models or `herd` misses bugs, language features, and behaviours otherwise overlooked.

We found nine bugs and while this compares favourably with the number of bugs found by prior work, it is perhaps less than one would expect from a thesis on compiler testing. We test the limits of model implementations using Téléchat. We explore examples including: a study on how concurrent programs interact with sequential undefined behaviour, how they interact with `const`-qualified atomic types, and a comparison of some C/C++ models to assess their relative strengths with respect to a corpus of tests.

Chapter 6, Related Work: We compare our work with the state of the art in compiler testing for concurrency, and other related works.

Chapter 7, Conclusions and Further Work: We summarise our contributions, explore their implications, and outline future lines of inquiry.

Artifacts and Experiment Results: We provide a number of artifacts to reproduce the results in our work. Guides on how to access and use these artifacts are included in the appendices.

Atomics ABI for the Armv8 Architecture: We provide a copy of the C/C++ Atomics Application Binary Interface Standard for the Arm® 64-bit Architecture we developed with our Arm colleagues. The document was released as an official specification in the Q3 2024 release of the Arm ABI.

Chapter 2

Foundations

In this chapter we describe concurrent programs (§2.1), memory models (§2.2), tools (§2.3), and compiler testing (§2.4). We assume the reader is aware of compilers, architectures, and concurrency, but is not necessarily an expert.

2.1 Program Syntax and Semantics

We first describe the syntax and semantics of the concurrent programs we use.

2.1.1 Concurrent Program Syntax

We define two families of languages for the source and target domains. Each family differs to account for the syntax of C, C++ and Armv8, RISC-V and so on, but all languages represent concurrent programs. The first family of high-level C-like languages represents the source languages of the compilers under test. Source programs consist of *statements* including assignments, control flow, and the sequencing of *expressions* that read or write to local or shared variables. The second family describes assembly-like languages that represent compiler target languages. Assembly programs break expressions (and hence statements) down into sequences of *instructions* with explicit control flow. Assembly instructions differ from source expressions in that they read and write to either local registers or memory locations rather than variables.

We first define the syntax of a simplified C-like language. We assume the reader is familiar with the C or C++ languages whose syntax mirrors the syntax described below. As such we do not provide a full grammar of C-like

statements but rather focus on the syntax used in this thesis. Variables are defined as an infinite set of distinct locations including *shared* variables (`x`, `y`, `z`, ...) and *local* variables (`r0`, `r1`, `r2`, ...). Likewise, values are defined as an infinite set of constants. Local variables are lexically scoped to their enclosing thread whereas shared variables are globally visible. Expressions are built from variables, values, operations, and so on, ... Statements are either assignments, compound statements, function calls, or read-modify-write statements. *Compound* statements include sequences, control-flow, and iterators, which contain nested sequences of expressions and statements. A *thread* is then one or more statements referred to by a unique thread ID (`P0`, `P1`, `P2`,...).

Definition 2.1.1. *Concurrent Program*: A *program* is a set of threads and a *concurrent* program P is a program where at least two distinct threads refer to a common shared variable. Each thread of P is a statement[1] that has a unique *thread id*. Figure 2.1 summarises the syntax.

```
Prog     := Set(Thread)                 TID      := { P0, P1, ... }

Thread   := TID () {                    Stmt     := [type] Variable = Expr
              Stmt                              |  Cmp
            }                                   |  Variable rmw-op Expr
                                                |  Variable(Expr[,Expr])
Expr     := Variable
          | Value                       Cmp      := Stmt; Stmt;
          | Variable bin-op Expr                |  if( Expr ) { Stmt }
                                                |  while ( Expr ) { Stmt }
Variable := { x, y, r0, r1, ... }
Value    := { 0, 1, 2, 3, ... }         rmw-op   := { +=, -=, ... }
bin-op   := { +, -, ... }               and so on ...
```

Figure 2.1: We formalise concurrent programs as sets of threads. For a more complete reference of C/C++ see, for example, Stroustrup [149].

Concurrent programs abstract idiomatic patterns of access to shared memory found in production software. Our testing focuses on simple idiomatic patterns of C/C++ and assembly code (of multiple architectures) for analysis.

[1]Defining the thread body as one statement implicitly encodes the AST structure of the program via sequencing, control-flow, and other compound statements.

```
P0 () {                  // The "Producer"
  *x    = 1;             // a computes some message
  *flag = 1;             // b signals P1 that message is ready
}
P1 () {                  // The "Consumer"
  int r0 = *flag;        // c checks for a ready flag
  int r1 = *x;           // d before reading the new message
}
```

Figure 2.2: Message Passing Program.

Example 2.1.1. Consider the C program in Figure 2.2, which describes the *Message Passing (MP)* idiom. Message passing describes a system of communicating data between multiple threads. There are two threads `P0` and `P1`, shared variables `x` and `flag`, and locals `r0` and `r1`. Each thread `P0` and `P1` consists of sequences of stores and loads to `x` and `flag`, respectively. Thread `P0` proceeds by `a`) writing to some shared location `x` `b`) setting a `flag` to indicate `x` is ready to be read. In parallel, `P1` `c`) reads the `flag` and then `d`) reads `x`.

We now describe the syntax of assembly-like languages, using AArch64 assembly as a concrete instance.

AArch64 assembly (A64) syntax: Arm assembly programs use a mix of registers (`W0`, `X0`, `W1`, ...) throughout this work. Likewise, constants use `#`. The `MOV`, `LDR`, and `STR` instructions take a comma-separated list of registers as input where the first register is typically the *destination* register into which data is loaded, or the *source* register from which data is written. In either case the *address* register in brackets `[]` holds the location of data being accessed.

Example 2.1.2. The Arm AArch64 assembly program in Figure 2.3 has two threads (`P0` and `P1`) lined up in columns, where each column is a sequence of A64 assembly instructions. Each column consists of sequences of `MOV`, `LDR`, and `STR` instructions, which access shared locations `x` and `flag` stored in the local `X`-registers on each thread. For tooling convenience reasons we use symbolic `X` registers for each thread `Pn` containing location `x` (`X%Pn_x`). Assembly programs for other assembly languages have a similar shape but use a different instruction syntax. We define the syntax of such assembly programs as they arise.

```
P0                    | P1
MOV W0,#1             | LDR W1,[X%P1_flag]
STR W0,[X%P0_x]       | LDR W3,[X%P1_x]
MOV W2,#1             |
STR W2,[X%P0_flag]    |
```

Figure 2.3: Arm Assembly MP program. For a more complete reference of AArch64 instructions, see Arm Limited [18].

2.1.2 Concurrent Program Semantics

The semantics of a program is defined in terms of *events*. An *event* is a tuple (l:A[loc]=val) that characterises an access (A=R (a read), W (a write), or RMW (a read-modify-write)) to a location (loc), with a particular value (val). Events are also given labels (l=a, b,...). For example, the statement `x=1` gives rise to the event a:W[x]=1, which describes a write of value `1` to the location `x`. Compound statements are described by sets of events.

Composing events can induce relations between them. Intuitively, some events *communicate* through shared memory, influencing other events. Other events simply arise in *program order* (*po*) in a given thread. Some events are thus *related*. For instance, in the sequence `x=1; r0=x` the read event b:R[x]=1 of the statement `r0=x` *reads from* (*rf*) the write event a:W[x]=1 of the statement `x=1`. Of course not all events are related, for instance in a *parallel* program that consists of multiple threads where each thread accesses distinct memory. An *execution* is thus a partial order over the events of a program. For tooling convenience reasons, executions are bounded by fixed parameters on loop unrolling, function calls, and recursion unless otherwise stated.

Definition 2.1.2. *Execution* [71]: Given a program P and the set of P's events $E =$ `{ a, b, c, d }` as derived from the semantics of the program, an execution is a partial order (denoted for example abcd) over E.

A concurrent program may exhibit multiple executions owing to the non-deterministic nature of executions involving unrelated events and the order in which threads proceed. When a read event occurs before any write to the same location in an execution, it reads from the *initial state*. A *state* is a function

from local and shared variables to values. We define a convenient set-like syntax `{ x=1, P0:r0=0 }` to represent states. The syntax `P0:r0` means the local variable `r0` owned by thread `P0`. We drop thread identifiers (`P0`) when the variable `r0` is clear from the context of its use. An *initial* state is a special state that declares and initialises variables as a set of write events to complete reads-from. By convention, we zero-initialise locations that are not defined.

Example 2.1.3. Figure 2.4 shows four possible executions produced by running Figure 2.2 given the initial state `{ x=0, flag=0 }`. The execution cdab arises when running `P1` followed by `P0`. The execution dabc arises if `P1` reads `x` *first*, then runs `P0`, followed finally by a late read of `flag` on `P1`.

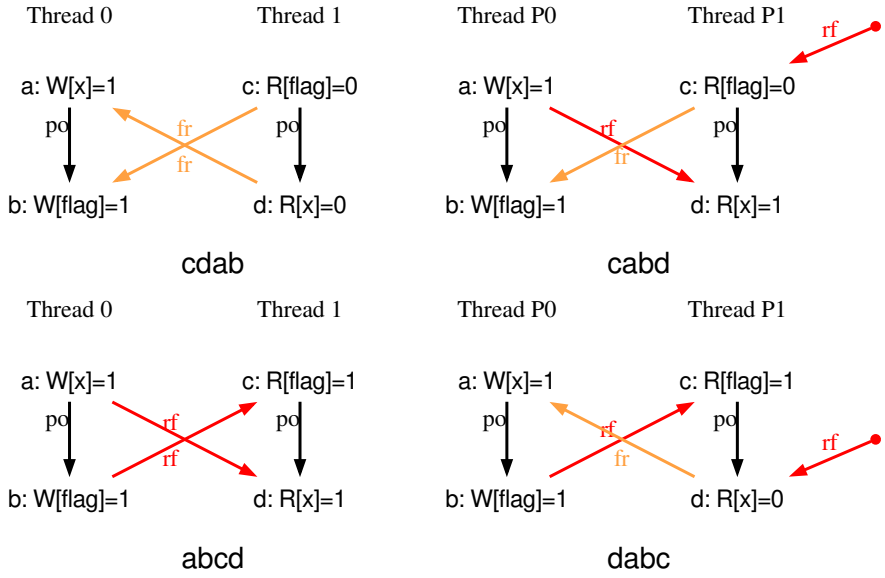

Figure 2.4: Message Passing Executions. For now *fr* is undefined.

The state of the program at the end of an execution is known as an *outcome of execution (outcome)*. A program that exhibits multiple executions beginning from some fixed initial state may also exhibit multiple distinct outcomes.

Definition 2.1.3. *Outcome* [71]: Given a concurrent program s and fixed initial state i, an outcome is a final state reached by an execution of $s(i)$. Outcomes are states expressed as assignments to shared memory locations (e.g. `{ y=2 }`) and thread-local data (e.g. `{ P1:r0=1 }`) to values.

Example 2.1.4. Consider Figure 2.2 once more. Given the initial state `{ x=0, flag=0 }`, we execute `P0` and `P1` in parallel, and get back an outcome. If the outcome `{ P1:r0=1, P1:r1=0 }` arises, then there is an execution (dabc) where the location `x` is read before the `flag` is set. In other words, thread `P1` has read the old value `0` from the initial state of `x`. This outcome may be undesirable, but at this point we make no statement about which outcomes are allowed.

Example 2.1.5. Consider Figure 2.5, which shows the outcomes of executing Figure 2.2 from the initial state `{ x=0, flag=0 }`. The outcome `{ P1:r0=1, P1:r1=0 }` is marked with `!!` to indicate a late read of `flag`.

```
  { P1:r0=0, P1:r1=0 }
  { P1:r0=0, P1:r1=1 }
!!{ P1:r0=1, P1:r1=0 }!!
  { P1:r0=1, P1:r1=1 }
```

Figure 2.5: Message Passing Outcomes. Shared locations `x` and `flag` are always `1`, so MP checks the locals `r0` and `r1` instead.

A program's executions can also be *unconstrained*. So far we have seen that related events can constrain the order and contents of events. For instance the reads-from relation requires two events that have the same location and value. A program may exhibit executions which have no such constraints. For instance the C/C++ standards define [80] the notion of *Undefined behaviour (UB)* which permits anything to happen. We consider the outcomes of a program's executions (unconstrained or otherwise) from some initial state to be its *behaviour* $\mathcal{B}$.

To summarise, given a concurrent program, we derive the semantics of events from the expressions, statements, or instructions on each thread. We then construct partial orders of those events to form executions. Connecting those executions to a fixed initial state produces a set of outcomes. The set of outcomes defines the program's *behaviour* $\mathcal{B}$. This holds regardless of the choice of source or assembly language. The process of constructing executions to produce a set of outcomes is known as *shared memory concurrency*.

So far we have covered program syntax, initial states, executions, and outcomes. We have made no judgement of whether a program's behaviour is *interesting* (good or bad) or observed at all. For this we use litmus tests.

2.1.3 Litmus Test Syntax and Semantics

A litmus test consists of an initial state, a program, and a predicate over the final state. Litmus tests are used to ask *if this program is run from the initial state, do we observe a particular outcome?* Following the conventions of the state of the art, we use litmus tests in the following chapters for their concise notation.

We define a *predicate over the final state* as a predicate logic formula with an outermost universal (`forall`) or existential (`exists`) quantifier and an inner propositional formula over states (`P1:r0 = 1 /\ P1:r1 = 0`). The predicate is a formula over final outcomes that uses standard connectives and grouping rules for conjunction (`/\`), disjunction (`\/`), and negation (`not (...)`).

Definition 2.1.4. *Litmus Test*: A *litmus test* is a labelled record {`init:`I`, prog:`P`, pred:`F}, where P is a program (named `prog`), I is a fixed initial state for P (named `init`), and F is a final state predicate for P (`pred`).

We follow the convention of specifying exactly one initial state and using predicates to check the reachability of 'bad' final states. Historically, litmus tests were used to understand (and model) the concurrent behaviour of either hardware, a programming language, or a system. The predicate over the final state is typically used to *test* for forbidden or undesirable behaviour. For instance, it is reasonable that a message passing system should forbid the read of `flag` *after* `x` and the outcome `{ P1:r0=1, P1:r1=0 }`. This notion of a test can be easily conflated with our test for concurrency-related compiler bugs later, but it is instructive to describe the precedent behind litmus tests.

Example 2.1.6. Figure 2.6 shows Figure 2.2 as a litmus test. The `exists` clause in the predicate over the final states returns true if the specified final state `{ P1:r0=1, P1:r1=0 }` is reachable from the fixed initial state. We do

```
{ *x = 0, *flag = 0 }

P0 () {
  *x    = 1;
  *flag = 1;
}
P1 () {
  int r0 = *flag;
  int r1 = *x;
}

// Predicate over the final states
exists (P1:r0 = 1 /\ P1:r1 = 0)
```

Figure 2.6: Message Passing Litmus Test.

not need predicates for compiler testing, but it is instructive to use them to restrict focus onto potentially 'buggy' states introduced by the compiler. We are now equipped to explore models of concurrent systems.

2.2 Memory Consistency Models

Figure 2.5 highlights that concurrent programs can exhibit multiple outcomes that are unintuitive. We must address the *test oracle problem*, which broadly covers the challenge of *"distinguishing the corresponding desired, correct behaviour from potentially incorrect behaviour"* [25] given an input to a system. A number of questions follow, including what behaviour do we *expect* a system to exhibit? and what behaviour does the system *actually* exhibit? These questions are the purview of memory consistency models.

Models can help distinguish the right behaviours from wrong. Many outcomes can be described by interleaving the events on each thread as they appear in program order. However, multicore processors can execute the instructions on each thread *out-of-order*. Such executions influence the execution of other threads through shared memory and increase the possible behaviours a program may exhibit. In light of such behaviours it is not trivial for an engineer who is not necessarily a concurrency expert to determine whether a behaviour is correct. Given our motivations stem from the desire to test concurrency compilation for an engineering team, we use models as oracles in this thesis.

A *Memory Consistency Model (model)* determines the allowed behaviour of concurrent programs by constraining executions to filter out forbidden outcomes. Both C/C++ and the various architectures have models that must be respected by the compiler. Running a concurrent program s from a fixed initial state i under a model $\mathcal{M}$ or a machine implementation of that model should produce a set of outcomes allowed by $\mathcal{M}$, denoted $\mathcal{B} = \mathit{outcomes}(s(i), \mathcal{M})$. Constraints assert requirements on the relations, and while most models have different relations, there are four relations that they all share.

Definition 2.2.1. Relations Between Events [142]

Program Order ($\xrightarrow{\text{po}}$): *program order (po)* is an irreflexive and transitive total order on events with the same thread ID.

Reads-from ($\xrightarrow{\text{rf}}$): *reads-from (rf)* relates writes and reads with the same location and value where there is at most one rf edge for each read[2].

Coherence ($\xrightarrow{\text{co}}$): *coherence (co)* is an irreflexive and transitive total order over writes to the same address.

From-read: ($\xrightarrow{\text{fr}}$): *from-read (fr)* is a derived relation that orders reads between writes in coherence order.

More Derived Relations: The Cat [5] language derives more events and relations (such as `sm`, `fre`, and `coe`) from the base relations `po`, `rf`, and `co` where convenient. We will cover derived concepts as we need them. For now, `e` annotations constrain relations so that their events are on different threads. So `fre`, or *from-read external* means that an event reads from a write on a different thread.

Example 2.2.1. Consider the executions in Figure 2.4. Execution `cdab` occurs since `c` $\xrightarrow{\text{po}}$ `d` and `a` $\xrightarrow{\text{po}}$ `b`, but also `d` $\xrightarrow{\text{fr}}$ `a`. Connecting `cdab` to the initial state `{ x=0, flag=0 }` gives rise to the final outcome `{ P1:r0=0, P1:r1=0 }`. This execution effectively *interleaves* the events on each thread.

[2]We assume the initial state is a set of write events, simplifying this definition somewhat.

In this thesis we use axiomatic models defined in the relational algebra style [17]. Axiomatic models define and constrain relations over events. We define constraints and models, and illustrate these ideas using Lamport's [91] model of *Sequential Consistency* (SC).

Definition 2.2.2. *Constraint*: A constraint is a predicate over relations. The predicate asserts that relations are either `acyclic`, `empty`, or `irreflexive`. Relations (sets of pairs) are composed using set theory union (`|`), intersection (`&`), complement (`~`), difference (`\`), and sequencing (`;`) operators with lexical scoping `()` rules. Constraints are named using the `as` construct.

Definition 2.2.3. *Axiomatic Memory Model*: Given a program s written in a language *SRC*, a *model* $\mathcal{M}_{SRC}$ is a set of constraints that determines whether an execution of s is allowed. An execution is *consistent* if the model's constraints (defined in terms of the relations *po*, *rf*, and *co*) hold. Models in this style constrain the behaviour of s, leaving only the consistent executions.

Example 2.2.2. Figure 2.7 shows the SC model [107], expressed using the Cat model language, and the outcomes of Figure 2.2 allowed by SC. The SC model assumes there is a single coherent memory (`sm`) space and is known as the *interleaving* model, which allows any execution whose events appear in the order specified by each thread of execution. In other words, the SC model forbids executions that do not respect program order. The cyclic execution dabc, in Figure 2.4 and outcome `{ P1:r0=1, P1:r1=0 }` are forbidden, since they can only occur if the read of `flag` happens after the read of `x`, which opposes the model's `acyclic sc` constraint in Cat.

```
(* Atomic - we describe this later *)
empty rmw & (fre;coe) as atom

(* Sequential consistency *)
acyclic po | ((fr | rf | co);sm) as sc
```

```
{ P1:r0=0, P1:r1=0 }
{ P1:r0=0, P1:r1=1 }
{ P1:r0=1, P1:r1=1 }
```

Figure 2.7: (left) SC model, (right) outcomes of Figure 2.2 under SC.

2.2.1 Architecture Memory Models

Unfortunately, the SC model is too restrictive to model modern multicore processors. Most processors exhibit reordering behaviours due to design choices made in the pursuit of performance and parallelism. These choices include but are not limited to instruction pipelining, out-of-order execution, and using multiple caches organised in memory hierarchies. Compiling and running Figure 2.2 on such a machine would risk the undesirable outcome `{ P1:r0=1, P1:r1=0 }`. Multicore processors effectively relax the constraints of SC, leading to *Relaxed Memory Models (relaxed models)*. We illustrate these ideas using the Arm assembly program in Figure 2.8.

Example 2.2.3. Figure 2.8 shows an Arm AArch64 (64-bit) assembly litmus test, and its outcomes allowed by the official Arm AArch64 memory model (AArch64 model) [19]. The statement `*x = 1` from Figure 2.2 is compiled to a move (`MOV`) instruction that puts `1` into the 32-bit register `W0`, and a store (`STR`) instruction that writes the contents of `W0` to the location in the 64-bit symbolic register `X%P1_x` (ie it writes to `x`). The statement `int r0 = *flag` is compiled to a load (`LDR`) instruction, and the outcomes check the contents of local registers `W1` and `W3` of `P1`.

Definition 2.2.4. *Relaxed Memory Model*: A relaxed memory model (or relaxed model) is a memory consistency model that relaxes the constraints of the SC model [91]. Relaxations include the reordering of stores, loads, fences, and read-modify-write operations on each thread.

```
{ *x = 0, *flag = 0 }

P0                    | P1                       { P1:W1=0, P1:W3=0 }
MOV W0,#1             | LDR W1,[X%P1_flag]       { P1:W1=0, P1:W3=1 }
STR W0,[X%P0_x]       | LDR W3,[X%P1_x]          { P1:W1=1, P1:W3=0 }
MOV W2,#1             |                          { P1:W1=1, P1:W3=1 }
STR W2,[X%P0_flag]    |

exists (P1:W1=1 /\ P1:W3=0)
```

Figure 2.8: (left) Arm Assembly MP litmus test (right) Outcomes allowed under AArch64 model [19]. Note the model allows `{ P1:W1=1, P1:W3=0 }`.

```
{ *x = 0, *flag = 0 }

P0                    |P1
MOV  W0,#1            |LDAR W1,[X%P1_flag]    { P1:W1=0, P1:W3=0 }
STLR W0,[X%P0_x]      |LDAR W3,[X%P1_x]       { P1:W1=0, P1:W3=1 }
MOV  W2,#1            |                       { P1:W1=1, P1:W3=1 }
STLR W2,[X%P0_flag]   |

exists (P1:W1=1 /\ P1:W3=0)
```

Figure 2.9: (left) Arm Assembly MP litmus test with Release-Acquire Instructions (right) Outcomes allowed under AArch64 model [19].

There are relaxed models of multiple languages including C/C++ [46, 26, 27, 90, 28, 31], Armv8 AArch64 [19, 127], Armv7 [11, 7], RISC-V [106], Intel x86-64 [110, 136], MIPS [109], IBM PowerPC [108, 7, 135], NVIDIA PTX [103, 102], and more. Each language provides instructions furnished with special ordering semantics that prevent the reordering of events. These instructions are used to *synchronize* accesses to data, preventing for instance the undesirable outcome `{ P1:W1=1, P1:W3=0 }`.

AArch64 assembly semantics: Certain Arm assembly instructions exhibit *acquire* or *release* semantics [20]. Load instructions with acquire semantics ensure that all memory access events after the load are observed after executing the load. Dually, store-release instructions ensure all access events before the store are observed before executing the store. There are also fence instructions (such as `DMB ISH`) that offer a stronger synchronisation method, and are used by architectures where release-acquire instructions are insufficient. Lastly there are instructions with more relaxed (such as `LDAPR`) or no ordering semantics (`LDR`, `MOV`, or `STR`). We refer the reader to the Arm ARM [18] for more information.

Example 2.2.4. Consider Figure 2.9 that shows Figure 2.8, but using load-acquire (`LDAR`) and store-release (`STLR`) instructions. These instructions prevent the outcome `{ P1:W1=1, P1:W3=0 }` when running the test under the AArch64 model. Alternatively, placing a `DMB ISH` fence in between each pair of `LDR` and `STR` instructions in Figure 2.8 prevents the outcome.

We now explore the source C/C++ memory models.

Memory Order Parameter (abbrev.)	Meaning
`memory_order_seq_cst` (`sc`)	SC semantics
`memory_order_acquire` (`acq`)	acquire semantics per location
`memory_order_release` (`rel`)	release semantics per location
`memory_order_acq_rel` (`acq_rel`)	acquire and release per location
`memory_order_relaxed` (`rlx`)	no ordering semantics
`memory_order_consume` (`con`)	consume semantics

Table 2.1: Memory order parameters and abbreviations used throughout.

2.2.2 The C/C++ Memory Model

Not wanting to outlaw these commonplace behaviours, the C/C++ model is also relaxed. We summarise C/C++ model features relevant to this thesis.

Atomic syntax: Since C11, C/C++ has supported the `_Atomic`-qualified integer types (`atomic_int`) of various sizes or sign, and their associated atomic operations. These operations include `atomic_load_explicit` (`load`), `atomic_store_explicit` (`store`), and `atomic_thread_fence` (`fence`). Each operation takes a memory order parameter that determines its ordering semantics. Table 2.1 details the C11 memory order syntax and their semantics.

Atomic semantics: *Atomicity* ensures that an operation's accesses execute to completion without being interrupted by another thread. This means there are no executions where a load or store can *tear* [55] and another thread interleaves an access, leading to nonsense or partial data. Assembly instructions also support atomicity typically by ensuring instructions are naturally aligned whilst accessing normal memory (each language or architecture has its own rules). Natural alignment ensures that data is loaded on a single cache line, avoiding the need for two loads that could be interrupted. With these concepts we can rewrite Figure 2.2 to use C/C++ atomic operations, finally outlawing the outcome `{ P1:r0=1, P1:r1=0 }`.

Example 2.2.5. The MP litmus test in Figure 2.10 uses C/C++ atomic operations. The types of `x` and `flag` are declared atomic in the initial state, and atomic operations are used in the program. The atomic `load` operations have acquire ordering semantics and the `stores` have release ordering semantics.

```
{ atomic_int *x, *flag = 0 }

P0 () {
  store(x, 1, rel);
  store(flag, 1, rel);
}
P1 () {
  int r0 = load(flag, acq);
  int r1 = load(x, acq);
}

exists (P1:r0 = 1 /\ P1:r1 = 0)
```

```
{ P1:r0=0, P1:r1=0 }
{ P1:r0=0, P1:r1=1 }
{ P1:r0=1, P1:r1=1 }
```

Figure 2.10: (left) MP litmus test rewritten with C/C++ Atomic Operations. (right) outcomes under C/C++ model.

The outcome `{ P1:r0=1, P1:r1=0 }` is forbidden by the C/C++ model and compiling this example (targeting `"-march=armv8-a"`) leads to Figure 2.9.

Non-atomic semantics: Not all operations have ordering or atomicity guarantees. The C/C++ languages existed long before atomics were bolted on, and regular operations such as those in Figure 2.2 are typically *non-atomic*. Non-atomic operations have no ordering semantics (they can reorder) and no atomicity semantics (they can tear). Non-atomics introduce a new form of *UB* in concurrent contexts, called a *data race (race)*. A data race can occur if a concurrent program has two conflicting accesses to a location, where one access is non-atomic, and neither access happens before the other [80]. Two accesses *conflict* if one access is a write and the other is a read or write [80]. If a program exhibits any form of UB then the C/C++ language standards provide no guarantees. In other words anything can occur during execution. In practice, language implementations (the compiler) may emit assembly instruction counterparts to non-atomic operations, including plain `LDR` or `STR` instructions. Concurrent programs are *well-defined* if they contain no UB.

Read-modify-write semantics: Lastly, there are atomic *Read-modify-write (RMW)* operations. RMWs are statements that access data multiple times in one event. Much like the non-atomic increment (`x+=1`), atomic RMW operations may read `x`, optionally operate on them, and write back the resulting value.

Unlike the increment operator, RMW operations have atomicity and ordering semantics. Extra care must be taken when compiling atomic RMW operations. The SC model, and every C/C++ model since, forbids any `rmw` event that reads-from external (`rfe`) locations (writes from different threads) followed by intervening externally coherent writes (`coe`), and the RMW's write back. The `atom` constraint in Figure 2.7 ensures that atomic RMW operations do not tear. It is also worth noting that some architectures do not have atomic RMW assembly instructions. Compilers implementing C/C++ RMW operations may use sequences of instructions for such architectures. Competing threads may interleave with such RMW sequences, and so proper checks and memory order semantics must be implemented by the compiler to prevent reordering.

Example 2.2.6. Figure 2.11 shows an MP litmus test. Figure 2.11 is the same as Figure 2.10 except it uses an `atomic_fetch_sub_explicit` (`fetch_sub`) operation in place of the first load on `P1`. The `fetch_sub` reads `flag`, storing the value into `r0`, then it subtracts the value `1` from the `flag`, and writes that value back to memory. In this example the acquire-release semantics are used to ensure `flag` is written before `x` on `P1`, and `flag` acts as a binary semaphore on `x`. This means `P0` sets `flag` to indicate `x` is ready for accessing, and `P1` loads and decrements `flag`, indicating it is about to read `x`.

```
{ atomic_int *x, *flag = 0 }

P0 () {
  store(x, 1, rel);
  store(flag, 1, rel);
}                                          { P1:r0=0, P1:r1=0 }
P1 () {                                    { P1:r0=0, P1:r1=1 }
  int r0 = fetch_sub(flag, 1, acq);        { P1:r0=1, P1:r1=1 }
  if (r0 == 1) {
    int r1 = load(x, acq);
  }
}

exists (P1:r0 = 1 /\ P1:r1 = 0)
```

Figure 2.11: (left) MP litmus test rewritten with Atomic RMW Operations. (right) outcomes under C/C++ model.

This example does not forbid intervening writes to `x`. It is possible that another thread may write to `x` after the successful read of `flag`, but before the read of `x` on `P1`. A more realistic example might repeatedly poll on atomic RMWs until ready to implement mutual exclusion algorithms that ensure safety and fairness properties for concurrent programs.

We have now covered litmus test syntax, semantics, and the models that constrain their behaviour. We now turn to the techniques that generate tests, run tests on hardware, or simulate test behaviour under models.

2.3 Test Generation, Execution, and Simulation

We summarise the tools used by the state of the art and direct the reader to Chapter 6 where we survey recent works.

2.3.1 Litmus Test Generation

Litmus tests are either constructed manually, with a directed specification, or by undirected exploration. Manually constructed tests typically require concurrency experts to examine the system under test and create a test stimulus to reproduce a behaviour. When a model exists, tests may be generated by specifying executions that probe a model's constraints. Such specifications direct test generation down individual paths. We used the `diy` tool [15] to generate numerous C/C++ and AArch64 assembly litmus tests in this chapter. The `diy` tool takes a specification of a cyclic execution as input and produces a litmus test checking whether that cycle is allowed. Litmus tests and their predicates are generated using the algorithm in §4.6.1 of Alglave et al. [15].

Example 2.3.1. Consider Figure 2.12 that shows the execution dabc from Figure 2.4 specified as a string. Feeding the string `"PodWW Rfe PodRR Fre"` to `diy` generates the C/C++ litmus test in Figure 2.6. The string is a cyclic MP specification, defined as an explicit sequence of reads and writes connected by communicating relations between threads. Since this execution has a race

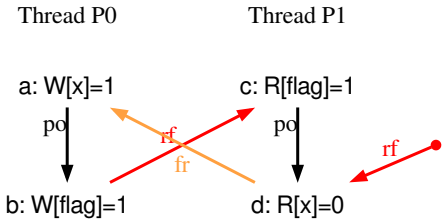

Figure 2.12: Generating the Figure 2.2 litmus test from a cyclic specification.

the program has UB. Cycles in UB-free executions are however forbidden and may induce concurrency bugs if exhibited after compilation. For example, to generate Figure 2.10, we specify: `"PodWW Rel Rfe Acq PodRR Acq Fre Rel"`, which adds acquire (`Acq`) and release (`Rel`) annotations to the edges of the previous specification. This cycle is forbidden by the C/C++ model and is a concurrency-related compiler bug (concurrency bug) if exhibited after compilation.

2.3.2 Hardware Execution and Program Simulation

Litmus tests need to be run to observe their behaviours. The most straightforward execution strategy is to compile and run tests on hardware. Concurrent programs must be run many times on a particular piece of hardware to collect its behaviours. One of the state of the art tools uses hardware to this end.

Example 2.3.2. Table 2.2 which the outcomes of running Figure 2.9 on an Apple M1 machine 100,000,000 times. Outcomes 1 and 3 are exhibited just under 50,000,000 times each, whereas outcome 2 arises just over 201,000 times. We used the `litmus` tool [16] to collect these outcomes. The `litmus` tool takes an assembly litmus test, embeds it in a test rig, runs it on the underlying machine, and collects the resulting outcomes. Hardware executions are influenced by the runtime state of the test environment including the hardware, the operating system, and flags passed to the tools. As such hardware execution environments must be stressed in order to increase the chances of observing all behaviour implemented by that hardware.

Of course, C/C++ litmus tests cannot run directly on hardware. Such

#	Outcome	Frequency
1	`{ P1:W1=0, P1:W3=0 }`	49,999,646
2	`{ P1:W1=0, P1:W3=1 }`	201,469
3	`{ P1:W1=1, P1:W3=1 }`	49,798,885

Table 2.2: Frequency of outcomes when running Figure 2.9 on an Apple M1 machine (8 cores, 8 GB memory), it takes around 17 seconds.

litmus tests must either be compiled and run, which precludes testing under the C/C++ model, or they can be run in a simulator. A simulator takes a litmus test and a memory model as input, computing the allowed outcomes of executing that test under the model. Simulators may be designed for a particular model, such as C/C++ or Arm AArch64, or they may be parameterised over models. We use the `herd` simulator [17] to compute the executions and outcomes in this thesis, and we highlight alternative tools in related work. Given a litmus test s and model $\mathcal{M}_S$ the `herd` simulator proceeds as follows:

1. Parse the litmus test s and compute the set of events E.
2. Compute the base relations B (po, rf, and co) between the events in E.
3. Compute the derived relations D using B, as specified in $\mathcal{M}_S$.
4. Compute the set of executions X using B and D when running s from its fixed initial state.
5. Filter X according to the constraints specified by $\mathcal{M}_S$.
6. Compute the set of outcomes $\mathcal{B}_S = \mathit{outcomes}(s(i), \mathcal{M}_S)$ using the predicate over the final state of s.
7. Print the outcomes and warnings if a program has concerning behaviour such as data races.

Simulators are not subject to the test environment of the underlying machine, but are slower as they lack hardware acceleration and are subject to the complexity [38] of generating executions.

In summary, when you have a model it can serve as the basis of specification, simulation, and generation. When you do not have a model, or the model is untrustworthy, you can use hardware execution to generate behaviours.

2.4 Compiler Testing

We focus on finding bugs that arise when compiling C/C++ programs to target processor (CPU) architectures. While the principles below are general, we do not test GPU compilation (see Chapter 6). In this thesis we test the `clang/gcc` compilers used to invoke the LLVM/GCC toolchains. We input a syntactically valid C/C++ concurrent program s, and a fixed initial state i to a compiler *comp* and observe its response. A compiler will:

- Crash due to an internal compiler error, such as a segmentation fault.
- Report a static error in the source program.
- Produce an assembly program whose behaviour is unconstrained, because the C/C++ source program has undefined-behaviour.
- Produce an assembly program that must be checked for bad behaviour.

For concurrency compilation testing we require that *every concurrent behaviour exhibited by the compiled program must also be a behaviour exhibited by the source program.* If the compiled program behaviours are represented by some set A and the source behaviours by some set S, then this requirement is described by a subset: $A \subseteq S$. When this requirement does not hold ($\not\subseteq$) there is a concurrency-related compiler bug. Of course the compiled program behaviours have a different type to source program behaviours, and so we also require some state *mapping* function from the state in A to the state in S. With these components we can formalise a concurrency test activity and concurrency test oracle using the notation from Barr et al. [25].

Definition 2.4.1. *Concurrency compilation test activity*: Let s be a well-defined *concurrent* source program, i be its fixed initial state, $a = comp(s)$ be

the compiled program produced by translating s with some compiler *comp*. Let S be the set of concurrent behaviours exhibited by s when run from the initial state i and A be the set of concurrent behaviours exhibited by a when also run from i. Then a concurrency compilation test activity is given by the stimulus-response pair and the testing requirement $(s, a) : [f(a) \subseteq s]$. Repeating this test activity for multiple source tests (a test suite) describes compiler testing for concurrent programs.

Definition 2.4.2. *Compiler testing oracle for concurrency* is a partial function from a concurrency compilation test activity sequence to true or false. If the conjunction of all test activities in the sequence is true in the oracle's test environment, then testing under that sequence reveals no concurrency related compiler bugs. The test environment computes the program behaviour before and after compilation.

Prior work differs in behaviours, requirements, and the test oracle. Firstly, prior work defines test behaviours as either executions [118, 41] or outcomes [160, 159]. Secondly, these works define the testing requirement in terms of a subgraph check [118, 41] or an outcome subset check [160, 159]. Thirdly, these works use either hardware execution [160, 159], or simulation under models [118, 41, 160, 159] (or both) to compute behaviours. In this thesis we define behaviours as outcomes that are compared using a subset check and produced by simulation under source and target models. These concepts are at the heart of this thesis.

2.4.1 Our Approach to Finding Bugs

We are looking for behaviours in the form of either $comp(s)(i)$ crashing at runtime or exhibiting buggy program outcomes in a test environment. The test environment is a simulator running a source or target memory model. Our oracle compares the outcomes of running a litmus test before and after compilation.

We test compilation from source languages to assembly languages under memory models $\mathcal{M}_{SRC}$ and $\mathcal{M}_{ARCH}$, respectively. A compiler bug arises if,

for any syntactically valid source program s that does not exhibit undefined behaviour in the source semantics, any outcome $b \in \mathcal{B}_{ARCH}$ of a compiled program $comp(s)(i)$ is not a outcome $\mathcal{B}_{SRC}$ of the source program $s(i)$. Of course we cannot directly compare $\mathcal{B}_{ARCH}$ with $\mathcal{B}_{SRC}$ since $\mathcal{B}_{ARCH}$ represents *machine* outcomes whereas $\mathcal{B}_{SRC}$ represents *source* program outcomes. We therefore require a state mapping function $f : \mathcal{B}_{ARCH} \to \mathcal{B}_{SRC}$ from source variables to target registers[3], derived from the debug data emitted by the compiler. We can now define a bug:

Definition 2.4.3. *Concurrency-related compiler bug*: Let s be a well-defined *concurrent* source program, i be its fixed initial state, and f be a mapping from machine to source states. Let $\mathcal{M}_{SRC}$ be the source model, and let $\mathcal{M}_{ARCH}$ be the architecture model. Let $\mathcal{B}_{SRC} = outcomes(s(i), \mathcal{M}_{SRC})$ be the set of allowed outcomes of $s(i)$ with respect to the model $\mathcal{M}_{SRC}$, and let $\mathcal{B}_{ARCH} = outcomes(comp(s)(i), \mathcal{M}_{ARCH})$ be the set of allowed outcomes of $comp(s)(i)$ with respect to the model $\mathcal{M}_{ARCH}$ after compilation with a compiler *comp*. Then *comp* exhibits a concurrency bug if $\exists o \in \mathcal{B}_{ARCH}.\ f(o) \notin \mathcal{B}_{SRC}$.

Definition 2.4.4. *State mapping* A state mapping is a projection from a machine location to a source location denoted using a set notation: `{ a→b }`.

Example 2.4.1. Consider Figure 2.13, which shows the outcomes of executing an unspecified source program and its compiled counterpart, under source and architecture models respectively. We use C/C++ and AArch64 outcomes in this example but the principle is general. Given the state mapping $f =$ `{ P1:W1→P1:r0, P1:W3→P1:r1 }` observe that the compiled program exhibits the outcome `{ P1:W1=1, P1:W3=0 }`, for which there is no source counterpart, indicating a bug in the compiler.

For each litmus test s in a test sequence S, our oracle for compiler testing is defined by the procedure in Definition 2.4.5.

[3]which is a simplification of the relationship between source and target state, but is sufficient for our needs.

```
{ P1:r0=0, P1:r1=0 }      { P1:W1=0, P1:W3=0 }
{ P1:r0=0, P1:r1=1 }      { P1:W1=0, P1:W3=1 }
                        !!{ P1:W1=1, P1:W3=0 }!!
{ P1:r0=1, P1:r1=1 }      { P1:W1=1, P1:W3=1 }
```

Figure 2.13: Illustration of a concurrency-related compiler bug.

Definition 2.4.5. *Our compiler testing oracle*

1. Compute the *expected behaviour* $\mathcal{B}_{SRC} = outcomes(s(i), \mathcal{M}_{SRC})$ using a simulator.

2. Compute the *actual behaviour* $\mathcal{B}_{ARCH} = outcomes(comp(s)(i), \mathcal{M}_{ARCH})$ also using a simulator.

3. Derive state mappings f from the compiler *comp*.

4. Return false if compiler bugs (Def. 2.4.3) are found.

Under multi-copy atomic models, concurrency bugs can arise due to reordering of events [24]. Indeed, almost[4] every concurrency bug report we know of is an expression over test outcomes before and after compilation. New outcomes arise due to the reordering of events on individual threads. Re-orderings occur when either the compiler's *mappings* are incorrect or compiler optimisations re-order or modify assembly sequences, and hence events, on a thread.

2.4.2 Mappings from C/C++ to Assembly Sequences

C/C++ models may be equipped with *compilation schemes*. Compilation schemes are idealised mappings from C/C++ atomic operations to assembly sequences. Some [90, 28, 31] schemes are proven sound, such that if a compiler were to implement them, they should not introduce concurrency bugs. Unfortunately, compilers often step outside the domain of proof when new mappings arise. The interoperability of multiple mappings has not been tested, as the state of the art testing tools operate on a *closed-world assumption* [123], finding bugs when *whole* programs are compiled under one atomics mapping.

[4]Non multi-copy atomic architectures such as Power can exhibit bugs that are not-solely the result of thread-local effects.

Definition 2.4.6. *Mapping*: A mapping is a function from a source language statement, such as a C/C++ atomic operation, to an assembly sequence. We use operation and statement interchangeably throughout this thesis.

Compiler Profiles: A compiler *implements* mappings under *Compiler Profiles (profiles)*. A compiler profile is a mapping from the triple (*comp*, *arg*, *op*) to assembly sequences generated by compiler *comp*, for that atomic operation *op*, using that command-line argument *arg*. Since a compiler may emit different sequences based on the profile used, the compiler actually implements *multiple* mappings for a given C/C++ operation.

Example 2.4.2. Consider Table 2.3 that shows some compiler profiles for load acquire and store release atomics implemented in LLVM. Compiling a load acquire using `clang` and targeting the Armv8-A architecture generates a `LDAR` instruction, whereas targeting Armv8-A when the *Release Consistency Processor Consistency (RCPC)* extension is enabled generates an `LDAPR`. The RCPC extension introduces the `LDAPR` instruction, which has different semantics [153] from `LDAR`. In this case the optimisation flag `-O3` is omitted as it has no effect. It is possible to target other architectures using profiles for Armv7-A, RISC-V, Intel x86-64, IBM PowerPC, and MIPS. GCC has a similar command-line interface.

Atomic Operation	Command	Assembly Sequence
`load(loc,acq)`	`"clang -march=armv8"`	`LDAR W2, [loc]`
	`"clang -march=armv8+rcpc"`	`LDAPR W2, [loc]`
`store(loc,val,rel)`	`"clang -march=armv8"`	`MOV W2, #val` `STLR W2, [loc]`

Table 2.3: Some of LLVM's acquire and release mappings from C/C++ to Armv8-A and Armv8-A with the RCPC extension enabled.

Chapter 3

Téléchat: Testing Using Models

In this chapter we address the problem of observability. The problem is, given a concurrent C/C++ litmus test, can we *observe* concurrency bugs introduced by the compiler under test? This is not a straightforward question to answer since concurrent programs can exhibit multiple unintuitive behaviours. The state of the art compiler testing techniques [118, 41, 160] have made good progress in finding bugs under the C/C++ model by matching expected executions or outcomes of litmus tests before and after compilation. In recent decades, numerous architecture models have been developed and some, such as the AArch64 model [19], have been ratified by processor companies. We develop a technique that leverages both the source and architecture models in our compiler testing oracle to find concurrency bugs in compilers.

We present the Téléchat automated testing framework for concurrent C/C++ programs. The technique compiles the source C/C++ program expressed as a litmus test s, getting an assembly program represented as a litmus test c; simulates s and c under their respective source and architecture memory models; and detects bugs if the assembly litmus test exhibits an outcome that is not exhibited by the source test under the source model (after mapping machine to source states). We implement the technique in the Téléchat toolchain and used it to find a number of new concurrency bugs in LLVM and GCC. Some of these bugs have been fixed and others are triaged for fixing by engineers. Téléchat finds behaviours missed by other techniques on the same inputs. We

conduct an extensive testing campaign of LLVM and GCC, generating over 9,000,000 tests targeting Arm, Intel, RISC-V, PowerPC, and MIPS.

The remainder of this chapter is structured as follows. §3.1 describes the observability problem, §3.2 describes the technique, §3.3 describes the Téléchat toolchain, §3.4 describes how we evaluate our work, and we end with a discussion in §3.5.

3.1 The Observability Problem

We begin by describing the problem faced by concurrency compilation bug finders. To detect bugs we must address the test *oracle problem* [83, 82]. The test oracle problem is the challenge of *"distinguishing the corresponding desired, correct behaviour from potentially incorrect behaviour"* [25] given an input to a system. Since concurrent programs can exhibit multiple behaviours, we refine the problem to a question of observability: given a litmus test and a compiler, do we *observe* the correct sets of behaviours before and after the program has been compiled? Answering this question is tricky, since concurrent programs often exhibit unintuitive behaviours that require specific conditions to arise.

Example 3.1.1. Figure 3.1 shows a C/C++ MP litmus test and an AArch64 assembly test. The outcome `{ P1:W8=0, y=2 }` checked by the `exists` predicate should be forbidden in both cases. The assembly test is produced by compiling the source test using `"clang -march=armv8.2-a -O3"`. These tests expose a concurrency bug, as the compiled test exhibits an outcome not exhibited by the source, under the C/C++ model [80] and AArch64 model [19], respectively. The `SWP` instruction reads from the location `y` using the address stored in the register `X%P1_y` and writes the contents of `W9` back to `y`. Normally, The value read from `y` is stored in the register `WZR`, however `WZR` is a read-only register, and so no data is stored. The write back to `y` has release (`L`) semantics. The `DMB ISHLD` instruction is a memory barrier that ensures all loads prior to the barrier complete before any subsequent memory accesses occur. The outcome `{ P1:W8=0, y=2 }` arises since the effects of executing the `SWPL` may

Message Passing Example

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  store(x,1,rlx);
  fence(rel);
  store(y,1,rlx);
}
P1 () {
  exchange(y,2,rel);
  fence(acq);
  int r0 = load(x,rlx);
}

exists (P1:r0=0 /\ y=2)
```

C/C++ Outcomes

```
{ P1:r0=0, y=1 }
{ P1:r0=1, y=1 }
{ P1:r0=1, y=2 }
```

Assembly Litmus Test

```
{ *x = 0, *y = 0 }

P0              |P1
MOV W9,#1       |MOV W9,#2
STR W9,[X%P0_x]|SWPL W9,WZR,[X%P1_y]
DMB ISH         |DMB ISHLD
STR W9,[X%P0_y]|LDR W8,[X%P1_x]

exists (P1:W8=0 /\ y=2)
```

AArch64 Outcomes

```
  { P1:W8=0, y=1 }
!!{ P1:W8=0, y=2 }!!
  { P1:W8=1, y=1 }
  { P1:W8=1, y=2 }
```

Figure 3.1: (Top) MP litmus test that induces the bug [60] in the test (bottom) after compilation using LLVM.

be reordered past the effects of `DMB ISHLD` and `LDR` on `P1`. The reordering is allowed because C/C++ `atomic_exchange_explicit` (`exchange`) maps to the `SWPL` instruction, but the Arm Architecture Reference Manual [18] states that "*instructions where the destination register is* ***WZR*** *or* ***XZR****, are not regarded as doing a read for the purpose of a* ***DMB ISHLD*** *barrier*". The LLVM dead register definitions (DRD) pass [100] rewrote the destination register of the `SWPL` instruction to be `WZR`, since the result of reading `y` was not used in the source program. The fix, as implemented following our bug report, is to prohibit the DRD pass from zeroing out the destination register of `SWP*` instructions.

Unfortunately, prior testing techniques may not observe these behaviours. The state of the art tools either rely on hardware [160], or do not test compilation

down to the assembly [41]. This is problematic in two ways. Firstly, Table 2.2 of the previous chapter shows that hardware executions give a probabilistic chance of observing a given outcome. While probabilistic methods [160] are effective [162] in finding bugs, it is possible that repeating the same test run twice will produce different sets of outcomes in a concurrent context. This assumes that the underlying hardware implements the behaviour at all. Secondly, techniques that do not test compilation down to assembly will miss bugs in target dependent optimisations, such as the DRD pass. By testing using models of the architecturally allowed envelope of all conforming hardware, we sidestep these issues.

We present the Téléchat compiler testing technique, which finds bugs when an outcome of a compiled litmus test under its architecture model is not an outcome of the source test under its source model. We implement this technique in the Téléchat toolchain. By using the `herd` simulator and `diy` test generator, Téléchat finds bugs *automatically*. By testing under executable models of both the source and target languages, Téléchat covers compilation from the source to assembly. We use Téléchat to conduct a range of experiments and a campaign of compiler testing. In doing so we uncover a number of new, developer-confirmed compiler bugs. One such bug is in Figure 3.1. We contribute experimental evidence that suggests Téléchat finds behaviours missed by the state of the art on the same inputs. Since we parameterise testing over both source and target models we can test compilation targeting Armv8, Armv7, Intel, RISC-V, PowerPC, and MIPS. By testing compilation under these models we achieve test coverage of over 9,000,000 tests, which is the most extensive concurrency compilation testing campaign as far as we know. We depend on model correctness, and so we compare test results under numerous source and architecture models, finding a bug in an unofficial model of Armv7.

3.2 The Téléchat Technique

We present the Téléchat technique in Figure 3.2, which proceeds as follows:

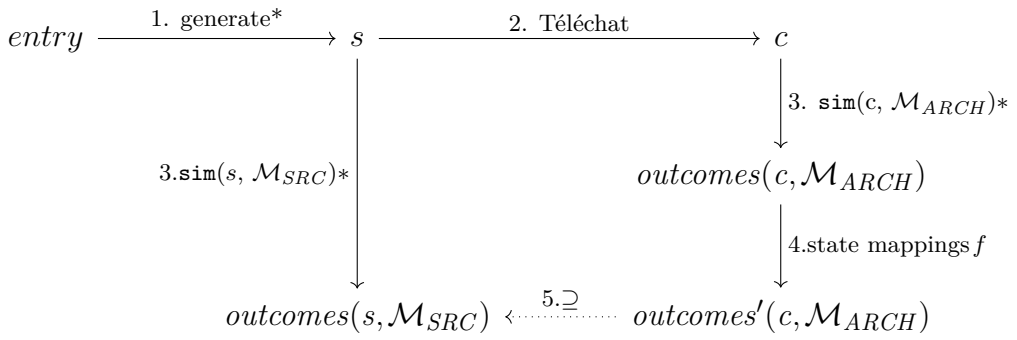

Figure 3.2: The Téléchat tool and prior work (marked $*$).

1. Generate a concurrent C/C++ litmus test s.
2. Given a compiler profile *comp*, Téléchat prepares s for compilation, compiles it using *comp*, disassembles *comp*(s), and returns an assembly litmus test c and state mappings f (from assembly to source program states).
3. Simulate s under a C/C++ model, get the set $outcomes(s, \mathcal{M}_{SRC})$, simulate c under its architecture model, get the set $outcomes(c, \mathcal{M}_{ARCH})$.
4. Apply f to $outcomes(c, \mathcal{M}_{ARCH})$, get the set $outcomes'(c, \mathcal{M}_{ARCH})$.
5. If $outcomes'(c, \mathcal{M}_{ARCH}) \not\subseteq outcomes(s, \mathcal{M}_{SRC})$, then there is a bug.

The technique straightforwardly implements Def. 2.4.3 from the previous chapter. Téléchat enables automatic testing from source to assembly by completing the graph in Figure 3.2. We extend the `diy` [15] test generator, the `herd` [17] simulator, and `mcompare` [12] outcome comparison tools to make this possible. This technique had not been automated before, since neither the models nor the tools were mature enough to be used in this way. The `litmus` [16] tool can also compile C/C++ litmus tests, and additionally run them on hardware. The Téléchat toolchain differs in that `litmus` embeds the program in a test rig that is repeatedly run on hardware to increase the chances of observing all outcomes, whereas Téléchat needs no such skeleton as the simulator takes on the responsibility of computing executions. We thus instantiate our compiler testing oracle (Def. 2.4.5) with `herd` and its implementation of the models in question.

3.3 The Téléchat Toolchain

We describe the Téléchat toolchain and challenges faced during development. We use the running example to illustrate how each stage of the toolchain works.

3.3.1 Téléchat Tool Implementation

The Téléchat toolchain consists of `l2c`, `c2s`, and `s2l`. By passing files along at each step, we can customise the input parameters for the target architecture, compiler, and so on. Step 2 (Téléchat) of Figure 3.2 proceeds as follows:

1. *Input*: A C/C++ litmus test s and a compiler profile *comp*. Tests must use the `.litmus` test format.

2. The *litmus2c* (`l2c`) tool prepares s for compilation, returning a program s'. Since C/C++ litmus tests are typically simplified programs often lacking required components, such as a `main` function, function return types, and `pthread` invocations of the threads in question, we add these features to the program so that it can be compiled either as a library (without `main`) or as an executable, both with or without POSIX threads.

3. The *c2assembly* (`c2s`) tool compiles s' with ELF relocations (requires flags `-c -g`) and disassembles the object file. `c2s` returns an assembly file o and state mappings f as derived from the compiler emitted metadata and symbol table information.

4. The *assembly2litmus* (`s2l`) tool parses o and constructs an optimised assembly litmus test c. This test is also in the `.litmus` format.

5. *Output*: Pass s and c to `herd` for simulation under source and architecture models (step 3 of Figure 3.2). Pass f to `mcompare` (step 4) to project machine outcomes onto source outcomes so that a comparison can be made (step 5).

Example 3.3.1. Figure 3.3 shows the C program output by `l2c`, given the C/C++ litmus test in Figure 3.1 as input.

```
/*****************************************************/
/*                  the Telechat toolsuite           */
/*                                                   */
/* Luke Geeson, University College London, UK.       */
/*                                                   */
/* This C source is a product of l2c and includes source  */
/* that is governed by the CeCILL-B license.         */
/*                                                   */
/*****************************************************/
//
// Generated with:
// l2c bug.litmus
//
// C test
//
// { *x = 0; *y = 0; }

#ifndef __cplusplus
#include <stdatomic.h>
#include <stddef.h>
#include <stdint.h>
#else
#include <atomic>
using namespace std;
#endif

// globals
int* P1_r0;
_Atomic int* x;
_Atomic int* y;

// Test body
void P0() {
  atomic_store_explicit(x,1,memory_order_relaxed);
  atomic_thread_fence(memory_order_release);
  atomic_store_explicit(y,1,memory_order_relaxed);
}
void P1() {
  int r0;
  atomic_exchange_explicit(y,2,memory_order_release);
  atomic_thread_fence(memory_order_acquire);
  r0 = atomic_load_explicit(x,memory_order_relaxed);
  *P1_r0 = r0;
}

// exists (P1:r0=0 /\ y=2)
```

Figure 3.3: The output of `l2c`, given Figure 3.1 as input.

Figure 3.4 shows the AArch64 assembly program output by `c2s`, given Figure 3.3 as input. This program is represented using the *Executable and Linkable Format (ELF)*. ELF files lay out the assembly for each function in

segments within the `.text` section. Each assembly instruction is prefixed by its address, and certain instructions are followed by `R_AARCH64_*` relocations. Relocations provide metadata for the linker to resolve numeric addresses with symbolic locations. In this case the ADRP instruction at address `0` in `P0` computes the address of `x` and stores it in the register `X8`.

```
file format elf64-littleaarch64

Disassembly of section .text:
0000000000000000 <P0>:
       0:        adrp  x8, 0x0 <P0>
    0000000000000000:  R_AARCH64_ADR_PREL_PG_HI21 x
       4:        mov w9, #1
       8:        adrp  x10, 0x0 <P0+0x8>
    0000000000000008:  R_AARCH64_ADR_PREL_PG_HI21 y
       c:        ldr x8, [x8, #0]
    000000000000000c:  R_AARCH64_LDST64_ABS_LO12_NC x
      10:        str w9, [x8, #0]
      14:        dmb ish
      18:        ldr x8, [x10, #0]
    0000000000000018:  R_AARCH64_LDST64_ABS_LO12_NC y
      1c:        str w9, [x8, #0]
      20:        ret x30

0000000000000024 <P1>:
      24:        adrp  x8, 0x0 <P1>
    0000000000000024:  R_AARCH64_ADR_PREL_PG_HI21 y
      28:        mov w9, #2
      2c:        adrp  x10, 0x0 <P1+0x8>
    000000000000002c:  R_AARCH64_ADR_PREL_PG_HI21 x
      30:        ldr x8, [x8, #0]
    0000000000000030:  R_AARCH64_LDST64_ABS_LO12_NC y
      34:        swpl  w9, wzr, [x8]
      38:        adrp  x9, 0x0 <P1+0x14>
    0000000000000038:  R_AARCH64_ADR_PREL_PG_HI21 P1_r0
      3c:        dmb ishld
      40:        ldr x8, [x10, #0]
    0000000000000040:  R_AARCH64_LDST64_ABS_LO12_NC x
      44:        ldr w8, [x8, #0]
      48:        ldr x9, [x9, #0]
    0000000000000048:  R_AARCH64_LDST64_ABS_LO12_NC P1_r0
      4c:        str w8, [x9, #0]
      50:        ret x30
```

Figure 3.4: The output of `c2s`, given Figure 3.3 as input.

Figure 3.5 shows the output of `s2l`, given Figure 3.4 as input. Figure 3.5 shows the AArch64 litmus test that has three major changes. Firstly, the

program no longer features ADRP; LDR sequences, instead the initial state declares symbolic registers Pn_x=x for each location x and thread n. Secondly, the relevant instructions use symbolic registers directly (e.g. SWPL W9, WZR, [X%P1_y]. Lastly, the final state predicate replaces locals P1:r0 with globals P1_r0 where local state is stored. We justify these changes in the following sections. Before we do, we detail how compilers and mappings are represented.

```
AArch64 test

{ *x=0; *y=0; P1_r0=0;
  uint64_t .bss[3]={x,y,P1_r0};
  uint64_t %P0_x=x; uint64_t %P0_y=y;
  uint64_t %P1_r0=P1_r0;
  uint64_t %P1_x=x; uint64_t %P1_y=y }

(*************************************************************)
(*                  the Telechat toolsuite                   *)
(*                                                           *)
(* Luke Geeson, University College London, UK.               *)
(*                                                           *)
(* This Assembly is a product of s2l and includes source     *)
(* that is governed by the CeCILL-B license.                 *)
(*                                                           *)
(* Further this toolchain uses compiler toolchains and is    *)
(* subject to license requirements of those toolchains       *)
(*                                                           *)
(* Compiler:                                                 *)
(* clang -c -g -O3 -march=armv8.2-a  -pthread --std=c11      *)
(*                                                           *)
(* Disassembler:                                             *)
(* objdump -Dr --disassemble --section=.text [...]           *)
(*                                                           *)
(* Generated with:                                           *)
(* s2l -jsonfile profiles.json -profile                      \
   llvm-O3-AArch64-8.2 bug.litmus -no-compress -save-temps*)
(*                                                           *)
(*************************************************************)

  P0                  |  P1                        ;
   MOV W9,#1          |   MOV W9,#2                ;
   STR W9,[X%P0_x]    |   SWPL W9, WZR,[X%P1_y]    ;
   DMB ISH            |   DMB ISHLD                ;
   STR W9,[X%P0_y]    |   LDR W8,[X%P1_x]          ;
   RET X30            |   STR W8,[X%P1_r0]         ;
                      |   RET X30                  ;

exists (P1_r0=0 /\ y=2)
```

Figure 3.5: The output of s2l, given Figure 3.4 as input.

Figure 3.6 shows the compiler profile and state mappings, both stored as JSON files. The profiles file contains one or more compiler profiles used to translate a C/C++ program, disassemble it, and parse metadata from the ELF file. The mappings file contains mappings from target to source locations, including their locations in the `.bss` section of the ELF file. To reproduce similar tests, please see the artifact in Appendix A. This bug has been fixed in compilers that ship with systems including MacOS — you may not observe it outside of the artifact environment.

```
{
  "comment": "Comment summarises profiles in this file",
  "profiles": [
    {
      "comment":"clang/lld/gnu-objdump",
      "name":"llvm-O3-AArch64-8.2",
      "comp":"clang",
      "arch":"AArch64",
      "cmp-args":"-c -g -O3 -march=armv8.2-a  -pthread \
          --std=c11 [...]",
      "dis-tgt": "objdump",
      "dis-args": "-Dr --disassemble --section=.text   \
          -Mno-aliases --no-show-raw-insn",
      "sym-tgt": "objdump",
      "sym-args": "-t",
      "model":"aarch64.cat"
    },
  ]
}
```

```
[
  {
    "bug": {
      "mappings": [
        { "P1_r0": "P1:r0" },
        { "y": "y"         }
      ],
      "sections": [
        { "P0": ".text"    },
        { "x": ".bss"      },
        { "y": ".bss"      },
        { "P1": ".text"    },
        { "P1_r0": ".bss"  }
      ]
    }
  }
]
```

Figure 3.6: The compiler profile and state mappings.

3.3.2 Challenges Faced During Implementation

In implementing the Téléchat toolchain we faced two major implementation challenges: representing compiled programs and supporting compiled programs.

Compiled programs are binary files. Litmus tests are symbolic programs. Representing compiled programs as litmus tests is not a direct conversion, since addresses can be computed from numbers (such as the base address of a `.bss` section in an ELF file), which can be manipulated with arithmetic instructions. For example, in Figure 3.4, the `ADRP` instruction computes the PC-relative address of `x`, stores the result in register `X8`, and then the `LDR` at address `0xc` loads the pointer to the address from the register `X8`, adds the offset `#0`, and stores the address back in `X8`. A linker may change all numeric addresses above, all instructions concerning numeric addresses, and will remove all metadata including symbolic locations. Litmus tests on the other hand represent memory locations as symbolic variables `x` that have no memory layout constraints.

Our first attempt at representing compiled programs involved formalising the semantics of numeric addresses, which requires changing the memory access semantics of `herd` to support both numeric and symbolic addresses. This did not change the semantics of existing symbolic programs, but it was not accepted by code reviewers since it changed the paradigm of litmus tests. Instead, we use DWARF metadata and symbol table information to map numeric addresses to symbolic locations. We use the 'compile-only' (`-c`) flag in LLVM and GCC, which emits assembly programs with symbolic locations intact, that is before the linker has applied memory layout, numeric address translation, and link-time optimisations[1]. We replace `ADRP` with instructions that access a symbolic location `Pn_x` directly. As such the tests that Téléchat generates today, and `herd` accepts, are as accurate as the compiler metadata, but do not represent programs exactly as they appear in ELF files. This is fine for programs that use virtual memory, but may not support embedded applications where memory layout constraints map directly onto hardware.

[1]Running the assembly litmus tests in all experiments in this thesis produce the same outcomes using our numeric address patch or with the symbolic mode of `herd`.

Developing Téléchat required significant improvements to the `herd` tool-suite [12]. Improvements included formalising the semantics of, and implementing, many new instructions, developing new data vector types, and generally improving the existing tools. Adding new instructions took the most time, as `herd` requires that the semantics of instructions are manually specified. We implemented many new instructions for Armv8, Armv7, Intel x86-64, RISC-V, MIPS, and IBM PowerPC architectures. We found that the semantics of many existing instructions had changed over the years, and so we added a regression suite for the `herd` tool-suite itself. Compiled programs often store data together, which required us to implement a vector datatype in `herd` to model memory layout and store pair instructions that span contiguous locations. Our vector work was then used as the basis of multiple projects modelling the NEON [13] and SVE [14] extensions of the Arm architecture. Lastly, we added state mapping support to the `mcompare` tool, which enabled us to compare outcomes of tests of different languages.

3.4 Evaluation

We evaluated Téléchat by conducting experiments using multiple compilers. Our results suggest Téléchat improves on the state of the art as we found numerous bugs and behaviours missed by prior work when run on the same inputs.

3.4.1 Comparing Téléchat Against `c4`

Téléchat was developed at the same time as `c4` [160, 159, 93], but neither group was aware of the others' work. Both tools use the `herd` tool-suite [12], but Téléchat relies on simulation *only* whereas `c4` relies on hardware executions to collect assembly outcomes. To compare these tools, we compare the results of passing the same 89 tests as input to each tool. The results in this case are litmus test outcomes before and after compilation. We found that Téléchat finds outcomes that `c4` does not observe on the same input tests.

Example 3.4.1. Figure 3.7 shows a Load Buffering (LB) litmus test and its outcomes allowed by the RC11 model [46]. We compile Figure 3.7 (top) using Téléchat and `"clang -O3"` to get the Arm AArch64 litmus test in Figure 3.7 (bottom). Observe that the outcome `{ P0:r0=1, P1:r0=1 }` in the `exists` clause is forbidden by the RC11 model, but its compiled counterpart is exhibited by the compiled program. Windsor et al. state [159] that `c4` missed this compiled program outcome when executing the test on a Raspberry Pi. We elaborate below on why hardware may not exhibit such outcomes. We observe the outcome when running the Téléchat-generated test using `herd`. To be clear, this is not a compiler bug, since ISO C/C++ explicitly permits load buffering — observing the compiled program behaviour is the focus.

Load Buffering Example

C/C++ Litmus Test

```
{ atomic_int *x, *y = 0 }

P0 () {
  int r0 = load(x,rlx);
  fence(rlx);
  store(y,1,rlx);
}
P1 () {
  int r0 = load(y,rlx);
  fence(rlx);
  store(x,1,rlx);
}

exists (P0:r0=1 /\ P1:r0=1)
```

RC11 Outcomes

```
{ P1:r0=0, P1:r0=0 }
{ P1:r0=0, P1:r0=1 }
{ P1:r0=1, P1:r0=0 }
```

AArch64 Litmus Test

```
{ x, y = 0 }

P0                |P1
MOV X1, #1        |MOV X1, #1
LDR X0, [X%P0_x]|LDR X0, [X%P1_y]
STR X1, [X%P0_y]|STR X1, [X%P1_x]

exists (P0:X0=0 /\ P1:X0 = 0)
```

AArch64 Outcomes

```
{ P0:X0=0, P1:X0=0 }
{ P0:X0=0, P1:X0=1 }
{ P0:X0=1, P1:X0=0 }
!!{ P0:X0=1, P1:X0=1 }!!
```

Figure 3.7: RC11 forbids `{ P0:r0=1, P1:r0=1 }` but numerous compiled programs exhibit it under their respective architecture models.

It is well known that many (though not all[2]) implementations of the Arm architecture do not exhibit this load buffering behaviour and so it is perhaps not surprising that `c4` misses it. However, the Arm architecture, and by extension its memory model, permits load buffering and so compiler testing techniques should aim to detect such behaviours.

We found hundreds of litmus tests that induce load buffering behaviour when compiled by either LLVM or GCC. We also observe LB when targeting Armv7, IBM PowerPC, and RISC-V. Further, Téléchat observes these same test outcomes every time whereas `c4` requires that the hardware implements this behaviour and that users can 'stress-test' [160] the hardware to reproduce it. `c4` is not guaranteed to observe the same outcomes on different machines, or even the same machine.

Téléchat is useful when hardware is inaccessible. For instance, we assisted Arm's engineers with a query from one of Arm's partners, who proposed to change the compilation of C/C++ atomic acquire loads when targeting Armv8.3-A. The change had promising performance characteristics on unspecified hardware, but correctness was untested beyond reading the specifications. The `c4` tool could not be deployed here, since the user requires access to hardware that may not be available. We tested the proposal under simulation and Arm's engineers accepted the proposal based on our findings.

Téléchat does not subsume the state of the art as, for example, simulation does not terminate when checking large programs. The `c4` tool can thus detect behaviours in large concurrent programs that simply do not terminate under our simulation-based approach. Further, the `c4` tool conducts rigorous fuzzing to increase the chances of observing concurrent behaviours. Téléchat does no such fuzzing, but rather focuses on testing the mappings of small litmus tests. We explore the intersection of these techniques in Chapter 5.

[2]Sarkar et al. [135] observe LB on an Apple A9 and NVIDIA Tegra2 chips `https://www.cl.cam.ac.uk/~pes20/arm-supplemental/arm001.html#toc5`

3.4.2 The Local Variable Problem

The local variable problem affects all concurrency compilation testing techniques and masks a subset of compiler bugs, known as *Heisenbugs*, if not explicitly addressed. The problem concerns transformations allowed by source models, in our case the C/C++ [41] model, that delete data required to observe bugs, but instrumenting the program to preserve such data *prevents* observation of the bug. In hindsight this is not surprising, but it appears to counter the claim that *"optimisations affecting only the thread-local state cannot induce concurrency compiler bugs"* made by prior work [118]. We reported a new Heisenbug [60] (see Figure 3.1), reproduced two such bugs for Arm's engineers, and discuss how we address the local variable problem in Téléchat.

Example 3.4.2. Figure 3.8 (top left) shows a non-atomic load buffering test. When simulated under a model, the values of the local data `P0:r0` and `P1:r0` are tracked for use in the final program outcomes. The C/C++ memory model [80] permits the compiler to remove unused local data however. A litmus test that refers to *deletable* data [41] in its final outcomes, such as `P0:r0` and `P1:r0`, will have no data to refer to if the compiler removes it. When compiling Figure 3.8 (top left) with `"clang -O2"` we get Figure 3.8 (bottom left, in C for illustration). The only allowed outcome of Figure 3.8 (bottom left) is `{ P0:r0=0, P1:r0=0 }` since `herd` assumes data is zero-initialised. On hardware one may observe nonsense data when attempting to observe deleted data.

Unfortunately, we cannot ignore local data. Local reads are snapshots of shared data at points in a program's execution. Many concurrency idioms rely on local data to check whether the thread-local reordering of accesses occurs. For example, LB checks whether for instance caches *buffer* loads during execution. Further, thread-local reordering is common in many processors - we cannot ignore it, even if the C/C++ model permits the removal of local data.

Today's techniques may miss bugs in optimisations that delete local data. Such techniques cannot test the compilation of load buffering, message passing, and others unless local data persists. Authors of prior work either overlook the

C/C++ Litmus Test Before Optimisation

C/C++ Litmus Test | RC11 Outcomes

```
{ int *x, *y = 0 }

P0 () {
  int r0 = *x;
  *y = 1;
}
P1 () {
  int r0 = *y;
  *x = 1;
}

exists (P0:r0=1 /\ P1:r0=1)
```

```
{ P1:r0=0, P1:r0=0 }
{ P1:r0=0, P1:r0=1 }
{ P1:r0=1, P1:r0=0 }
{ P1:r0=1, P1:r0=1 }
```

After Optimisation

```
{ int *x, *y = 0 }

P0 () {
  // deleted
  *y = 1;
}
P1 () {
  // deleted
  *x = 1;
}

exists (P0:r0=1 /\ P1:r0=1)
```

```
{ P1:r0=0, P1:r0=0 }
```

Figure 3.8: (Top) LB litmus test. (bottom) Test after `"clang -O2"` deletes data.

issue [41, 160] or claim [118] local optimisations cannot induce concurrency-related bugs. In particular, Morisset et al. [118] claim that *"optimisations affecting only the thread-local state cannot induce concurrency compiler bugs"*. We now explore this claim further.

Example 3.4.3. Figure 3.9 shows a message passing litmus test and its outcomes under the C/C++ model. `P1` uses an `atomic_fetch_add_explicit (fetch_add)` RMW operation that reads the value of `y` into `P1:r1`, adds 1 to it, and writes the new value to `y`. In this case `P1:r1` is unused and is deletable This induced *two* bugs in past versions of LLVM and GCC, first by targeting the incorrect Arm instruction and second by deleting `P1:r1`. In both cases, the outcome `{ P1:r0=0, y=2 }` is forbidden by the C/C++ model, but exhibited

MP Litmus Test Example

C/C++ Litmus Test

```
{ atomic_int *x, *y = 0 }

P0 () {
  store(x,1,rlx);
  fence(rel);
  store(y,1,rlx);
}
P1 () {
  int r1 = fetch_add(y,1,rlx);
  fence(acq);
  int r0 = load(x,rlx);
}

exists (P1:r0=0 /\ y=2)
```

Outcomes

```
{ P1:r0=0, y=1 }
{ P1:r0=1, y=1 }
{ P1:r0=1, y=2 }
```

Assembly Litmus Test

```
{ x, y = 0 }

P0                |P1
MOV W9, #1        |MOV W9, #2
STR W9, [X%P0_x]|LDADD W9, WZR, [X%P1_y]
DMB ISH           |DMB ISHLD
STR W9, [X%P0_y]|LDR W8, [X%P1_x]

exists (P1:W8=0 /\ y=2)
```

AArch64 Outcomes

```
  { P1:W8=0, y=1 }
!!{ P1:W8=0, y=2 }!!
  { P1:W8=1, y=1 }
  { P1:W8=1, y=2 }
```

Figure 3.9: A previously miscompiled MP litmus test that demonstrates thread-local optimisations can induce bugs. Found by Will Deacon.

by the compiled program when targeting Arm AArch64. Engineers fixed the first bug by replacing the incorrect `STADD` instruction with `LDADD`. The second bug is observed when the LLVM DRD pass [100], a thread-local optimisation, zeroes the destination-register of `LDADD` as it did in Figure 3.1. The outcome `{ P1:r0=0, y=2 }` is observed in the latter case as `LDADD` aliases `STADD` when the destination register is the zero register.

Interestingly, these bugs disappear if one attempts to study them. Message passing tests classically check the reordering of `P1:r0` and `P1:r1`. If instead we delete `P1:r1` and check the value of `y`, then we see the reordering. In other words, you only catch the bug through *indirect* observation. It appears that thread-local optimisation can introduce bugs via indirect observation.

We implement a mitigation in Téléchat but acknowledge it isn't a general solution. Téléchat augments Figure 3.9 with global variables that store local data at the end of each thread (see Figure 3.3 for an example). This augmentation is optional to allow thread-local optimisations to be tested. The original code under test remains, but with the additional constraint that local data persists after compilation. We update the initial and final states to reflect this new data. It is unsatisfactory to modify the test, but we have had success thus far (detecting the bug in Example 3.1.1). We are open to better alternatives.

3.4.3 Bug-Finding Campaign

While developing Téléchat we reported four new concurrency bugs [58, 61, 60, 57] including the bug in Figure 3.1, and a missed optimisation [66] for the MIPS backend of GCC. We describe the four bugs and the MIPS example.

First bug: We reported a bug [58] in the compilation of 128-bit sequentially consistent [91] `load` operations. The bug occurs when the load is implemented using a load pair (`LDP`) instruction when targeting Armv8.4. The Armv8.4 Large Systems Extension (v2) ensures load or store pair instructions are single-copy atomic [18], assuming accesses are 16-byte aligned to normal memory. This means you can use an `LDP` instruction in place of a potentially more costly compare-and-swap (CAS) loop. `LDP` has no ordering requirements and so accesses made by `LDP` can be reordered before a prior store that has no synchronisation. We propose to fix sequential consistency in LLVM by adding synchronisation, following GCC [51].

Second bug: We reported a wrong-endian bug [61] in the compilation of 128-bit atomic stores. Since AArch64 has 64-bit register sizes, an 128-bit store is implemented using a *pair* of 64-bit registers. We report that the order registers are written to memory is flipped by atomic store operations. This affects store-release-exclusive pair instructions in CAS loops (for Armv8.3 or below), and individual store pair instructions (Armv8.4 or above). We propose to flip order of the writes to fix the bug.

Third bug: Is described in Example 3.1.1.

Fourth bug: We reported [57] a bug in the implementation of 128-bit `const`-qualified atomic loads. We found that the program crashes at run-time, as the load is compiled to a store instruction that attempts to write to read-only memory. This case study required some elaborations on the AArch64 memory model and so we describe it in Chapter 5.

Optimisation opportunity: We reported [66] an optimisation opportunity in the MIPS backend of GCC. We discovered that GCC (and LLVM) are conservative in optimising instructions that access `atomic` data. Extra code is emitted, since `atomic`-accessing code cannot inhabit branch delay slots. GCC maintainers note that `atomic` data is considered `volatile` for practical reasons, despite no change in compiled program outcomes under models. Whether it is *still* valid to treat `atomic` as `volatile` is further work.

3.4.4 Large-Scale Differential Testing

We use Téléchat to conduct differential testing of LLVM and GCC. We check compatibility between compilers, for multiple architectures and optimisations. We generate over 9,000,000 tests that have 2 to 5 threads, up to 5 shared variables, and up to 50 lines of compiled assembly code. We test:

- LLVM and GCC compilers: From C/C++ to Armv8 AArch64 (64-bit), Armv7-a (32-bit), RISC-V, Intel x86-64, IBM PowerPC, and MIPS.
- Optimisation levels: `-O1`, `-O2`, `-O3`, `-Ofast`, and for GCC `-Og`.

C/C++ constructs:	`(atomic operations|non-atomic operations` `|fences|control-flow|straight-line code)+`
Compiler under test:	`(LLVM|GCC)`
Optimisation flags:	`(-O1|-O2|-O3|-Ofast|-Og)+`

Target Architecture:	`(Armv8 AArch64 (64-bit official)`
	`| Armv7-a (32-bit unofficial)`
	`| RISC-V (64-bit official)`
	`| Intel x86-64 (64-bit)`
	`| MIPS (64-bit)`
	`| IBM PowerPC (64-bit))`

Table 3.1: We test C/C++ constructs × Compiler × Flags × Arch.

- Compare a compiler with itself at increasing optimisation levels: e.g. `"clang -O1"` vs. `"clang -O2"`.
- Compare compilers at each level, e.g. `"clang -O1"` vs. `"gcc -O1"`.

Table 3.1 defines the combinations of test, compiler, and architecture under test. Our tests feature control-flow, atomic operations, non-atomic operations, fences, straight-line code, signed integers, unsigned integers, and access sizes ranging from 8-bits up to 64-bits. Following the steps in §3.2, we generate multiple source C/C++ *test sets* enumerating the features in Table 3.1 using `diy` [15]. For each test set, we use Téléchat with the compiler under test to generate *multiple* assembly test sets according to multiple compiler profiles. Both source and target tests are passed to `herd` for simulation under the RC11 [46] model and architecture model, respectively. Lastly, we apply state mappings to the compiled test outcomes (see Figure 3.6) and use `mcompare` to find assembly program outcomes that are not outcomes of the source program after mapping machine states to source states. We split up the test results by tests where the compiled program outcomes are not a subset of the source program outcomes ($\not\subseteq$), or vice versa ($\not\supseteq$).

Table 3.2 details our results and suggests Téléchat is effective as it found tricky concurrency behaviours hidden in over 9,000,000 compiled tests, given 167,184 tests as input. The 2,352 tests common to Armv8 (official), Armv7 (unofficial), RISC-V (official), and IBM PowerPC are due to 294 variants of the load buffering pattern in Fig 3.7. Since Intel x86-64 implements the total-store order model [136] there are no differences. To be clear, these results are not *bugs* in today's compilers, since we used the RC11 model [90] that is not ratified by the C/C++ standards. The ISO C/C++ standards explicitly permit load-to-store reordering (§7.17.3 of C23 [80]), whereas RC11 forbids it. Téléchat is parameterised over models, and we repeat testing using a modified `rc11+lb.cat` model. Under this new model all 2,352 results drop to 0 if load buffering is permitted. This suggests our technique is particularly sensitive to the correctness of models.

	-O1	-O2	-O3	-Ofast	-Og	Total %
Armv8 AArch64			clang			
$\not\subseteq$	2,352	2,352	2,352	2,353	-	0.23%
$\not\supseteq$	44,300	44,300	44,300	44,300	-	4.42%
Armv7-a			clang			
$\not\subseteq$	2,352	2,352	2,352	2,352	-	0.25%
$\not\supseteq$	68,228	68,228	68,228	68,228	-	6.91%
RISC-V			clang			
$\not\subseteq$	2,352	2,352	2,352	2,352	-	0.23%
$\not\supseteq$	34,204	34,204	34,204	34,204	-	5.44%
IBM PowerPC			clang			
$\not\subseteq$	2,352	2,352	2,352	2,352	-	0.23%
$\not\supseteq$	43,956	43,956	43,956	43,956	-	4.38%
Intel x86-64			clang			
$\not\subseteq$	0	0	0	0	-	0.0%
$\not\supseteq$	64,112	64,112	64,112	64,112	-	6.39%
MIPS			clang			
$\not\subseteq$	0	0	0	0	-	0.0%
$\not\supseteq$	69,664	69,664	69,664	69,664	-	7.09%
	-O1	-O2	-O3	-Ofast	-Og	Total %
Armv8 AArch64			gcc			
$\not\subseteq$	2,352	2,352	2,352	2,352	2,352	0.23%
$\not\supseteq$	44,300	44,300	44,300	44,300	44,300	4.42%
Armv7-a			gcc			
$\not\subseteq$	3,480	2,352	2,352	2,352	2,352	0.25%
$\not\supseteq$	69,890	70,220	70,220	70,220	70,220	6.91%
RISC-V			gcc			
$\not\subseteq$	2,352	2,352	2,352	2,352	2,352	0.23%
$\not\supseteq$	70,772	70,772	70,772	70,772	70,772	5.44%
IBM PowerPC			gcc			
$\not\subseteq$	2,352	2,352	2,352	2,352	2,352	0.23%
$\not\supseteq$	43,956	43,956	43,956	43,956	43,956	4.38%
Intel x86-64			gcc			
$\not\subseteq$	0	0	0	0	0	0.0%
$\not\supseteq$	64,112	64,112	64,112	64,112	64,112	6.39%
MIPS			gcc			
$\not\subseteq$	0	0	0	0	0	0.0%
$\not\supseteq$	72,488	72,008	72,008	72,008	72,488	7.09%

Table 3.2: Results under RC11. Since `clang` does not support `-Og`, these results are marked '-'. 167,184 tests input, 9,027,936 tests output.

3.4.5 Complexity Can Impede Bug Finding

We end by discussing our attempts to scale simulation using `herd` and how we reduced its complexity. Téléchat [71] relies on the `herd` [12] simulator to compute source and compiled program behaviour under memory models. Unfortunately, simulation complexity expands factorially [71] in the size of the litmus test and relaxations of the model. Simulation timeouts are not acceptable for automated testing, since testing must keep pace with developers. We describe how we address scalability in practice so that testing terminates in seconds. Simulation scalability remains a problem for finding bugs.

Example 3.4.4. Figure 3.10 (top) is a three thread LB variant of Figure 3.8 that uses relaxed atomic operations. Initially Téléchat [71] compiled Figure 3.10 (top) directly into Figure 3.10 (middle). We have not however observed Figure 3.10 (middle) terminate using `herd` after 72 hours. The problem is that each source statement (e.g. `int r0 = load(x,rlx)`) is translated into a pointer load (`MOV;LDR`) and a value load/store (`LDR`/`STR`). Each assembly instruction contributes to the reads-from (rf) relation computed by `herd` and the number of generated executions explodes.

To mitigate this we optimise Figure 3.10 (middle) into Figure 3.10 (bottom), which terminates in milliseconds. By removing the indirection via pointer loads we obtain tests that are minimal in a sense that they check for cyclic executions specified by the source litmus test without unnecessary instructions. As such, litmus tests up to 5 threads typically terminate within seconds. We do not prove that our optimisations are sound, but an informal soundness argument is that `herd` uses symbolic locations which cannot be accessed by other threads. The locations associated with removed memory accesses cannot be named by other threads and an access cannot side effect other symbolic locations (assuming no overlaps in the address space). Soundness thus depends on the non-interference of other threads after applying our optimisations. This holds for multi-copy atomic models [24] such as the AArch64 model [19], where concurrency bugs primarily arise due to reordering of thread-local events rather than interference.

Input Load Buffering Test

```
{ *x = 0, *y = 0, *z = 0 }

P0 () {
  int r0=load(x,rlx);
  store(y,1,rlx);
}
P1 () {
  int r1=load(y,rlx);
  store(z,1,rlx);
}
P2 () {
  int r2=load(z,rlx);
  store(x,1,rlx);
}

exists (P0:r0=1 /\ P1:r1=1 /\ P2:r2=1)
```

Unoptimised Test (does not terminate under simulation)

```
{ *x=0, *y=0, *z=0 }

 P0                | P1                | P2
  MOV X8,X%P0_px   |  MOV X8,X%P1_py   |  MOV X8,X%P2_pz
  MOV X9,X%P0_py   |  MOV X9,X%P1_pz   |  MOV X9,X%P2_px
  LDR X8,[X8]      |  LDR X8,[X8]      |  LDR X8,[X8]
  LDR X9,[X9]      |  LDR X9,[X9]      |  LDR X9,[X9]
  MOV W10,#1       |  MOV W10,#1       |  MOV W10,#1
  LDR W8,[X8]      |  LDR W8,[X8]      |  LDR W8,[X8]
  STR W10,[X9]     |  STR W10,[X9]     |  STR W10,[X9]
  MOV X9,X%P0_r0   |  MOV X9,X%P1_r1   |  MOV X9,X%P2_r2
  LDR X9,[X9]      |  LDR X9,[X9]      |  LDR X9,[X9]
  STR W8,[X9]      |  STR W8,[X9]      |  STR W8,[X9]

exists (P0_r0=1 /\ P1_r1=1 /\ P2_r2=1)
```

Optimised Test (terminates in seconds)

```
{ *x=0, *y=0, *z=0 }

P0                | P1                | P2
MOV W10,#1        | MOV W10,#1        | MOV W10,#1
LDR W8,[X%P0_x]   | LDR W8,[X%P1_y]   | LDR W8,[X%P2_z]
STR W10,[X%P0_y]| STR W10,[X%P1_z]| STR W10,[X%P2_x]
STR W8,[X%P0_r0]| STR W8,[X%P1_r1]| STR W8,[X%P2_r2]

exists (P0_r0=1 /\ P1_r1=1 /\ P2_r2=1)
```

Figure 3.10: Three thread Atomic LB variant of Figure 3.8.

This is acceptable for automated testing, and enabled our experiments in §5.5 to finish. Unfortunately, compiling larger C/C++ tests still do not terminate under `herd`.

We conclude that simulation scalability limits bug-hunting, and other techniques should be considered. It is unfortunate that accelerating testing by running tests on hardware [160, 159] may miss bugs [71], and that simulation suffers from scalability issues. That said, most concurrency–related compiler bugs we know of can be reduced to small litmus tests and run using a tool like `herd`. Of course there are bugs [96, 43] that do not straightforwardly fit into litmus tests, for which scalable techniques *are* required. To summarise, simulation may be used to validate bugs once they have been found, but the task of finding bugs still requires scalable tools.

3.4.6 Testing When Proof is Unavailable

C/C++ models are often accompanied by compilation schemes. Compilation schemes are idealised mappings from C/C++ to assembly sequences that are proven sound under the source and architecture models. These mappings should not introduce concurrency bugs. Unfortunately, these mappings and their proofs become outdated as architectures are extended with new instructions. Compilers routinely use new mappings in the pursuit of performance, stepping outside the domains of earlier proofs. We use Téléchat [71] to test the correctness of a patch proposing a new mapping, which was subsequently accepted based on our results. We emphasise that there are limits to what can be achieved by compiler testing under models and the need for verification.

Following compelling performance metrics on hardware, engineers proposed to change the implementation of C/C++ acquire loads to use the `LDAPR` instruction instead of `LDAR` when the Armv8.3-A weak release consistency extension is enabled. The `LDAPR` instruction allows more re–orderings than `LDAR`; however experts failed to find a bug under this proposal. Reviewers were inclined to accept the proposal without a correctness proof, which was estimated to take three months.

Store Buffering Test

C/C++ Litmus Test

```
{ *x, *y = 0 }

P0 () {
  store(x,1,rel);
  int r0 = load(y,acq);
}

P1 () {
  store(y,1,rel);
  int r1 = load(x,acq);
}

exists (P0:r0=0 /\ P1:r1 = 0)
```

C/C++ Outcomes

```
{ P0:r0=0, P1:r1=0 }
{ P0:r0=0, P1:r1=1 }
{ P0:r0=1, P1:r1=0 }
{ P0:r0=1, P1:r1=1 }
```

Assembly Litmus Test

```
{ x = 0, y = 0 }

P0                 | P1
MOV W0,#1          | MOV W0,#1
STLR W0,[X%P0_x]   | STLR W0,[X%P1_y]
LDAR W3,[X%P0_y]   | LDAR W4,[X%P1_x]

exists (P0:W3=0 /\ P1:W4=0)
```

AArch64 Outcomes

```
{ P0:W3=0, P1:W4=1 }
{ P0:W3=1, P1:W4=0 }
{ P0:W3=1, P1:W4=1 }
```

Figure 3.11: (Top) Store Buffering litmus test with acquire release semantics. compiled to AArch64 test (Bottom) using STLR and LDAR pairs.

Example 3.4.5. Consider Figure 3.11 which shows a SB litmus test before and after compilation using LLVM. In this case the C/C++ store release operations are compiled to STLR instructions, and load acquires are compiled to LDAR. The AArch64 model [19] model forbids the outcome `{ P0:W3=0, P1:W4=0 }`, but if LDAR is replaced with LDAPR, then the outcome is observed. The C/C++ model permits the '0-0' outcome, so there is no bug.

This patch was time–sensitive and the LLVM reviewers were inclined to accept it without a correctness proof. We spent three days using Téléchat [71] to provide experimental confidence in the patch with several thousand litmus tests. We generated 5,200 C/C++ tests that exercise combinations of at least one acquire or consume load with other accesses. We used Téléchat to generate AArch64 litmus tests with clang before and after the patch was applied.

Our test results suggest that for the tests we checked, replacing LDAR with LDAPR is safe. We observed 96 tests that represent variants of the load buffering behaviour we covered in §5.5 these are present before and after LDAPR is used. The 121 tests reduce to 65 after the patch is applied, which implies LDAPR does not allow more outcomes than those allowed by the RC11 model. This reduction occurs since the LDAPR tests have the same outcomes as their C/C++ counterparts, increasing the number of tests with equal outcomes from 4,981 to 5,037. After seeing these results, Arm's compiler team decided to provisionally accept the patch and published a blog on the issue [153]. Of course litmus test generation is not exhaustive and a formal proof of mapping equivalence is needed. We are not aware of such a proof.

3.5 Discussion

We end this chapter by discussing questions concerning Téléchat.

Why does prior work miss bugs found by Téléchat? Prior work either computes outcomes using hardware or does not test compilation down to the assembly. It is possible to miss architecturally allowed outcomes if compiled program outcomes are computed using hardware. Techniques that do not test compilation down to assembly will miss bugs in target dependent optimisations.

Why is Telechat repeatable? Our test oracle is repeatable as it relies on simulators to compute program behaviour. This means our oracle computes the same source and compiled program outcomes every time.

What is the automation gap for prior work? Figure 3.2 describes the `diy`, `herd`, and `mcompare` tools, and shows how Téléchat completes the diagram. The `diy` tool can generate C/C++ litmus tests, but they are not syntactically valid programs. The `l2c` tool in the Téléchat toolchain turns `diy`-generated tests into syntactically valid programs. Likewise the `herd` tool required significant improvements to support compiler-generated

tests. We extended `herd` with several new instructions, a vector data type, and deployed testing of `herd` itself to catch regressions. Lastly, the `mcompare` tool could not compare the outcomes of programs written in different languages. We extended `mcompare` to accept state mappings to compare outcomes of programs written in different languages.

The state of the art techniques (Table 6.1) addressed similar gaps. The `cmmtest` and `validc` authors had to develop test generators and graph transformation techniques to compare program executions. The `c4` authors had to adapt the `litmus` tool to compiler testing.

Does the diversity of litmus tests limit bug finding ability? Yes, since litmus tests have very little diversity. Litmus tests focus on a finite set of concurrency idioms (see Figure 2.2) using a finite set of C/C++ atomic operations, fences, and dependencies. During simulation, each operation corresponds to a partial or totally ordered memory operation with synchronisation. The synchronisation, be it a fence or otherwise, either is or is not there in the program. This is the primary focus of litmus tests. Sufficiently large programs might induce or reveal bugs in atomic re-ordering, but they are beyond the scope of this work.

Could Gödel Testing and Genetic Programming help? Gödel testing could help improve our test coverage. If an LLM were to be equipped with a test generation framework, such as that of `Litmustestgen` [104] then the AI could interactively search for litmus test shapes that we have not yet put through the compiler.

Genetic programming could assist in fuzzing the compiler. One could mutate a seed set of litmus tests by transforming atomic operations into equivalent source operations. The challenge would be defining appropriate mutations that preserve the atomicity and memory order requirements of the source program.

Does the size of programs limit test coverage? Yes. Litmus tests are designed to test small concurrency patterns over accesses to memory. As such litmus tests typically contain only the reads, writes, and fences needed to induce unexpected outcomes of execution. Litmus tests will miss bugs in compiler optimisations applied to large programs. Our coverage is thus bounded to bugs in atomics mappings, and optimisations on those mappings, as observed when testing the compilation of small litmus tests.

Does the choice of predicate limit test coverage? Possibly. The predicates over the final states of litmus tests (as generated by `diy`) are designed to check for cyclic executions forbidden by the memory model under test. Litmus tests generated in this style can be considered *negative tests*. Such predicates restrict the outcomes of interest to the states affected by concurrent accesses to memory. It is possible that testing using litmus tests may miss bugs in other parts of the program state not probed by the test's predicate. It is further work to investigate the extent to which this occurs and its implications for coverage.

Chapter 4

Atomic-mixer: Mix Testing

This chapter concerns program interoperability. *Interoperable* programs do not exhibit bugs when mixed with others. *Mixing* occurs when programs such as linkers combine assembly programs produced by different compilers. The interoperability of assembly is specified using an *application binary interface (ABI)*, which defines calling conventions, error handling, and so on. Compilers should implement ABIs, and testing tools should check for compliance against the ABI. Unfortunately, there are no official concurrency ABIs, and no tools to test their interoperability. Consequently, concurrent assembly programs which are correct in isolation might exhibit bugs when mixed.

We present the *mix testing* technique which finds concurrency bugs when different parts of a C/C++ litmus test are translated by different compilers. The technique splits a C/C++ litmus test s into its statements; compiles each statement separately with different compiler profiles; and combines the resulting assembly into the set of assembly tests C. Each c in C is the result of compiling s with a different combination of profiles. We check for a specific kind of bug, called a *mixing bug*, which arises if any $c \in C$ exhibits a concurrency bug with respect to s. The `atomic-mixer` tool implements mix testing. We use `atomic-mixer` to reproduce an existing non-mixing bug and find four new mixing bugs. Additionally, we found and prevented a mixing bug from arising in mappings proposed for the Java Virtual Machine. Lastly, we worked with Arm's engineers to develop the Atomics ABI for Armv8 [69].

The rest of this chapter is structured as follows. §4.1 describes the mixing problem, §4.2 and §4.3 describe mix testing and the `atomic-mixer` tool, which we evaluate in §4.4. §4.5 describes the ABI, and we discuss mixing in §4.6.

4.1 The Mixing Problem

The question of program *interoperability* arises when mixing binaries produced with different mappings from C/C++ atomics to assembly sequences. The question is: can binaries produced by different mappings be mixed without inducing bugs in the final executable?

Concurrent program interoperability is less well studied than whole programs. There is a widely held belief that concurrent code should be compiled using one mapping, and mixing should not occur. Indeed, prior work [160, 71, 118, 41] exclusively tests the compilation of *whole* programs under one mapping. No guarantees are given if mappings are mixed together. However, the widespread use of dynamically linked shared objects makes some mixing all but inevitable in practice. When discussing the trailing and leading fence mappings for IBM PowerPC, Batty et al. [29] state that "*for code produced by different compilers to correctly interoperate, they must all make the same choice of mapping*". The academic state of the art is Sewell's mappings web page [144], which does not yet account for all mappings in today's compilers.

Mixing mappings does however occur in industry applications. Generally, mixing code is allowed for CPUs that can co-exist in the same shared-memory system, and mixing occurs in projects where portability is key. For instance, mixing MSVC and LLVM-generated code occurs on Windows on Arm where MSVC's C/C++ STL accesses [91] are mixed with LLVM's mappings. Likewise, the developers of Mono, a tool for creating portable applications, insert barriers [116] when mixing LLVM's and GCC's mappings for the Arm architecture. Kernel developers [48] resolve correctness issues associated with concurrency mixing via online discussions. The industry state of the art is largely contained in mailing lists, some of which have disappeared.

Unfortunately, concurrency bugs can arise when mixing. Such bugs (herein called *mixing bugs*) are concurrency bugs that arise only when compiled concurrent programs are composed. Mixing bugs can arise as architectures evolve, bringing with them new mappings from C/C++ to assembly. Without a concurrency ABI, there are no constraints on what compilers *should* do beyond those constraints imposed by the memory model. Without testing techniques, the complicated task of concurrency interoperability must be checked manually. Further, there are currently no link-time checks in LLVM or GCC to ensure that mappings are interoperable. Such a check would compare a program's mappings against accepted mappings and warn if new mappings are detected. As such, mappings are chosen based on what is best for an architecture in isolation or by a user proposing mappings that are best for their workloads. Today's compilers have mixing bugs as a result.

Example 4.1.1. The store-buffering test in Figure 4.1(a) has sequentially consistent [91] ordering. The outcome `{ P0:t=0, P1:u=0 }` is forbidden by the C/C++ memory model [80]. After compiling this whole program using `"clang -O3 -march=armv7-a"` (Figure 4.1(b)), the compiled program does not exhibit `{ P0:t=0, P1:u=0 }` under either the unofficial Armv7-A model or the newer AArch32 model [64]. This is because the `store` statements map to sequences that end with DMBs, which prevent reordering with the subsequent `load` (LDR). Compiling the whole program using `"clang -O3 -march=armv8"` (Figure 4.1(c)) does not expose the outcome under the Armv8 [64] model either. With this mapping, the `store` no longer has a trailing fence; instead, the store-to-load reordering is enforced by mapping the `load` to an acquire-load (LDA), which cannot be reordered with the store-release (STL).

However, the constituent operations of Figure 4.1(a) may be compiled for different (compatible) architectures and mixed together. For example, suppose we separately compile the `store` statements using `"-march=armv8"` and the `load` statements using `"-march=armv7-a"`, and combine the resulting binaries into a final executable This executable exhibits the unwanted outcome under

(a) The store-buffering C/C++ litmus test with `sc` ordering.

```
        { atomic_int *x = 0, *y = 0 }
P0 () {                  | P1 () {
 store(x,1,sc);          |  store(y,1,sc);
 int t = load(y,sc);     |  int u = load(x,sc);
}                        | }
exists (P0:t=0 /\ P1:u=0) // No.
```

(b) Using `"clang -march=armv7-a -O3"` introduces no bugs since the barriers ● preserve the store-to-load ordering.

```
P0                       | P1
// store(x,1,sc) ↴       | // store(y,1,sc) ↴
  MOV R1, #1             |   MOV R1, #1
  DMB ISH                |   DMB ISH
  STR R1, [x]            |   STR R1, [y]
● DMB ISH                | ● DMB ISH
// t = load(y,sc) ↴      | // u = load(x,sc) ↴
  LDR R0, [y]            |   LDR R0, [x]
  DMB ISH                |   DMB ISH
exists (P0:t=0 /\ P1:u=0) // No.
```

(c) Using `"clang -march=armv8 -O3"` is also ok, since the store-releases (◓) and the load-acquires (◒) preserve ordering.

```
P0                       | P1
// store(x,1,sc) ↴       | // store(y,1,sc) ↴
  MOV R1, #1             |   MOV R1, #1
◓ STL R1, [x]            | ◓ STL R1, [y]
// t = load(y,sc) ↴      | // u = load(x,sc) ↴
◒ LDA R0, [y]            | ◒ LDA R0, [x]
exists (P0:t=0 /\ P1:u=0) // No.
```

(d) Using `"clang -march=armv8 -O3"` to compile the stores and `"clang -march=armv7-a -O3"` to compile the loads reveals a mixing bug. The lone store-release (◓) is not sufficient to preserve ordering.

```
P0                       | P1
// store(x,1,sc) ↴       | // store(y,1,sc) ↴
  MOV R1, #1             |   MOV R1, #1
◓ STL R1, [x]            | ◓ STL R1, [y]
// t = load(y,sc) ↴      | // u = load(x,sc) ↴
  LDR R0, [y]            |   LDR R0, [x]
  DMB ISH                |   DMB ISH
exists (P0:t=0 /\ P1:u=0) // Yes.
```

Figure 4.1: Example of a mixing bug [62] that is missed by ordinary testing. State mapping $f = \{$ `P0:R0→P0:t, P1:R0→P1:u` $\}$.

the Armv8 model, because the Armv7-A mapping expects a barrier after the `store` that the Armv8 mapping does not provide, and the Armv8 mapping expects a load-acquire that the Armv7 mapping does not provide.

Mix testing takes a C/C++ litmus test and a set of compiler profiles and finds mixing bugs. Mix testing splits the litmus test into its statements, which are compiled separately using each profile, and each assembly sequence is then combined into one of *multiple* assembly litmus tests that represent combinations

of concurrency implementations of the original C/C++ test. Concurrency-related compiler bugs are then detected using the technique in Chapter 3. Mix testing demonstrates another challenge for concurrency compilation testing, that cannot simply be addressed by testing atomics mappings in isolation, but rather by strategically testing in the presence of exponentially many choices of mappings, both now and as architectures evolve.

We implement the technique in the `atomic-mixer` tool. Mix testing strictly generalises prior testing work with respect to a single compiler profile, which tests whole programs under one mapping at a time. We use `atomic-mixer` to discover four previously unknown mixing bugs in LLVM and GCC, one of which has been fixed, and the others confirmed and triaged for fixing by compiler engineers. We also use `atomic-mixer` to find a non-mixing bug found by prior work [71]. We found one of the mixing bugs [50] in GCC's `_Atomic struct` implementation manually, since we rely on the `herd` simulator, which does not support `struct`s. Lastly, we found a mixing bug in mappings proposed for the Java Virtual Machine, which did not make it to production code.

Significant work is required to reduce the test space of mix testing, since the number of compiled tests expands exponentially in the size of the input programs and the number of compiler profiles under test. Since our primary motivation was to specify an ABI, we develop an atomics ABI [69] for Armv8-A AArch64 with Arm's compiler teams. We specify mappings from C/C++ atomics to AArch64 assembly sequences and special cases that must be implemented to prevent mixing bugs. We use `atomic-mixer` to automatically validate ABI-compatibility of LLVM and GCC (modulo the bugs we found). As far as we know this is the industry's first open source specification of an atomics ABI with a tool to automatically check compatibility.

4.2 The Mix Testing Technique

Figure 4.2 details how the *mix testing* technique works. Given a C/C++ litmus test s, and a set P of compiler profiles under test, we produce a set C of

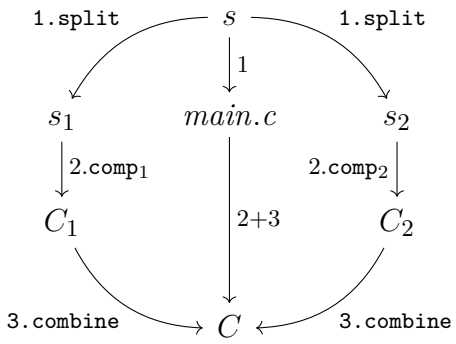

Figure 4.2: Mix testing technique.

compiled litmus tests. If any compiled litmus test $c \in C$ exhibits a concurrency bug (Def. 2.4.3) with respect to the source test s then there is a mixing bug.

4.2.1 Definition and Mix Test Notation

Mix testing is defined as the process of: splitting up s into its *statements*; compiling each statement separately using profiles; combining the resulting assembly sequences into the set of assembly litmus tests C; and checking whether any $c \in C$ exhibits a concurrency bug with respect to s. We now use the running example in Figure 4.1 to describe the technique.

First, we split Figure 4.1 into its statements using a *splitting function*:

Definition 4.2.1. *Splitting function.* Let s be a source litmus test, we define $split(s)$ to be the set of all statements of the test's program `prog`.

In this work we split a program by its simple (non-compound) statements (as defined in Figure 2.1). We split litmus test's program `prog` as follows:

- For each thread t in `prog`, get the statements *stmts* in t by case analysis on the structure of statements:

    1. For simple (non-compound) assignment statements, function calls, or RMW statements *stmt* in *stmts*, return *stmt*.
    2. For compound statements (sequences, conditionals, and iterators) recurse on the sub-statements until a simple statement is reached, return the set of all statements within the compound statement.

We split programs by statements since we are focusing on the mappings from simple atomic `load` and `store` statements to their assembly sequences. The splitting function determines the set of statements I to be compiled separately. There is a trade-off here: the finer-grained the split, the more opportunities there are for problematic interactions between compiler mappings, but the larger the search space of possible sequences. The simplest function does not split the source test at all and just tests compilation using each profile $p \in P$, that is $|I| = 1$. This corresponds to (non-mix) testing as conducted by prior work [160, 71, 118, 41]. Mix testing strictly generalises prior work, which compiles the whole program under one profile. Another choice of function splits each source test s into its constituent K threads and compiles those under different profiles; then we must test $|P|^{|K|}$ different threads. In this case a single compilation unit will be compiled with the same profile, but compiler and link-time optimisations create possibilities for mismatches even within a single thread. Such mismatches would not be caught by function that splits on whole threads. Intuitively many[1] concurrency bugs arise due to thread-local reordering [24, 118], so if each thread is compiled using one mapping and each mapping is correct in isolation, then no bugs should arise. We expect most bugs found by splitting at the thread boundary are caught by non-mix testing. Therefore, we split litmus tests at the statement level as shown in Figure 4.3 ($|I| = 4$ in this case), so that I is the number of statements of the input test. This splitting function offers a reasonable trade-off between the likelihood of finding bugs and the complexity of splitting the test into smaller fragments.

Next, each statement is compiled separately using the profiles. For example, the profile `"clang -march=armv7-a -O3"` compiles the `load` of `y` on `P0` in Figure 4.1(a) to the `"LDR;DMB"` sequence in Table 4.1. We assume that the compilers under test use *fixed* (compile time) mappings between C/C++ atomics and assembly sequences. We explore runtime mappings in §4.4.4.

Definition 4.2.2. *Compilation.* Let *stmts* be a set of source statements and

[1] with the exception bugs that arise under non-multi copy atomic models

Atomic Operation	Compiler Profile	Assembly Sequence
`load(loc,sc)`	`"clang -march=armv8 -O3"`	`LDA R0, [loc]`
	`"clang -march=armv7-a -O3"`	`LDR R0, [loc]` `DMB ISH`
`store(loc,val,sc)`	`"clang -march=armv8 -O3"`	`MOV R1, #val` `STL R1, [loc]`

	`"clang -march=armv7-a -O3"`	`MOV R1, #val`
		`DMB ISH`
		`STR R1, [loc]`
		`DMB ISH`

Table 4.1: Some of LLVM's sequentially consistent [91] mappings from C/C++ to Armv7-A and Armv8-A.

P be a set of compiler profiles. We define $\mathit{compile}(\mathit{stmts}, P)$ to be the set of assembly sequences produced by compiling each statement with each profile: $\mathit{compile}(\mathit{stmts}, P) = \{\, \mathit{comp}(\mathit{stmt}) \mid \mathit{stmt} \in \mathit{stmts},\, \mathit{comp} \in P \,\}$

Then we combine assembly sequences with the original program structure to produce compiled litmus tests using a *combining function* (Def. 4.2.3). In this step we compile the remaining compound statements that determine the sequencing, control flow, and iterator structure of the program. We do so by replacing each simple statement with a function call, leaving only compound statements that have yet to be compiled. We then compile the program and link against the assembly sequences generated in the previous step. The aforementioned function calls then branch to the assembly sequences that represent the compiled atomic statements. Combining the compiled sequences with the original program structure produces exponentially many assembly litmus tests in the number of compiler profiles and statements, all of which are valid combinations of the compiler profiles under test. We discuss how to prune this space of tests, to prioritise tests that are likely to be interesting, in §4.2.2.

Definition 4.2.3. *Combining function.* Let s be a source litmus test and *asms* be a set of assembly sequences. We define $\mathit{combine}(\mathit{asms}, s)$ to be the set of assembly litmus tests produced by linking the assembly sequences against the program structure, and copying over the (rewritten) initial and final state of s.

Concretely given a litmus test s and set of assembly sequences *asms* we

produce the final set of litmus tests as follows:

- Get the program structure *struct* by rewriting each thread t in the program `prog`. Rewrite by case analysis on the statements in each t:

  1. For simple statements, emit an external function call in place of the statement and record its positional information.
  2. For compound statements (sequences, conditionals, and iterators) recurse on the sub-statements until a simple statement is reached, emit a function call and return the new compound statement.

- Compile *struct* and link with each *asm* in *asms*, respecting the positional information of each statement. Get the set of compiled and linked assembly programs L.

- Get the initial state `init` and predicate over the final state `pred` from the test s and rewrite using the state mappings from the compiler. Get an assembly initial state `init'` and predicate `pred'`. Combine `init'` and `pred'` with each l in L to get the set of final litmus tests.

We make the same assumption as in the previous chapter, notably that registers have state mappings to global or shared variables. The mix testing technique applies splitting, compilation, and combining steps in sequence:

Definition 4.2.4. Mix Testing. Let s be a source litmus test and P be a set of compiler profiles. We define `atomic-mixer`(s, P) to be the set of assembly litmus tests produced by mix testing s with P:

$$\texttt{atomic-mixer}(s, P) = \mathit{combine}(\mathit{compile}(\mathit{split}(s), P), s)$$

Example 4.2.1. Mix testing is illustrated in Figure 4.3.

Each source test compiles to multiple assembly tests. We check if any c in the set of compiled litmus tests C exhibits a bug with respect to the source litmus test s. We can now define a *mixing bug*.

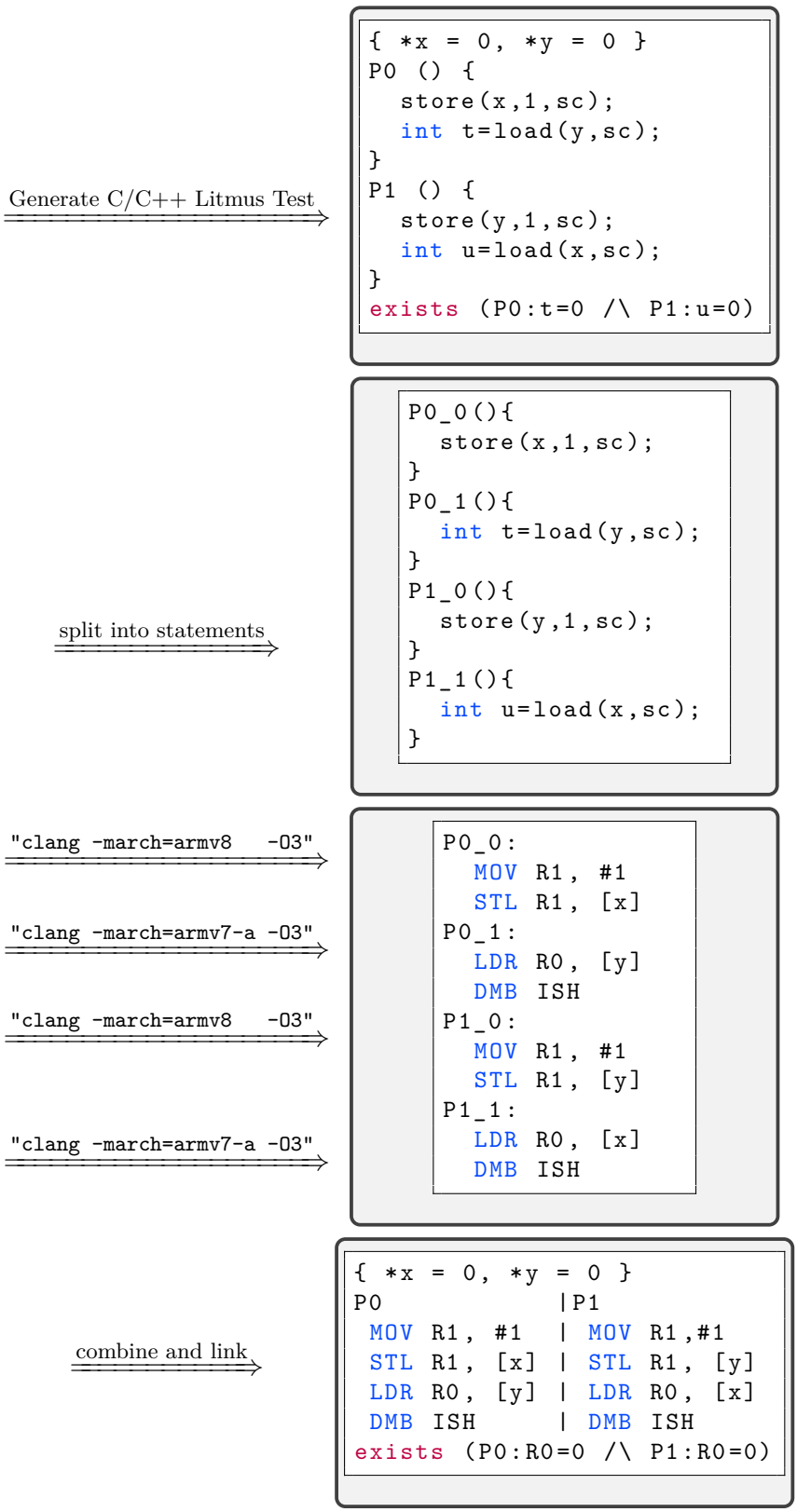

Figure 4.3: Mix testing Figure 4.1(a) produces multiple mix tests. Splitting produces multiple statements which are compiled separately as functions.

Definition 4.2.5. *Mixing bug*: For a well-defined concurrent source program s and its set C of compiled litmus tests:

$$MixingBug(s, C) = \exists c \in C. ConcurrencyBug(s, c) \qquad \text{(applies Def. 2.4.3)}$$

Since mix testing generates many litmus tests we introduce a *Mix test* notation to pinpoint tests that induce mixing bugs. A mix test is a record $\{\texttt{test}: LitmusTest_{src}, \texttt{assignment}: Set(Stmt) \rightarrow CompilerProfile\}$ consisting of a source litmus test that is partitioned into its statements (`{P0_0, P1_0,...}`) and an *assignment* of profiles to the statements they compile.

We only mix test compiled programs that are instruction-set architecture (ISA) *compatible*. This means their binary representation can be combined and executed without fault, as permitted by the envelope of the ISA. We do not yet require *atomics ABI-compatibility* in as far as we cannot find any official atomics ABIs outside what we contribute in §4.5, but we do require that compiled programs are ABI-compatible in every other way. This is why our combining function maintains the original program structure, testing only the mappings within. In general, mix-testing applies to code generated for CPUs of any architecture that can co-exist in the same shared-memory system, but for this work we limit our focus to a subset of recent Arm architectures.

Example 4.2.2. Mix testing the test in Figure 4.4 (top) produces Figure 4.4 (bottom). Figure 4.4 (bottom) arises when the operations of Figure 4.4 (top) are compiled for Armv8-A and Armv7-A. Running Figure 4.4 (bottom) under the Armv8 model produces the outcome `{ P0:R0=0, P1:R0=0 }`. The mixing bug occurs since the load of `y` on `P0` is compiled to the `"LDR;DMB"` sequence. Since the `LDR` instruction is missing a leading `DMB` barrier and it has no ordering semantics with respect to `STL`, it can reorder before the `STL` instruction on `P0`.

4.2.2 The Complexity of Mix Test Generation

The number of compiled mix tests we generate is exponential in the number of compiler profiles and number of program statements. Mix testing takes a set of S source litmus tests as input. Each $s \in S$ is split into a set of I program

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  store(x,1,sc);
  int t=load(y,sc);
}
P1 () {
  store(y,1,sc);
  int u=load(x,sc);
}

exists (P0:t=0 /\ P1:u=0)
```

Outcomes

```
{ P0:t=0, P1:u=1 }
{ P0:t=1, P1:u=0 }
{ P0:t=1, P1:u=1 }
```

{ test = Figure 4.4 (top),
 assignment = {
  P0_0 ↦ $comp_1$, P1_0 ↦ $comp_1$,
  P0_1 ↦ $comp_2$, P1_1 ↦ $comp_2$}}

where:

$comp_1$ = "clang -march=armv8-a -O3"
$comp_2$ = "clang -march=armv7-a -O3"

```
P0_0 = store(x,1,sc)
P0_1 = load(y,sc)
P1_0 = store(y,1,sc)
P1_1 = load(x,sc)
```

$comp_1$(P0_0) = "MOV;STL"
$comp_2$(P0_1) = "LDR;DMB"
$comp_1$(P1_0) = "MOV;STL"
$comp_2$(P1_1) = "LDR;DMB"

AArch64 Litmus Test

```
{ *x = 0, *y = 0 }

P0             |P1
 MOV R1, #1    | MOV R1,#1
 STL R1, [x]   | STL R1, [y]
 LDR R0, [y]   | LDR R0, [x]
 DMB ISH       | DMB ISH

exists (P0:R0=0 /\ P1:R0=0)
```

Outcomes

```
!!{ P0:t=0, P1:u=0 }!!
  { P0:t=0, P1:u=1 }
  { P0:t=1, P1:u=0 }
  { P0:t=1, P1:u=1 }
```

Figure 4.4: Example from Figure 4.1 using Mix test notation. State mappings = { P0:R0→P0:t, P1:R0→P1:u }

statements using a splitting function (Def. 4.2.1). Each $i \in I$ is compiled separately to each distinct assembly sequence for that statement produced by all the profiles p in the set of P compiler profiles. Each source litmus test then yields a set C of different compiled litmus tests, where $|C| = |P|^{|I|}$. For example, mix testing Figure 4.1 (a) yields $|P| = 2$, $|I| = 4$ that is $|C| = 16$ possible compiled tests. The number $|C|$ of tests for each $s \in S$ rapidly increases as the size of the input test increases. We therefore reduce P, S, and I as much

as possible whilst maximising code coverage. We do so by:

Curation of P: We omit compiler profiles that do not change the code generation of atomics relative to others. For instance `"clang -O1"`, `"-O2"`, and `"-O3"` use the same atomics mappings, but apply different optimisations. To maximise the chances of catching bugs we use `"-O3"`. We have worked with Arm's compiler experts to pick profiles that use different atomic mappings targeting Arm assembly. Of course, it is possible that certain optimisations will do different things to (different kinds of) dependencies and so a finer-grained study of the effect of optimisations on mappings is worthwhile. For now, such studies are out of scope.

Symmetry reduction on S: We do not generate source tests where the contents of each thread are simply swapped.

Bound $|I|$ by fixing the splitting function: The number of source statements is determined by the splitting function. By splitting litmus tests at the statement level we bound the number of generated tests. The number of compiled tests $|C|$ for each $s \in S$ is still exponential in I and P, and it is possible that duplicate tests exist in each set C. In the worst case when all compiler mappings in P are disjoint — that is a given C/C++ atomic operation compiles to a different instruction (sequence) for each profile $p \in P$ — the complexity is $|P|^{|I|}$ for each $s \in S$. In the best case when every compiler implements one set of mappings, we only need to test one compiler, and so $|P| = 1$ and the number of generated tests is $1^{|I|}$ or 1 for each $s \in S$. The best case rarely happens in practice, since a given architecture has multiple possible atomics mappings, for each architecture sub-version, and hence multiple compiler profiles to test. For example, LLVM implements atomics differently for at least Armv8, Armv8.1, Armv8.2, Armv8.3, and Armv8.4. Further, new atomic assembly instructions are announced with new architecture versions to improve performance of concurrent workloads. This means mix testing

the compilation of concurrent programs is unfortunately a practical necessity at least until everyone agrees on an ABI that specifies common atomics mappings. Lastly, compilers are routinely revised and the code they generate often changes. As such our analysis is not exhaustive or even timeless, and we must periodically revise P and S.

4.3 The `Atomic-mixer` Tool

We present the `atomic-mixer` tool that implements the mix testing technique. The `atomic-mixer` tool takes a source litmus test s and a set P of compiler profiles as input, and returns a set C of assembly litmus tests.

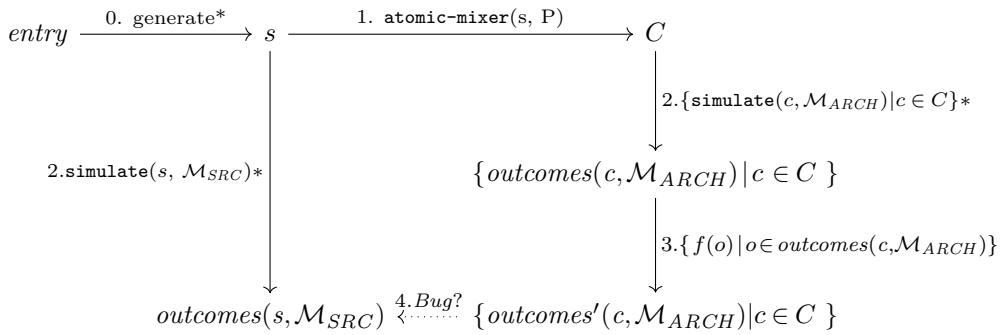

Figure 4.5: Mix testing implementation using the new `atomic-mixer` tool and prior work (marked $*$) [71, 12].

4.3.1 `Atomic-mixer` Tool Implementation

The `atomic-mixer` tool builds on the Téléchat toolchain, re-using a lot of the machinery, and so we refer the reader to the previous chapter for details. Figure 4.5 shows how we implement the mix testing technique. Given a source litmus test s and compiler profiles P, mix testing proceeds as follows:

1. Generate assembly mix tests C and the state mappings f for each $c \in C$.
2. Simulate s under a C/C++ model to get the set outcomes(s, $\mathcal{M}_{SRC}$). Simulate each $c \in C$ under its model to get outcomes(c, $\mathcal{M}_{ARCH}$).
3. Map f over outcomes(c, $\mathcal{M}_{ARCH}$) to get Set(outcomes$'(c, \mathcal{M}_{ARCH})$).
4. For each $c \in C$ check for mixing bugs (Def. 4.2.5) with respect to s.

We use the `diy` [15] test generator to produce the test s. We simulate source and compiled tests using the `herd` [12] simulator. We compare program outcomes using the `mcompare` tool [12]. The `atomic-mixer` tool itself extends the Téléchat toolchain [71], which handles the non-mix testing case by generating one compiled litmus test for each profile, increasing coverage by enumerating atomics mappings of multiple profiles. Provided prior work is kept up to date, `atomic-mixer` is independent of the precise details of any particular assembler or architecture simulator, as long as parsers exist to consume their output.

4.3.2 Challenges Faced During Implementation

The key to efficient mix testing is knowing the number of mappings of a given atomic operation. There are many different compilers (for example, GCC and LLVM) and their code generation may change at any time to support new architectures, new optimisations, or modifications of existing implementations. A naïve approach is to test all compiler releases for each architecture. Despite the theoretical possibility that each release implements entirely different code generation, the reality is that few changes to atomics occur in practice. We therefore look for changes in code generation and only test each variant once. We describe some of the challenges we faced and how we address them.

Simulation penalty: Simulating behaviour under models is expensive. We must simulate each source program s to collect its behaviours. We must also simulate every $c \in C$. Our goal is thus to reduce the number and size of the set C for each s whilst increasing the coverage of code generated by compilers. We do so by hashing the generated assembly code.

We group C by hashes and check one representative of each group. It is possible that changing the compiler profile only changes one or two atomics mappings whilst other mappings remain unchanged. For example Armv8.3-A changes the mapping of acquire loads to use the `LDAPR` instruction instead of the existing Armv8 `LDAR` instruction. Since all other atomics remain unchanged many compiled programs are very likely to

have the same static hash and behaviour under simulation. Only one program with a given hash needs to be simulated in a group. We compute hashes using the `mshowhashes` tool [12]. By doing so we only need to simulate one test from each group of tests with the same hash.

I/O bound on mix testing: Even if we can discard duplicates using hashing, `atomic-mixer` must still *generate* them. `atomic-mixer` must generate all $c \in C$ as `mshowhashes` cannot compute the hashes until it has tests. One can remedy this by tracking changes to particular mappings. The statements used in litmus tests are reused a lot from test one to another - especially in `diy`-generated tests. One could exploit this by simply checking whether the compilation of individual statements varies with profiles and compiler versions, rather than checking the hash of the final tests. Implementing this optimisation is further work.

The branching problem: Splitting a test introduces branch and return instructions. A compiled litmus test has a branch instruction to the code that is separately compiled. For instance, LLVM generates *branch-with-link* (`BL`) and *return* (`RET`) instructions when targeting AArch64. This can be problematic if processors implement the branch using a control-flow dependency that constrains the order of execution. Since we are looking for reordering bugs, we do not want to introduce such constraints. For small tests, adding branches does not impede `herd` performance.

Example 4.3.1. Figure 4.6 shows an AArch64 load buffering litmus test, where each access on `P0` is replaced by a `BL` instruction to a function that does a load or store. The AArch64 model permits the outcome `{ P0:X0=1, P1:X0=1 }` and hence allows re-ordering across unconditional branches. This means the effects of instructions after each branch can reorder before events of instructions prior. For instance, in Figure 4.6 the effects of executing the `STR` in branch `P0_func_2` can reorder before the effects of executing the `LDR` in `P0_func_1`, enabling the outcome `{ P0:X0=1, P1:X0=1 }`.

Litmus Test AArch64 Outcomes

```
{ *x = 0, *y = 0 }
P0                | P1
  MOV X2, #1      |   MOV X2, #1
  BL P0_func_1    |   LDR X0, [%y]
  BL P0_func_2    |   DMB ISH
  B end           |   STR X2, [%x]    { P0:X0=0, P1:X0=0 }
P0_func_1:        |   DMB ISH         { P0:X0=0, P1:X0=1 }
  LDR X0, [%x]    |                   { P0:X0=1, P1:X0=0 }
  RET             |                   { P0:X0=1, P1:X0=1 }
P0_func_2:        |
  STR X2, [%y]    |
  RET             |
end:              |
exists (P0:X0 = 1 /\ P1:X0 = 1)
```

Figure 4.6: AArch64 LB test where C/C++ relaxed loads are compiled to branch instructions on `P0` and outcomes under the AArch64 model [19].

4.4 Evaluation

We evaluate the mix testing technique and `atomic-mixer` tool by conducting a number of case studies using LLVM and GCC. We use `atomic-mixer` to reproduce a non-mixing bug found by prior work (§4.4.1), discover previously unknown mixing bugs (§4.4.2) of which one was found manually. We conduct a bug finding campaign, unearthing four new mixing bugs (§4.4.3), and found a mixing bug in mappings proposed for the JVM (§4.4.4).

4.4.1 Reproducing a non-mixing Bug

Since mix testing with one profile corresponds to non-mix testing it follows that `atomic-mixer` should be able to reproduce existing bugs. We reproduce a (non-mixing) bug using `atomic-mixer` from the previous chapter (Figure 3.1).

Example 4.4.1. Figure 4.7 (top) shows a C/C++ message passing litmus test and its outcomes. The outcome `{ P1:r0=0, y=2 }` is forbidden by the RC11 model [46]. Figure 4.7 (middle) describes the mix test that gives rise to this concurrency bug. Note how there is only one compiler profile in Figure 4.7 (middle). The compiled program in Figure 4.7 (bottom) exhibits the outcome.

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  store(x,1,rlx);
  fence(rel);
  store(y,1,rlx);
}
P1 () {
  exchange(y,2,rel);
  fence(acq);
  int r0 = load(x,rlx);
}

exists (P1:r0=0 /\ y=2)
```

C/C++ Outcomes

```
{ P1:r0=0, y=1 }
{ P1:r0=1, y=1 }
{ P1:r0=1, y=2 }
```

{ test = Figure 4.7 (top),
assignment = {
all ↦ *comp*}},
where:
comp = "clang -march=armv8.2-a -O3"

```
P0_0 = store(x,1,rlx)
P0_1 = fence(rel)
P0_2 = store(y,1,rlx)
P1_0 = exchange(y,2,rel)
P1_1 = fence(acq)
P1_2 = int r0=load(x,rlx)

comp(P0_0) = "MOV;STR"
comp(P0_1) = "DMB ISH"
comp(P0_2) = "STR"
comp(P1_0) = "MOV;SWPL"
comp(P1_1) = "DMB ISHLD"
comp(P1_2) = "LDR"
```

AArch64 Litmus Test

```
{ *x = 0, *y = 0 }

P0              |P1
MOV W9,#1       |MOV W9,#2
STR W9,[X%P0_x]|SWPL W9,WZR,[X%P1_y]
DMB ISH         |DMB ISHLD
STR W9,[X%P0_y]|LDR W8,[X%P1_x]

exists (P1:W8=0 /\ y=2)
```

AArch64 Outcomes

```
{ P1:W8=0, y=1 }
!!{ P1:W8=0, y=2 }!!
{ P1:W8=1, y=1 }
{ P1:W8=1, y=2 }
```

Figure 4.7: `atomic-mixer` finds a non-mixing bug [60]. State mappings = { `P1:W8→P1:r0, y→y` }.

4.4.2 Finding Bugs the State of the Art Cannot

Mix testing can find bugs that current tools cannot, since they require mixing and are thus out of scope for those tools. We checked a compiler patch and

found and reported [59] a mixing bug that was missed when we tested it back in January 2023. This example highlights the difficulty of testing the compilation of concurrency as a problem that cannot be addressed by testing atomics mappings in isolation, but rather by strategic testing in the presence of exponentially many choices of mappings.

Example 4.4.2. The Store Buffering (SB) litmus test in Figure 4.8 (top) forbids the outcome `{ P0:r0=0, P1:r0=0 }` under the C/C++ model [80]. Figure 4.8 (middle) shows the mix test case that exposes the bug when mix testing `_Atomic __int128` (i128) accesses using profiles targeting Armv8-A and Armv8.4-A. Figure 4.8 (bottom) shows the mix test generated by `atomic-mixer`. In this case the load pair (`LDP`) of `x` on `P1` has no leading barrier, and since `LDP` has no ordering semantics, its effects can be reordered before the store-release exclusive pair (`STLXP`) on `P1`. The compiled program exhibits the outcome `{ P0:r0=0, P1:r0=0 }` under the AArch64 model [19].

4.4.3 Bug-Finding Campaign

We found three mixing bugs automatically [62, 63, 59], and one bug manually [50]. We summarise them here, and describe one more.

1. 32-bit SC load is missing a barrier: See Figure 4.1 and report [62].
2. 64-bit SC load is missing a barrier: See report [63]. An analogue of (1), but for 64-bit loads when compiling to target 32-bit systems.
3. 128-bit SC load is missing a barrier: See report [59] and Figure 4.8.
4. `_Atomic` struct size and alignment differ between LLVM and GCC. See report [50] and Figure 4.9.

Each bug is triggered by a different tests and profiles. These tests were found using variants of store buffering tests with either sequentially consistent stores or read-modify-write operations. Picking the size of accesses from 32, 64,

128-bit Store Buffering Example

C/C++ Litmus Test

```
{ i128 *x, *y = 0 }

P0 () {
  store(x,1,sc);
  i128 r0=load(y,sc);
}
P1 () {
  store(y,1,sc);
  i128 r0=load(x,sc);
}

exists (P0:r0=0 /\ P1:r0 = 0)
```

C/C++ Outcomes

```
{ P0:r0=0, P1:r0=1 }
{ P0:r0=1, P1:r0=0 }
{ P0:r0=1, P1:r0=1 }
```

{ test = Figure 4.8,
assignment = map }
where:
map={P0_0 ↦ $comp_1$,P0_1 ↦ $comp_1$,
P1_1 ↦ $comp_1$,P1_0 ↦ $comp_2$}
$comp_1$="clang -march=armv8.4-a -O3"
$comp_2$="clang -march=armv8-a -O3"

```
P0_0 = store(x,1,sc)
P0_1 = i128 r0 = load(y,sc)
P1_0 = store(y,1,sc)
P1_1 = i128 r0 = load(x,sc)

comp1(P0_0) = "MOV;DMB;STP;DMB"
comp1(P0_1) = "LDP;DMB"
comp2(P1_0) = "MOV;LDAXP;...;CBNZ"
comp1(P1_1) = "LDP;DMB"
```

AArch64 Litmus Test

```
{ *x = 0, *y = 0 }

P0                  |P1
MOV X2, #1          |MOV X6, #1
DMB ISH             |loop:LDAXP X1,X2,[%P1_y]
STP X1,X2,[%P0_x]   |STLXP W4,X5,X6,[%P1_y]
DMB ISH             |CBNZ W4, loop
LDP X4,X0,[%P0_y]   |LDP X4, X0, [%P1_x]
DMB ISH             |DMB ISH

exists (P0:X0 = 0 /\ P1:X0 = 0)
```

AArch64 Outcomes

```
!!{P0:X0=0,P1:X0=0}!!
{P0:X0=0,P1:X0=1}
{P0:X0=1,P1:X0=0}
{P0:X0=1,P1:X0=1}
```

Figure 4.8: Mixing bug [59].

or 128-bits triggers different code paths in GCC and LLVM that are triggered by different compiler profiles. In other words we found three unique bugs.

The last bug [50] we describe was discovered manually while developing the ABI in §4.5. Manual effort was required since `atomic-mixer` depends on `herd`, which doesn't support `structs`. This bug arises when two different compilers translate code for the same ISA. We discovered that GCC and LLVM

```
typedef _Atomic struct { char a[5]; } X;
int size_x = sizeof(X); // GCC=5, LLVM=8
int align_x = __alignof(X); // GCC= 1, LLVM=8

X f(X *p) { return *p; }
// GCC=bl __atomic_load, LLVM=LDAR X2, [loc]

X g(X *p[2]) { return *p[1]; }
// GCC=__atomic_load, LLVM=LDR X1, [loc,#8]; LDAR X2, [X1]
```

Figure 4.9: Mixing struct implementations. This issue also effects x86 backends.

have incompatible implementations of `_Atomic structs`. Both the size and alignment requirement calculated in Figure 4.9 differs between compilers. The `sizeof` operator is used to determine the storage allocation and size of atomic assembly instructions to be used. In this case GCC's engineers chose to use an inefficient locking function (`__atomic_load`), whereas LLVM's engineers used a load acquire (LDAR) instruction. GCC's engineers chose to use the locking call, since there aren't any instructions to handle unaligned atomics or oddly-sized types, LLVM's engineers chose to use LDAR on the basis that every other access would share the same alignment values. Mixing code generated by both is problematic since LLVM may write struct padding bits to memory where GCC allocates entirely unrelated data — mixing LLVM and GCC code can invalidate data and hence program execution in unknown ways. The solution is to over align and pad atomic types to the next supported atomic size.

It is reasonable to question whether mixing bugs only arise when mixing acquire-release and barrier-based implementations. We now explore a mixing bug that does not require barriers.

4.4.4 Mixing Bugs in Proposed Mappings

Arm's engineers were considering a proposal to change the default mappings (Table 4.2) of SC [91] loads and stores when compiling for the release consistency processor consistency extension (RCPC). The RCPC extension introduces the LDAPR instruction whose effects can reorder before prior store-release (STLR) instructions that access different memory locations. The LDAPR instruction has the potential [153] to improve performance over the current LDAR instruction.

Atomic	Compiler Profile	Assembly
load(loc,sc)	"clang -march=current -O3"	LDAR W0, [loc]
	"clang -march=proposed -O3"	LDAPR W0, [loc]
store(loc,val,sc)	"clang -march=current -O3"	MOV W1, #val STLR W1, [loc]
	"clang -march=proposed -O3"	MOV W1, #val STLR W1, [loc] DMB ISH

Table 4.2: The proposal relaxed SC loads and strengthened SC stores.

Replacing LDAR with LDAPR alone however is unsound since it can reorder with prior stores (STLR). The change proposed to both strengthen the STLR with a trailing barrier (DMB ISH), and relax loads to use LDAPR, effectively preventing non-mixing bugs. We apply mix testing to show that this case would not be correct when mixing the proposed mappings in with code targeting Armv8-A.

Example 4.4.3. Figure 4.10 (top) shows a store buffering test. The outcome { P0:r0=0, P1:r0=0 } is forbidden by the C/C++ model [80]. Figure 4.10 (middle) shows the mix test case that arises when using the mappings in Table 4.2. Figure 4.10 (bottom) shows the mix test we found manually as no compiler implements the proposed mappings without which `atomic-mixer` cannot work. In Figure 4.10 (bottom) the store-release (STLR) on P0 has no trailing barrier, and the effects of executing the LDAPR can be reordered before the effects of the STLR. Figure 4.10 (bottom) exhibits the outcome { P0:r0=0, P1:r0=0 } under AArch64 [19].

There is nothing wrong with the proposed mappings, provided *all* stores are strengthened. In general, we cannot know if a compilation unit will be mixed with other code for different (yet compatible) architectures. The user can either guarantee the whole program is *always* compiled using the proposed mappings or otherwise every compiler implementation must change. This requires that every compiler that supports Armv8-A and above (including LLVM, GCC, and MSVC) strengthens their SC stores with a trailing barrier. Unfortunately such a wide-reaching change is unlikely to be accepted in practice. This proposal constitutes an ABI break with respect to today's compilers.

Prospective Store Buffering JVM Mixing Bug

Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  store(x,1,sc);
  int r0=load(y,sc);
}
P1 () {
  store(y,1,sc);
  int r0=load(x,sc);
}

exists (P0:r0=0 /\ P1:r0=0)
```

C/C++ Outcomes

```
{ P0:r0=0, P1:r0=1 }
{ P0:r0=1, P1:r0=0 }
{ P0:r0=1, P1:r0=1 }
```

{ test = Figure 4.10,
assignment = map }
where:
map={P0_0 ↦ $comp_1$,P1_0 ↦ $comp_1$,
P0_1 ↦ $comp_2$,P1_1 ↦ $comp_2$}
$comp_1$="clang -march=current -O3"
$comp_2$="clang -march=proposed -O3"

```
P0_0 = store(x,1,sc)
P0_1 = int r0 = load(y,sc)
P1_0 = store(y,1,sc)
P1_1 = int r0 = load(x,sc)

comp1(P0_0) = "MOV;STLR"
comp2(P0_1) = "LDAPR"
comp1(P1_0) = "MOV;STLR"
comp2(P1_1) = "LDAPR"
```

Litmus Test

```
{ *x = 0, *y = 0 }

 P0                 | P1
 MOV W1, #1         | MOV W1, #1
 STLR W1,[%P0_x]    | STLR W1,[%P1_y]
 LDAPR W0,[%P0_y]   | LDAPR W0,[%P1_x]

exists (P0:W0=0 /\ P1:W0=0)
```

AArch64 Outcomes

```
!!{ P0:W0=0, P1:W0=0 }!!
  { P0:W0=0, P1:W0=1 }
  { P0:W0=1, P1:W0=0 }
  { P0:W0=1, P1:W0=1 }
```

Figure 4.10: `{ P0:r0=0, P1:r0=0 }` is forbidden by the C/C++ model [80]. Mappings=`{ P0:r0→P0:W0, P1:r0→P1:W0 }`

It is possible to use the proposed mappings without mixing bugs. The mappings were proposed as a change to the Java Virtual Machine (JVM) implementation. When used in isolation these mappings are sound, since the JVM uses a JIT compiler that can dynamically generate code using the proposed mappings all at once. However, there are three cases where mixing bugs can arise. Firstly, heap locations may be written to by the JVM's C++ code using

SC atomics (the STLR instruction), but then later read by Java volatiles (using LDAPR). This can be fixed by inserting barriers after every C/C++ store in the JVM source. Secondly, a user's C++ code may share a memory buffer with Java code (for example, a `java.nio.ByteBuffer`), where the C/C++ code stores to the buffer and Java loads from it (using `VarHandle::getVolatile`). Again, the user must insert barriers after C/C++ stores. Thirdly, bugs may arise if the JVM interacts with C/C++ through foreign function interfaces (FFI) such as JNI (for instance using an API call to `SetIntField`). Assuming the JVM does not synchronize at FFI boundaries (see §4.3.2), barriers must be added here too.

4.5 Mix Testing an Atomics ABI

In this section we cover our experience applying mix testing in industry. We worked closely with Arm's compiler teams to develop an *atomics application binary interface* (ABI) that specifies the mappings of source-level atomics into AArch64 assembly sequences. As far as we know this is the industry's first published specification of an atomics ABI. We summarise the ABI as it is today (the ABI is in Appendix C), and refer the reader to the published document for updates [69].

4.5.1 An ABI Specification of Armv8 Atomics

The ABI is defined by a list of atomics mappings (§4.5.1.1), accessed through compiler profiles that generate assembly sequences. A compiler is *ABI compatible* if it implements the mappings in the ABI (§4.5.1.2), which means it should not exhibit mixing bugs when mixing with mappings in the ABI (§4.5.1.3).

4.5.1.1 Listing Atomics Mappings

We test atomics mappings produced by the compiler profiles in LLVM and GCC that use `"-march=armv8+{lse|rcpc|rcpc3|lse128}/armv8.4-a"`. Since architecture sub-versions such as `"-march=armv8.1-a"` imply a few of these flags they are omitted.

Example 4.5.1. Table 4.3 shows how a 32-bit integer exchange maps to either a CAS sequence when Armv8.0 is selected or a swap instruction (SWP) if the Armv8.1-A is selected.

Atomic Operation	Compiler Profile	Assembly Sequence
exchange(loc,val,rel)	Base "-march=armv8"	MOV W2, #val lbl:LDXR W4,[loc] STLXR W3, W2,[loc] CBNZ W3, lbl
	"+lse"	MOV W2, #val SWPL W2, W4,[loc]

Table 4.3: An exchange maps to a compare-and-swap loop or a SWPL instruction.

4.5.1.2 Statement of ABI Compatibility

We define a statement of compatibility with respect to compilers and their mappings against which we can test compatibility.

Definition 4.5.1. *ABI-compatibility A compiler that implements the stated mappings is ABI-Compatible.*

We can test compatibility up to bounds using `atomic-mixer`. Given a compiler *comp* that implements mappings under a set of compiler profiles P, and a C/C++ litmus test set S, *comp* is compatible with respect to S if mix testing using S and P finds no mixing bugs (Def. 4.2.5). This definition comes with the constraint that this is not a correctness guarantee, but rather a statement backed up by bounded testing. We test programs with a fixed initial state, loop unroll factor, and no recursion. The ABI does not make any statement about the compatibility of compilers outside the test bounds specified, the provided mappings are not exhaustive, the document makes no statement about the compatibility of optimised programs or concerning the performance of compiled programs under the provided mappings.

4.5.1.3 Mix testing mappings in the ABI

We generate a number of concurrency tests, checking ABI compatibility of compilers that implement the mappings in the document. At the time of writing the mixing bugs reported in GCC are fixed, but not in LLVM.

By following the steps in §4.3.1 we mix test LLVM and GCC given compiler profiles and tests as input. We generate tests that involve patterns of C/C++ atomic operations, memory order parameters, barriers, control-flow and straight line code up to 5 threads in size[2]. These tests are not exhaustive but aim to test atomic operations introduced in C/C++11. The ABI specifies mappings for C11 atomic operations for 8, 16, 32, 64, and 128-bit width accesses for both signed and unsigned integer types. Table 4.4 defines the test parameters.

C/C++ constructs:	`(atomic operations | non-atomics | barriers | control-flow | straight-line code)+`
Access width/sign:	`(u)int(8 | 16 | 32 | 64)_t`
Memory Order:	`(rlx | acq | rel | acq_rel | sc)+`
Target Architecture:	`(armv8 | armv8+lse | armv8+rcpc | armv8+rcpc3 | armv8+lse128 | armv8.4-a)+`

Table 4.4: Mix testing atomics implementations.

We generated thousands of C/C++ litmus tests using `diy` [12] and applied `atomic-mixer` to get millions of AArch64 assembly litmus tests. We used `mshowhashes` to remove redundant compiled tests (see §4.3.2) and `herd` [12] to search for mixing bugs. We parallelised [150] mix testing (with load balancing to reduce swap usage) on a 224 core ThunderX2 using 100 GB runtime footprint and found no mixing bugs besides those we document in §4.4.2. We do not auto-generate tests for all mappings in the ABI, since the `diy` [12] generator does not support all read-modify-write operations, such as `fetch_add`. We manually constructed tests with unsupported operations and applied `atomic-mixer` to check for mixing bugs in these cases.

There are many implementations of a given atomic operation in practice. Considering only the Armv8-A AArch64 backends of LLVM and GCC, there are up to 5 different mappings for each primitive, but many primitives also have mappings for each size and signedness. In addition, individual mappings were changed in compiler patches, but the changes were not consistently applied. As a result, LLVM and GCC are not currently interoperable, but specifying the ABI

[2]We reused tests from Chapter 3, testing with more instructions per thread is future work.

is one step towards addressing this. Altogether, the ABI specification fills over seventeen A4 pages, even with as much duplication removed as possible. Beyond this, there are also mappings used by proprietary compilers such as MSVC, which we have not yet considered. We expect that compiler implementations for other architectures have the potential for ABI mixing bugs too.

4.5.2 Special Cases

We detail two special cases that compilers should handle. Details of these bugs are in Chapter 3.

Read-modify-write should preserve read: (Example 3.1) Exchange can map to SWPL instructions (Table 4.3). However, according to the Arm Architecture Reference Manual [18] "*instructions where the destination register is WZR or XZR, are not regarded as doing a read for the purpose of a DMB LD barrier*". The bug in Figure 4.7 arises since the effects of executing a SWPL may be reordered past the acquire fence, we propose that compilers do not rewrite the destination register to be the zero register (WZR) in this case. This also applies to LD<OP> and CAS.

Mutable const-qualified data: (Example 5.2) Registers in AArch64 state hold 64-bit values. To load 128 bits atomically we must use a CAS loop (see Table 4.5) when Armv8.0 is selected. If `const`-qualified memory is marked read-only (and stored in, for example, `.rodata`) then executing the store-exclusive pair (STXP) instruction will crash the program. We propose that implementations mark `const`-qualified atomic locations as mutable This also affects x86 code generation of 64-bit access.

Atomic Operation	Compiler Profile	Assembly Sequence

`load(loc,rlx)`	Base `"-march=armv8"`	`lbl:LDXP X9,X10,[loc]`
		`STXP W3,X9,X10,[loc]`
		`CBNZ W3,lbl`

	`"+lse"`	`LDP W2,W4,[loc]`

Table 4.5: Some mappings for an 128-bit atomic load, in this case a CAS loop or an LDP instruction.

4.5.3 Sub-ABIs, ABI-Islands, and the Baseline ABI

There is no one 'true' ABI, but rather a specification that serves most purposes. The ABI we provide represents a *baseline* specification for any implementation that aspires to be compatible across all versions of the Armv8 architecture. Ideally, mainstream implementations such as LLVM and GCC will adhere to this ABI in the future. This ABI does *not* prevent implementors from creating their own ABI, whether it is a subset of the baseline (a *sub-ABI*) or an altogether different set of mappings (a disjoint *ABI-island*). A sub-ABI could induce mixing bugs on unsupported architectures (like in §4.4.4) and it would be up to the user of that sub-ABI to ensure such a situation cannot arise. Likewise, implementors may rely on an entirely different set of mappings that are disjoint from the baseline specification. Such an ABI-island would require similar restrictions to ensure correct execution. All ABI variants are of course relative to a baseline existing in the first place.

We observed that the absence of an explicit baseline led to the definition of implicit sub-ABIs. As architecture extensions (*ie* fast new instructions) are introduced, users quickly identify prospective mappings that offer performance improvements for their workloads. These sub-ABIs guide compiler development as they arise, but lacked ABI specification and testing until now. We provide a baseline ABI as guidance for numerous reasons. Firstly, mixing bugs have been introduced by accident (§4.4.2) and we want to prevent this as much as possible. Secondly, there have been numerous attempts to optimise special atomic sequences (see §4.5.2), motivating the need to collect these cases together. Thirdly, engineers have been asked whether the same set of prospective mappings is correct by multiple different partners, and writing down the known cases helps rule out incorrect mappings. Lastly, the collective knowledge of atomics ABIs exists as a series of online discussions and web pages [144], which do not yet contain all compiler mappings or have altogether disappeared (for instance when LLVM migrated from Phabricator to GitHub). We provide an ABI to help engineers and reduce the chances of mixing bugs arising in the future.

4.6 Discussion

We end this chapter by discussing questions concerning mix testing.

Could one reduce the complexity of mix testing? Yes. In §4.3.2 we discuss how to group tests by a static hash to reduce the number of tests we must simulate. A further optimisation is to regenerate test groups only for the mappings that actually change between compiler revisions. We leave this optimisation to further work.

Is it sufficient to check one test per hash group? Litmus tests are simple programs with a highly uniform structure. While there is no proof that testing one candidate per group is sufficient, it is exceedingly unlikely that hashes will collide with modern hashing technologies.

Could interaction testing help mix testing? Perhaps. Since mix testing relies on varying profiles and test statements, applying n-way testing to these parameters may prune redundant combinations. For instance, if a bug arises with a relaxed atomic mapping, then it may be unnecessary to test a stronger release mapping. However, this optimisation is not universal as a compiler may use two unrelated mappings, necessitating separate testing. This is to say our test parameters are not orthogonal.

Is there data about historic changes to compilers? There is some public information about the changes to compiler mappings[3] however a complete study is not in the scope of this work. We expect a study would need to address the following problems:

Bug fixes can be back-ported to older compilers to support software that relies on older toolchains. For example, the bug in §4.4.1 has been fixed in toolchains dating back to LLVM version 11. This means the accuracy of a study on the history of compiler correctness is subject to a compiler's mappings, and `Atomic-mixer` would not unearth historic behaviour in

[3]Consider the change log of Sewell's web page of mappings [144].

this case. When back-ports mask historic behaviour, one would need to study other sources of documentation to recover data.

Unfortunately, a lot of documentation is now inaccessible. For example, LLVM's Phabricator instance, that contained many discussions concerning compiler bugs, is now deprecated and is no longer accessible. As such manually reproducing a history of mixing bugs in compilers would require considerable effort and may miss data.

Are there mixing bugs in older compilers? Yes, and this is how we encountered mixing as a problem. In the past, a patch was merged into LLVM but testing it with Téléchat did not yield any bugs. A partner of Arm then asked if mixing the mappings from the patch with older mappings would induce bugs. After studying the patch in the context of mixing, we realised the patch *did* exhibit a mixing bug. This example is documented in §4.4.2. This motivated the development of the mix testing technique and subsequent analysis that became Chapter 4.

Are there bugs in compilers with respect to the ABI? Yes at the time of writing the mixing bugs reported in GCC are fixed, but not in LLVM. Further, we do not support testing of JIT compilers and §4.4.4 describes a case where we had to mix test the JVM manually.

Does the branching problem affect all architectures? For most modern architectures, unconditional branch and return instructions have no effect on the ordering semantics of memory accesses, and as such do not constrain the simulation of litmus tests. For architectures where a branch or return has memory side effects the order of execution may be constrained. For instance if a branch pushes the return address onto a stack and the return pops it off, then the order of execution may be affected. We did not test the branching problem on any architectures other than AArch64, but it would be valid to use unconditional branch and return sequences in the reassembled litmus test.

Chapter 5

On The Limitations of Models and Simulators

In this chapter we assess the limitations of today's models when used as *parts* of our compiler testing oracle. In our oracle, we use executable models and simulators to compute the expected and actual program behaviour (Def. 2.4.5). While we found nine bugs, which compares favourably with other concurrency testing work, it is perhaps less than one would expect from a thesis on compiler testing. The question arises how *sound* and *complete* are these parts?

We test the limits of model implementations and simulators. We find bugs in unsound models, compiler bugs missed by incomplete models, and behaviours otherwise overlooked by axiomatic model simulators. Such behaviours arise at the intersection of concurrent program semantics and other language features, such as `const` and `atomic` types. The examples we describe concern day-to-day issues for engineers, but are often not studied as they exist at the boundaries of what is tested. As such, using state of the art model simulators to study them required more work than we had expected. We conclude that further development of models and simulators is necessary if concurrency bug finding is to be automated to the degree that sequential compiler testing has been.

The remainder of this chapter is structured as follows. §5.1 revisits the oracle problem and §5.2-5.6 describe examples where testing under models and simulators is unsound or incomplete.

5.1 The Oracle Problem Revisited

We use simulators as parts of *oracles* [83, 82] for compiler testing. Oracles are tools that take a litmus test and compute, with respect to a specific model, what behaviour it allows (Def. 2.4.2). For our purposes, the `herd` simulator, paired with the C/C++ model produces the expected program outcomes, and `herd` paired with the architecture model produces the actual program behaviour (Def. 2.4.5). Comparing the outcomes from these parts of our oracle should be sufficient to find bugs.

Unfortunately, there are two cases where our oracle can still miss compiler bugs. These cases arise if the model or simulator are unsound, or when they do not implement the complete set of language features under test. To be clear we are focusing on executable models and simulators such as Cat models and the `herd` simulator, and *not* a model as it appears in a C/C++ standards document or an architecture specification. In this chapter we assess the limitations of models and their simulators with respect to compiler testing.

A model or simulator is *unsound*[1] when it forbids an implementation behaviour that should be allowed, or vice versa. In other words, one can use a simulator to show an outcome is reachable when it should not be, or unreachable when it should be. For example, a model may be unsound if there exists an outcome of executing a program on hardware that is forbidden by the model. In this case an expert must confirm[2] that the hardware behaviour is correct and the model differs. Dually, a model may be unsound if it allows an outcome that is forbidden by all implementations. We must be careful here as C/C++ and architecture models are intentionally loose to allow for implementation freedom. We focus on cases where the implementation is confirmed to be correct but the executable model and simulator disagrees. In practical terms unsoundness manifests as test results that do not match some ground truth. Testing with unsound models or simulators may miss bugs or raise false positives.

[1] Not to be confused with the soundness of mappings from a source to assembly. A model is unsound if it can prove a statement that should be false.

[2] Of course an implementation may be incorrect, but we focus on checking the models.

Likewise, a model or simulator is *incomplete* if there exists an outcome of executing a program under an idealised implementation that is *not exhibited* by the model or simulator. This case differs in that the model or simulator simply does not detect a behaviour, and hence does not allow or forbid it. In this case the model and simulator may silently pass over genuine bugs.

We study the unsoundness and incompleteness of the `herd` simulator and its models when used as parts of our oracle for compiler testing. Prior to the development of Téléchat, such analysis was done manually, by fuzzing, or using hardware [99, 4]. In general, verifying the correctness of an oracle is undecidable, and so we focus on directed testing using small litmus tests.

We study several cases where compiler testing can go wrong. Where possible, we adapt models or simulators to handle these cases, or highlight areas for future work. We first explore the effect of unsound model implementations on testing.

5.2 Unsound Model Implementations

The most straightforward way testing can go wrong is if a model or simulator is implemented incorrectly. We found a bug in the unofficial `arm.cat` implementation of the Armv7 model automatically when testing LLVM and GCC using Téléchat. There are outcomes of executing the compiled program under the buggy `arm.cat` model [11] (unofficial) that are forbidden under the RC11 model [46]. We fixed the model implementation to forbid the behaviour [65].

Example 5.2.1. In Figure 5.1 (top) the outcome `{ P0:r0=0, P1:r1=0 }` occurs if the store can be reordered past the load on either thread. This should never happen under the SC model [91], let alone a C/C++ model such as `rc11` [46]. RC11 is by no means an authoritative reference for ISO C/C++, but all C/C++ models we used had the same behaviour in this case. Figure 5.1 (bottom) shows the compiled program and the outcomes under the `arm.cat` model [11] when simulated using `herd`. The outcome is observed since the `fence` is compiled to a `DMB ISH` instruction, however `DMB ISH` was

Source Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  store(x, 1, sc);
  fence(sc);
  int r0 = load(y, sc);
}
P1 () {
  store(y, 1, sc);
  fence(sc);
  int r1 = load(x, sc);
}

exists (P0:r0=0 /\ P1:r1=0)
```

Outcomes

rc11.cat

```
{ P0:r0=0, P1:r1=1 }
{ P0:r0=1, P1:r1=0 }
{ P0:r0=1, P1:r1=1 }
```

Assembly Litmus Test

```
{ *x = 0, *y = 0 }

P0              |P1
MOV R0,#1       |MOV R2,#1
STR R0,[%P0_x]  |STR R2,[%P1_y]
DMB ISH         |DMB ISH
LDR R1,[%P0_y]  |LDR R3,[%P1_x]

exists (P0:R1=0 /\ P1:R3=0)
```

Outcomes

arm.cat (buggy)

```
!!{ P0:R1=0, P1:R3=0 }!!
  { P0:R1=0, P1:R3=1 }
  { P0:R1=1, P1:R3=0 }
  { P0:R1=1, P1:R3=1 }
```

Figure 5.1: (Top) C/C++ Store Buffering test. (Bottom) Compiled Armv7-A test.

not implemented correctly in `herd`. Specifically the fence was not marked as a barrier in the `herd` source code. It is unclear how long the bug existed in the `arm.cat` model before we found it. We fixed [65] the `arm.cat` model and `herd` so that it no longer exhibits `{ P0:R1=0, P1:R3=0 }` in this case.

We next explore a compiler bug that also lay dormant, since modelling efforts did not consider the interaction of `const` and atomic operations.

5.3 Incomplete Model Implementations

We report a runtime crash [57] that arises when compiling and executing an 128-bit `const`-qualified atomic load. This is not a concurrency bug since it neither exhibits re-ordering nor requires multiple threads. However, sequential testing will likely miss the interaction since atomics are associated with concurrency.

Source Program — C/C++ Outcomes

```
const _Atomic __int128 x = 0;

P0() {
    __int128_t r0 = load(&x,sc);        { P0:r0=0 }
}

// exists (P0:r0 = 0)
```

```
$ clang -O0 test.c -o test.o && ./test.o
[1] 55891 bus error ./test.o            (not detected below)
```

AArch64 Program — AArch64 Outcomes

```
{ __int128_t *x = 0 }

P0
  LDR X9, [X%P0_x]
  loop: LDAXP X8, X10, [X9]             { P0:X8=0, P0:X10=0 }
  STLXP W11, X8, X10, [X9]
  CBNZ W11, loop

exists (P0:X8 = 0 /\ P0:X10 = 0)
```

Figure 5.2: 128-bit Const Atomic Load. This program crashes when run.

Example 5.3.1. Figure 5.2 (top) probes `const` semantics in LLVM and induces a run-time crash when run. Figure 5.2 is a single-threaded program that `loads` an 128-bit integer with SC [91] ordering. Compiling Figure 5.2 (top) using LLVM 11 produces Figure 5.2 (bottom). In Figure 5.2 (bottom), the load is implemented using a store instruction inside a compare-and-swap (CAS) loop. The `LDR` instruction loads the address of `x` from memory through a pointer to `x`. `LDAXP` attempts an exclusive access on the 128-bits of `x`, loading the contents into the registers `X8` and `X10`. `STLXP` then attempts to write the same data back to `x`, and if it succeeds it will write `0` to `W11` or `1` if the exclusive access was interrupted. Then `CBNZ` checks `W11` and if zero will continue, otherwise it will branch to the `loop` label and retry as the store failed. Figure 5.2 (bottom) crashes at runtime - since the store to `x` fails as it is marked as read-only.

Unfortunately, we cannot just remove the store instruction. An engineer from Arm's GCC team stated [56]: *"Atomic loads have to work even if the object is marked const. C/C++ permits a non-const object pointer to be cast to*

a const object pointer. That means that any mechanism used for atomic loads has to work with const objects". Further, ISO C/C++ [80] specifies that the source program must not write to the object; however the implementation *may* write, if there are no side effects. In this case, the compiled program crashes which is a side effect.

The `herd` simulator equipped with the AArch64 model does not catch the crash in Figure 5.2. The outcomes of simulating the above tests are in Figure 5.2 (right) under the RC11 [46] and AArch64 [19] models, using `herd` [17]. The tests have the same outcomes after mapping states, but `herd` does not flag the crash since the AArch64 model does not account for read-only memory.

Example 5.3.2. We extend the AArch64 model to flag the crash. We describe our `const` Cat [5] model and use Téléchat [71] to show how `const` is miscompiled by LLVM-11. The syntax for Cat models is described in §2.2. Our model `includes` its semantics from the AArch64 memory model [19], and add rules for `const` objects. The set of `const-writes` is the intersection of accesses to `Const` regions of memory with program-write events, less the initial writes (`W \ IW`). We raise a `flag` if the set of `const-writes` is not `empty` as this contradicts the immutability of read-only memory. Figure 5.3 (top) shows our model. Figure 5.3 (bottom) shows the outcomes of simulating Figure 5.2 (bottom) under our model. The outcomes match those in Figure 5.2, but now flags const violations.

```
include "aarch64.cat"

let prog-wr = W \ IW
let const-writes = [Const] & (prog-wr * prog-wr)

flag ~empty const-writes as non-const
```

Outcomes under new model:

```
{ P0:X8=0, P0:X10=0 }
Flag non-const
```

Figure 5.3: (Top) Model of `aarch64+const`. (Bottom) The const requirement is now flagged under simulation.

This crash was fixed in LLVM for Armv8.4 or above. Unfortunately, the problem remains for Armv8.0 since there is no lock-free method of ensuring atomicity whilst keeping data in read-only memory [71]. To mitigate this issue, we specified in the Atomics Application Binary Interface [69] for the Arm 64-bit architecture that 128-bit integer `const`-qualified atomic data should be marked as mutable. We are aware that this also affects x86 code generation.

5.4 Incomplete Modelling of Undefined Behaviour

We now study how incomplete modelling of *undefined behaviour* leads to incorrect diagnosis of a behaviour as allowed. According to the ISO C23 standard [80], undefined behaviour (UB) is *"behaviour, upon use of a non-portable or erroneous program construct or of erroneous data, for which* [C23] *imposes no requirements"*. Informally, UB is often a proxy term for bad behaviour that should be removed. Indeed, compilers assume programs are free of UB when applying optimisations. Enumerating UB for the purposes of its removal is not however trivial [161]. We observe a case of undefined behaviour that was problematic for test engineers seeking to find bugs in their code where it interacts with atomics. We show how the lack of UB modelling can inhibit reasoning about the compilation of concurrent programs and induce incorrect results under C/C++ models.

Example 5.4.1. Figure 5.4 does not induce a data race, but returning from the `ub_if_returns` branch is undefined behaviour. Both LLVM and GCC assume programs have no undefined behaviour, inferring that `ub_if_returns` is never called, that `r0==0` must be false, and thus `if(r0)` is always true. LLVM removes the conditional branches in `P0`, and writes to `y`, allowing the outcome `{ P0:r0=1, P1:r1=1 }` to occur under AArch64 [19]. Debugging such an example could be challenging, since compilers can remove all code on the path to statically detectable undefined behaviour. Unfortunately neither `herd` nor any of `herd`'s C/C++ models flag this behaviour as UB.

```
{ *x = 0, *y = 0 }

[[noreturn]] inline void ub_if_returns() { return; }
P0 () {
  int r0 = load(x,acq);
  if (r0) {
    store(y,1,rel);
  }
  if (r0 == 0) {
    ub_if_returns(); // UB if this returns
  }
}
P1 () {
  int r1 = load(y,acq);
  store(x,1,rel);
}

exists (P0:r0=1 /\ P1:r1=1) // forbidden by rc11+lb
```

Figure 5.4: Goldblatt's [74] litmus test.

Example 5.4.2. Consider `"P0(){ store(y,1,rel);ub_if_returns(); }"`. If compiled to target Armv8.0-A using `"clang -O3"`, then the whole of `P0` is optimised away. Further, if `main` calls a function `f` that calls a `g` that calls `P0` then all functions including `main` are removed. If the example program is compiled and linked with a program that calls `P0` then executing the compiled code for `P0` may proceed to the code below `P0`. This could be a security issue if the linker stores sensitive code after `P0`.

Simulating these tests under `herd`'s C/C++ models will not detect such issues. Simulating the code in the above examples under `herd's` C/C++ models will flag data races as undefined behaviour, but other forms of undefined behaviour are not considered. Tools that rely on `herd`, or its models, may therefore exhibit false negatives. For instance the outcome `{ P0:r0=1, P1:r1=1 }` of Figure 5.4 would be incorrectly flagged as a compiler bug after compilation when it is due to undefined behaviour — it is a bad test. Likewise, if a compiled program links against the example code then the processor may fault at runtime; however, simulation (using for instance `herd` [12]) may not show it, since it does not check code outside the provided threads of execution. In either case, models do not help the user find the bug in their code.

Fortunately, GCC and LLVM have solutions for these problems. LLVM has `-mllvm -trap-unreachable`, which adds return statements to unreachable branches to prevent code from 'falling through'. Unfortunately this option is not on by default. GCC has `-funreachable-traps` for similar purposes at optimisation levels `-O1` and above. We encourage developers to use the appropriate compiler flags to avoid trying to diagnose the interactions between undefined behaviour and concurrency.

5.5 Comparing C/C++ Models

We now use Téléchat to identify discrepancies between C/C++ models. Since we use the source model and `herd`[3] simulator to produce expected program behaviour, it stands that an unsound C/C++ model may forbid behaviours exhibited by compiled programs, or allow behaviours not otherwise seen after compilation. In the former case such differences are either compiler bugs or cases where the model is *too strong*. In the latter case models are either deliberately loose or otherwise *too weak*. As our testing is particularly sensitive to the choice of source model, it is informative to compare the relative strength C/C++ models available [12]. In this section, we use Téléchat to automate the comparison of C/C++ models to determine their relative strengths and thus the limitations of each when used to produce source behaviours.

By comparing C/C++ models, we see where each model is too strong or weak. We show which source models match behaviours expected by C11 [80]. We then compare models using large suites of tests to identify tests where models are too strong or too weak. Lastly, we find that newer C/C++11 models allow source behaviours not seen when simulating any compiled program under `herd`'s architecture models. We make a case for further work on C/C++ models to create a model that is weak enough to permit behaviours expected by C/C++ standards, but strong enough to prohibit compiled program behaviours not seen under any architecture model. Figure 5.5 lists the models we use.

[3]We fix the simulator as `herd` from Git [12] commit version `#f0040aca50b`.

#	Cat Model $\mathcal{M}_{\texttt{SRC}}$	SRC Abbreviation
1	`c11_orig.cat` [26]	`orig`
2	`c11_partialSC.cat` [27]	`pSC`
3	`rc11.cat` [46]	`rc11`
4	`rc11+lb.cat`	`rc11+lb`

#	Cat Model $\mathcal{M}_{\texttt{ARCH}}$	ARCH Abbreviation
1	`arm.cat` [11]	`arm`
2	`aarch64.cat` [19]	`aarch64`
3	`riscv.cat` [106]	`riscv`
4	`x86tso-mixed.cat` [110]	`x86`
5	`ppc.cat` [108]	`ppc`
6	`mips.cat` [109]	`mips`

Figure 5.5: The models used during testing, from `herd` (commit ID `#f0040aca50b`).

These models are described as follows. The `orig` model is a formalisation of the original C/C++11 model. The model proposed by Boehm and Adve [36] laid down the basic ideas of what became the C++11 model, whose designs were tweaked and fixed by Batty et al. [28] (and many others) on the way to becoming the C++11 model and the work of Batty [32]. This model permits load buffering and forbids *thin-air reads*. Thin-air reads describes a modelling problem that requires that Load Buffering tests, such as Figure 5.8, to forbid outcomes whose values are not read from the program itself. The `pSC` model, also by Batty et al. [28], is a slightly stronger variant of `orig`. The `orig` model uses the S4 axiom [28] whereas the `pSC` model uses the S4a axiom of the same paper. The `pSC` and `orig` models are considered equivalent[4]. Like `orig`, the `pSC` model permits load buffering and forbids thin-air reads. The `rc11` model encodes the model of Lahav et al. [90]. The `rc11` model makes many improvements to prior models and is still used in many papers to date. Unlike previous models, `rc11` *forbids* load buffering and thin-air reads. The `rc11+lb` encodes the RC11 model [90] but allows load buffering.

To compare these models, we compare the outcomes of source and compiled tests. We use `herd` to collect the outcomes under each source model, and the compiled program outcomes under the architecture model below.

[4]From private communications with Luc Maranget, the maintainer of herdtools.

We first show how all source models, except `rc11`, match behaviours expected from the C/C++ standards (C23 [80] to be specific).

Example 5.5.1. Consider the load buffering (LB) tests in Figure 5.6 and Figure 5.7, which are in §7.17.3 of C23 [80]. C23 does not forbid the test in Figure 5.7, but paragraph 15 on page 266 in C23 states *"this is not useful behaviour, and implementations should not allow it"* [80].

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  int r0 = load(y,rlx);
  store(x,r0,rlx);
}


P1 () {
  int r1 = load(x,rlx);
  store(y,1,rlx);
}

exists (P0:r0=1 /\ P1:r1=1)
```

Outcomes

rc11
{ P0:r0=0, P1:r1=0 } { P0:r0=1, P1:r1=0 }
orig|pSC|rc11+lb
{ P0:r0=0, P1:r1=0 } { P0:r0=1, P1:r1=0 } !!{ P0:r0=1, P1:r1=1 }!!

Figure 5.6: LB test explicitly allowed by C23 [80]. The compiled program has the same or less (in the case of `x86`) outcomes as `rc11+lb`.

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  int r0 = load(y,rlx);
  if (r0 == 1) {
    store(x,r0,rlx);
  }
}
P1 () {
  int r1 = load(x,rlx);
  if (r1 == 1) {
    store(y,1,rlx);
  }
}

exists (P0:r0=1 /\ P1:r1=1)
```

Outcomes

rc11
{ P0:r0=0, P1:r1=0 }
orig|pSC|rc11+lb
{ P0:r0=0, P1:r1=0 } !!{ P0:r0=1, P1:r1=1 }!!

Figure 5.7: LB test explicitly allowed by C23 [80]. The compiled program exhibits `{ P0:r0=0, P1:r1=0 }` only.

Example 5.5.2. Consider the LB tests in Figure 5.8 and Figure 5.9. Figure 5.8 outlines an LB variant explicitly forbidden by C23. Figure 5.9 is not explicitly mentioned, but upon further reading is allowed. §5.1.2.4 of C23 [80] states *"relaxed atomic operations are not included as synchronisation operations although, like synchronisation operations, they cannot contribute to data races."*

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  int r0 = load(y,rlx);
  store(x,r0,rlx);
}
P1 () {
  int r1 = load(x,rlx);
  store(y,r1,rlx);
}

exists (P0:r0=1 /\ P1:r1=1)
```

Outcomes

rc11
{ P0:r0=0, P1:r1=0 }
orig|pSC|rc11+lb
!!{ P0:r0=S8, P1:r1=S8 }!! { P0:r0=0, P1:r1=0 }

Figure 5.8: LB variant explicitly forbidden by C23 [80]. `herd` does not create "thin-air" values and instead exposes its internal state `S8`. The compiled program exhibits the outcome `{ P0:r0=0, P1:r1=0 }`.

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0() {
  int r0 = load(x,rlx);
  fence(rlx);
  store(y,1,rlx);
}


P1() {
  int r1 = load(y,rlx);
  fence(rlx);
  store(x,1,rlx);
}

exists (P0:r0=1 /\ P1:r1=1)
```

Outcomes

rc11
{ P0:r0=0, P1:r1=0 } { P0:r0=0, P1:r1=1 } { P0:r0=1, P1:r1=0 }
orig|pSC|rc11+lb
{ P0:r0=0, P1:r1=0 } { P0:r0=0, P1:r1=1 } { P0:r0=1, P1:r1=0 } !!{ P0:r0=1, P1:r1=1 }!!

Figure 5.9: Load Buffering test implicitly allowed by C23 [80]. Variants without fences are also allowed. The compiled program exhibits the same (or less) outcomes as `rc11+lb`.

In summary load buffering is allowed, except for tests that allow thin-air behaviours [37]. Of course, it is not enough to test a small sample of tests, so we now test compilation using two larger suites of C/C++ tests. The first suite contains the default 89 litmus tests output by `"diy7 -arch C"`. The second suite contains the 167,184 litmus tests from prior work [71]. We feed these tests through the flow outlined in §3.2 under `SRC` and `ARCH` models.

We categorise test results by the number of outcomes that match before and after compilation. For each model we automatically identify tests where $outcomes'(comp(s), \mathcal{M}_{ARCH}) \not\subseteq outcomes(s, \mathcal{M}_{SRC})$ and explain what concurrency patterns cause this. We manually study tests where all compiled programs under their respective `ARCH` model exhibits $outcomes'(comp(s), \mathcal{M}_{ARCH}) \subset outcomes(s, \mathcal{M}_{SRC})$ to identify why the source model is more permissive.

We discuss the results of each C/C++ test suite in order, starting with the suite of 89 `diy` tests. Figure 5.10 presents 4 tables of results, one for each `SRC` model. Each column of each table lists results for `clang` and `gcc`, for each of the 6 `ARCH` models listed in the rows. The first thing to note is that Tables A and B have the same results, which suggests `orig` and `pSC` are the same for this small test sample. In Figure 5.10 the 6 tests that exhibit $\not\subseteq$ outcomes under `rc11` are due to variants of the load buffering (LB) test of Figure 5.9, since `rc11` forbids load buffering. Under `rc11+lb` these behaviours are allowed. Dually, `rc11+lb` has more tests that permit more outcomes than `rc11`.

We next applied the suite of 167,000 tests. Consider the tables in Figure 5.11. Once again we split up tables by `SRC` model, rows by `ARCH` model, and columns by compiler. Unlike the previous experiment, Tables E and F differ in the $=$ and $\not\supseteq$ columns. For instance, taking the $\not\supseteq$ value for `clang` targeting `aarch64` gives us 39,952-39,856=96 tests where the `pSC` model permits behaviours that `orig` forbids. This suggests that the `orig` and `pSC` models are not in fact the same. The results in Tables G and H are similar to the previous experiment for `rc11` and `rc11+lb`. The 2,352 $\not\subseteq$ tests arise since `rc11` forbids LB. Since `rc11+lb` permits LB, this number drops to 0.

ARCH	$\not\subseteq_{\texttt{clang}}$	$=_{\texttt{clang}}$	$\not\supseteq_{\texttt{clang}}$	$\not\subseteq_{\texttt{gcc}}$	$=_{\texttt{gcc}}$	$\not\supseteq_{\texttt{gcc}}$
arm	74	5	20	74	5	20
aarch64	74	9	12	74	9	12
riscv	74	9	8	74	5	20
x86	74	0	18	74	0	18
ppc	74	5	12	74	5	12
mips	74	5	20	74	5	20

Table A: `SRC=orig`

ARCH	$\not\subseteq_{\texttt{clang}}$	$=_{\texttt{clang}}$	$\not\supseteq_{\texttt{clang}}$	$\not\subseteq_{\texttt{gcc}}$	$=_{\texttt{gcc}}$	$\not\supseteq_{\texttt{gcc}}$
arm	74	5	20	74	5	20
aarch64	74	9	12	74	9	12
riscv	74	9	8	74	5	20
x86	74	0	18	74	0	18
ppc	74	5	12	74	5	12
mips	74	5	20	74	5	20

Table B: `SRC=pSC`

ARCH	$\not\subseteq_{\texttt{clang}}$	$=_{\texttt{clang}}$	$\not\supseteq_{\texttt{clang}}$	$\not\subseteq_{\texttt{gcc}}$	$=_{\texttt{gcc}}$	$\not\supseteq_{\texttt{gcc}}$
arm	6	36	47	6	36	47
aarch64	6	44	39	6	44	39
riscv	6	49	34	6	36	47
x86	0	40	49	0	40	49
ppc	6	48	35	6	48	35
mips	0	38	51	0	38	51

Table C: `SRC=rc11`

ARCH	$\not\subseteq_{\texttt{clang}}$	$=_{\texttt{clang}}$	$\not\supseteq_{\texttt{clang}}$	$\not\subseteq_{\texttt{gcc}}$	$=_{\texttt{gcc}}$	$\not\supseteq_{\texttt{gcc}}$
arm	0	30	59	0	30	59
aarch64	0	38	51	0	38	51
riscv	0	43	46	0	30	59
x86	0	25	64	0	25	64
ppc	0	42	47	0	42	47
mips	0	23	66	0	23	66

Table D: `SRC=rc11+lb`

Figure 5.10: Testing the compilation of the 89 `diy` tests.

	clang			gcc		
ARCH	$\not\subseteq$	=	$\not\supseteq$	$\not\subseteq$	=	$\not\supseteq$
arm	14,480	91,168	61,536	14,592	89,344	63,248
aarch64	14,592	112,736	39,856	14,592	112,736	39,856
riscv	14,592	122,736	29,856	14,480	88,704	64,000
x86	14,528	96,392	56,264	14,528	96,392	56,264
ppc	14,592	114,884	37,708	13,248	114,812	37,708
mips	14,528	89,208	63,448	14,456	87,008	65,720

Table E: SRC=orig

	clang			gcc		
ARCH	$\not\subseteq$	=	$\not\supseteq$	$\not\subseteq$	=	$\not\supseteq$
arm	14,480	91,072	61,632	14,592	89,248	63,344
aarch64	14,592	112,640	39,952	14,592	112,640	39,952
riscv	14,592	122,640	29,952	14,480	88,608	64,096
x86	14,528	96,296	56,360	14,528	96,296	56,360
ppc	14,592	114,788	37,804	13,248	114,716	37,804
mips	14,528	89,112	63,544	14,456	86,912	65,816

Table F: SRC=pSC

	clang			gcc		
ARCH	$\not\subseteq$	=	$\not\supseteq$	$\not\subseteq$	=	$\not\supseteq$
arm	2,352	96,604	68,228	2,352	94,612	70,220
aarch64	2,352	120,532	44,300	2,352	120,532	44,300
riscv	2,352	130,628	34,204	2,352	94,060	70,772
x86	0	103,072	64,112	0	103,072	64,112
ppc	2,352	120,876	43,956	2,352	119,948	43,516
mips	0	97,520	69,664	0	95,176	72,008

Table G: SRC=rc11

	clang			gcc		
ARCH	$\not\subseteq$	=	$\not\supseteq$	$\not\subseteq$	=	$\not\supseteq$
arm	0	86,628	80,556	0	84,636	82,548
aarch64	0	110,556	56,628	0	110,556	56,628
riscv	0	120,652	46,532	0	84,084	83,100
x86	0	88,392	78,792	0	88,392	78,792
ppc	0	110,900	56,284	0	110,412	55,356
mips	0	82,850	84,344	0	80,568	86,616

Table H: SRC=rc11+lb

Figure 5.11: Testing the compilation of the 167k tests. Creating each table takes 2 hours 15 minutes when fully parallelised on a 224 core Thunder X2 with a timeout of 120s, using GNU parallel [150].

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  store(x,2,rel);
  fence(sc);
  store(y,1,sc);
}
P1 () {
  int r1 = load(y,con);
  if (r1 == 1){
    store(x,1,sc);
  }
}

exists (P1:r1=1 /\ x=2)
```

Outcomes

orig
{ P1:r1=0, x=2 } { P1:r1=1, x=1 }
pSC|rc11|rc11+lb
{ P1:r1=0, x=2 } { P1:r1=1, x=1 } !!{ P1:r1=1, x=2 }!!

Figure 5.12: S Test. No ARCH model we checked exhibits `{ P1:r1=1, x=2 }`.

Example 5.5.3. We end by studying a test whose behaviours are not observed by any architecture model. Figure 5.12 shows an 'S' litmus test that checks for the outcome `{ P1:r1=1, x=2 }`. The outcome is forbidden by `orig` but allowed by `pSC`, `rc11`, and `rc11+lb`. All compiled programs under their respective ARCH models match the outcomes of `orig` in this case. Lahav et al. [90] (and Batty et al. [28]) state that `rc11` (`pSC`, respectively) does not support consume semantics. This also explains why `rc11+lb` does not support consume. By default `herd` allows outcomes (such as `{ P1:r1=1, x=2 }`) if they are not otherwise constrained. Taken together these facts suggest that, for this example, `pSC`, `rc11`, and `rc11+lb` are too weak. We reiterate that it is not *wrong* for source models such as `rc11` to be too weak, as the C/C++ model must balance concerns of optimisation, supporting new hardware, and programmability; however in this case the `pSC`, `rc11`, and `rc11+lb` models are too weak simply because they are incomplete and thus `herd` silently allows the outcome `{ P1:r1=1, x=2 }`. We do not believe this outcome was intended.

We note that C/C++11 is not C/C++26, or even C++23. While this thesis was under review [35] the C++26 committee agreed to deprecate consume semantics. This means that we would need a different C/C++ model and version of `herd` for C++26 that warns the user when consume tests are used.

Three conclusions can be drawn from this section. The first conclusion is that for the source model and `herd` pairs we tested, none fully meet the requirements of C++11 either because they are too strong (see `orig` and `pSC` results in Figures 5.10 and 5.11) or are too weak (see the `pSC` and `rc11{+lb}` outcomes in Figure 5.12). This matters for compiler testing, since we want to test all atomic statements supported by C/C++11.

The second conclusion is that any pair of `herd` and C/C++11 model would not also work for newer versions of C/C++. This conclusion is perhaps unsurprising, but is worth stating as it has consequences for compiler testing. LLVM and GCC have different code generation depending on which C or C++ version they are compiling (as specified by the profile `"-std={c++11, c++20, ...}"`). Further, some performance critical software[5] still depends on older versions of C/C++, and so we cannot just test compilation under the newest language version. In order to test the code generation of atomics under each C/C++ version, we need models and a simulator appropriate for each. It is further work to use standards-aware simulators, such as `CerberusBMC` [94], to conduct testing for each C/C++ version.

Third, we learned that the `orig` model is not the same as the `pSC` model. It was previously understood that these models were equivalent; however this is not the case. There are 96 tests that exhibit outcomes under `pSC` that are forbidden by `orig`. We do not believe these differences were intended.

There are limitations to this experiment of course. We assume that the most recent compiler correctly translates all of these tests, and that the source models support all behaviours exposed by ISO C/C++. This thesis shows that there are bugs in today's compilers, so the first assumption may not always hold. We also learned that not all models support consume semantics in light of the results above. In summary, we expect more refinements to the C/C++ models and `herd` are needed if they are to be used in further automated testing.

[5]For instance, at the time of writing CMSIS is C99 and C++03 complaint: `https://arm-software.github.io/CMSIS_6/latest/General/index.html#coding_rules`

5.6 Precision Testing Under Architecture Models

We show how testing for architecture-version-specific behaviours under architecture models can be a challenge. An architecture model can be thought of as the envelope of all permitted implementations. That is, by fixing some features the model becomes a particular architecture sub-version. As an architecture develops, the available features and their permitted behaviours can change. If we use `herd` and an architecture model to produce compiled program behaviour then it is possible that it permits behaviours that are not seen under a particular architecture sub-version. In order to get precise answers from the compiler testing oracle, the user must determine the subset of model behaviours that corresponds to their sub-version of interest. This can be especially challenging for engineers who are not necessarily concurrency experts or architects. We provide one such example that proves challenging for engineers debugging compiled program behaviours for a particular architecture sub-version.

```
{ i128 *x = {1,1} }
P0 () {
  store(x, 0, sc);
}
P1 () {
  i128 r1 = load(x, sc);
}
exists (P1:r1=0)
```

```
{ i128 *x = {1,1} }
P0
  MOV X8, #0
  MOV X10, #0
  STP X8, X10, [X%P0_x]
P1
  LDP X5, X7, [X%P1_x]
exists (P1:X5 = 1 /\ P1:X7 = 0)
```

Figure 5.13: 128-bit load store litmus tests.

Example 5.6.1. Figure 5.13 (left) shows a litmus test that loads an 128-bit atomic integer on one thread and stores it on the other. Figure 5.13 (right) shows a candidate target program which was considered an alternative to the CAS loop in Figure 5.2. We set the 128-bit location `x` as 2 64-bit values using the array notation `{ 1, 1 }`. If the outcome `{ P1:X5=1, P1:X7=0 }` arises, then the store pair (`STP`) instruction has done a partial write to `x`, which violates

atomicity. From Armv8.4-A onwards, LDP/STP instructions are guaranteed to be single-copy atomic assuming data accesses are naturally aligned in normal memory. Single-copy atomicity prevents atomics from *tearing* [55] and forbids the '1-0' outcome on implementations. No such guarantee exists for the Armv8.0-A architecture and tearing may be observed on a machine that implements Armv8.0-A. Figure 5.14, shows two possible sets of outcomes when running Figure 5.13 (right) under Armv8.0-A and Armv8.4-A architecture sub-versions:

Armv8.0-A
{ P1:X5=0, P1:X7=0 }
{ P1:X5=0, P1:X7=1 }
{ P1:X5=1, P1:X7=0 }
{ P1:X5=1, P1:X7=1 }

Armv8.4-A
{ P1:X5=0, P1:X7=0 }
{ P1:X5=1, P1:X7=1 }

Figure 5.14: Possible Armv8.4-A outcomes of Figure 5.13 (right).

The `herd` simulator returns the left table of outcomes when running Figure 5.13 (right) under the AArch64 model. In other words, the particular version[6] of `herd` and Cat model we used did not correctly capture the mixed-size architectural intent for Armv8.4-A. This means a concurrency expert must analyse the program to determine the subset of outcomes allowed under a particular architecture sub-version. Automated testing for the presence of the '1-0' case for a particular architecture sub-version cannot yet be achieved without experts available to interpret `herd`'s output.

[6]There is an open patch to address this issue at the time of writing: `https://github.com/herd/herdtools7/pull/670`. However the general problem of pinpointing sub-version specific behaviours remains.

Chapter 6

Related Work

In this chapter, we compare our work with the state of the art in compiler testing for concurrency. We also survey related work in interoperability and concurrency tools more broadly.

This chapter is structured as follows: §6.1 describes the state of the art compiler testing techniques specifically for relaxed memory concurrency, §6.2 describes work related to mix testing, and §6.3 describes litmus test generators, simulators, and adjacent work in program improvement.

6.1 Compiler Testing Techniques

This section surveys the state of the art compiler testing techniques. The state of the art tools are `cmmtest`, `validc`, and `c4` [118, 41, 160, 159]. We are not aware of any work newer than [159]. We summarise their contributions in Table 6.1 and elaborate on each below.

6.1.1 The `cmmtest` tool: Morisset et al. (2013)

Morisset et al. [118] conduct *differential testing* of GCC under the C/C++ model as formalised by Batty et al. [31]. Differential testing compiles a program with different compilers, comparing the behaviours of each. For instance, comparing the outcomes of running executables produced by `"clang -O1"` and `"clang -O3"`. Morisset et al. contribute a theory of *sound optimisations*, which compares executions to determine the validity of compiler optimisations. An execution of an optimised program is *valid* if it can be obtained by transforming a reference

Component	`cmmtest` (2013)	`validc` (2016)	`c4` (2021/2)
Generator	CSmith	Custom Fuzzer	Memalloy
Concurrent?	No	No	Yes
Behaviour	Execution Matching		Outcomes
Test Env	Hardware Exec	Model	Hardware+Model
Domains	Src-to-Asm	IR-to-IR	Src-to-Asm
Approach	Differential		Metamorphic
Models used	C/C++11	C/C++ and LLVM	C/C++
Test features	Non-atomic C/C++	LLVM atomics	C/C++
Compiler	GCC 4	LLVM 3.6	LLVM and GCC
Extendable	Proprietary tools	Open Source	
Bugs found	6	3	2

Table 6.1: Summary of the state of the art compiler testing techniques.

execution of the source program (allowed under the C/C++ model). The authors check soundness of sequential optimisations in a concurrent context by extending CSmith [162] with concurrency primitives and matching the generated test's executions before and after compilation. They collect hardware executions using Intel's proprietary pin tool [79], which instruments x86-based test environments to collect traces of program executions. They found six bugs in the GCC compiler. The `cmmtest` tool implements this technique as follows:

1. Generate a pseudo-random *single-threaded* C/C++ test s using CSmith.
2. Compile s using the profile `"gcc -O0"` (no optimisations), get c, execute c on a hardware-based test environment, get a reference execution ref.
3. Compile s again with optimisations (e.g. `"gcc -O1"`), execute in the same test environment and get optimised execution opt.
4. Using transformation rules, if there exists a transformation ref $\rightsquigarrow^*$ opt then the optimisation is valid.
5. If no such transformation is found, compose s with a thread that induces a concurrency bug.

The `cmmtest` tool and Téléchat differ in one important way. Firstly, `cmmtest` compares optimisations whereas Téléchat compares whole program

outcomes. This means `cmmtest` conducts finer grained analysis than Téléchat, but the matching algorithm may have to be adjusted to support new optimisations. Comparing outcomes is much simpler, and more robust against compiler changes, but it is possible to miss bugs in optimisations if a later pass masks buggy behaviour.

6.1.2 `validc`: Chakraborty and Vafeiadis (2016)

Chakraborty and Vafeiadis conduct translation validation of LLVM [41] under a C/C++ model [155] and an LLVM memory model formalised afterwards [39]. The authors study the differences between the C/C++ and LLVM memory models, and validate LLVM optimisation passes by matching executions before and after compilation following the technique of `cmmtest`. They conduct translation validation by matching *all* bounded executions allowed by models (for each initial state found by a solver). Since the LLVM model was being formalised at the time of the `validc` work, they created a custom fuzzer tool for the LLVM intermediate representation (IR) to increase the chances of finding bugs. They found three bugs. The `validc` tool works as follows:

1. Generate a pseudo-random C/C++ test s, using a custom fuzzer.

2. Compile s using `"clang++ -O0 -emit-llvm"` (no opt), get LLVM intermediate representation ir, analyse ir to extract a set of memory events refs allowed by C/C++ or LLVM model.

3. Compile s with optimisations (`"clang++ -O3"`), get optimised LLVM intermediate representation $iropt$, analyse $iropt$ to extract a set of optimised events opts under C/C++ or LLVM.

4. For each event opt in opts (for any ref in refs), if there exists a transformation ref $\rightsquigarrow^*$ opt where all path conditions match, then the optimisation is valid.

5. If no such execution match can be found, compose s with a thread that induces a concurrency bug.

The `validc` tool inherits some limitations of `cmmtest`, and addresses others. The authors make a similar assumption to Morisset et al., in that they rely on `"clang -O0"` to produce an unmodified IR that is presumed correct. It is possible that that atomics mappings emitted by `"clang -O0"` and operated on in subsequent optimisation passes is incorrect, but both `cmmtest` and `validc` are designed to test optimisations, rather than incorrect mappings outright. The `validc` tool operates on LLVM IR, which permits a finer grained analysis but is more complex — requiring two sophisticated execution matching algorithms. The `validc` tool does not however test compilation down to the assembly level, and may miss bugs in target-dependent optimisations. Téléchat faces no such restrictions on the languages used, as it is parameterised over models, and does not require execution matching, as it treats the compiler as a black box. Treating the compiler as a black box has its own limitations. Since the internals of a compiler are not visible in black box testing, pinpointing the cause of a bug can be a challenge.

6.1.3 The `c4` tool: Windsor et al. (2021, 2022)

Windsor et al. [160] conduct *metamorphic testing* of LLVM and GCC using both models and hardware. Metamorphic testing generates a variant s_2 of the source program s_1 that has the same behaviour as s_1, compiles both with the same compiler, and checks the behaviour of each is the same. For example, when compiling `"print(1+1)"` and `"print(2)"`, both compiled programs should output 2. The authors contribute an automatic technique for finding concurrency bugs. The technique is simpler than prior work in that it compares program outcomes rather than executions. If the outcomes of executing the compiled program are allowed by the source simulation then the compiler has not introduced a bug, within bounds. Automation is achieved by generating litmus tests using the `Memalloy` tool, simulating source program executions under using the `herd` simulator, and collecting hardware executions using the `litmus` tool [158, 17, 16]. To increase the chances of finding bugs, they fuzz C/C++ litmus tests (see §1.2.3) before compilation. They found two new bugs.

The `c4` tool was developed at the same time as Téléchat, and shares some similarities (see Chapter 3). The `c4` tool works as follows:

1. Generate a concurrent C/C++ litmus test s using `Memalloy` (see §6.3.1).
2. Simulate s under the C/C++ model to get $outcomes(s, \mathcal{M}_{SRC})$.
3. Fuzz s using semantic-preserving mutations and compile to get binary $comp(s)$.
4. Execute $comp(s)$ on hardware, get $outcomes(comp(s), hardware_{ARCH})$.
5. If $outcomes(comp(s), hardware_{ARCH}) \not\subseteq outcomes(s, \mathcal{M}_{SRC})$ there may be a concurrency bug.

The `c4` tool relies on hardware executions and may miss behaviours. To increase the chances of observing bugs, they must 'stress' the test environment by augmenting tests with additional threads that strain the memory system. Hardware execution is problematic, as it can miss bugs, if a given piece of hardware exhibits a problematic behaviour at all. It is possible that a hardware implementation is compliant with an architecture without exhibiting all behaviours that the architecture allows.

6.2 Concurrent Program Interoperability

In this section, we discuss related topics related to mix testing. Unlike the previous section, we do not directly compare these works with mix testing, since prior work does not test the mixing of assembly code. Instead, we survey other kinds of mixing of memory models and by inlining assembly. Table 6.2 summarises the work we compare.

Interoperability is a long-standing concern of engineers deploying portable code. The state of the art testing techniques in §6.1 assume the *whole* program is compiled at once, using one set of atomics mappings; this is the *closed-world* assumption [123]. Unfortunately, this assumption does not always apply as

	Heterogeneous (2023)	Inline Asm (2024)	Mix testing (2024)
Domain	CPU & GPU	Src & CPU	Src & CPU
Models	x86-TSO & PTX	RC11 & Ex86	RC11+lb & Armv7/v8
Interop	Inter-Thread	Intra-Thread	Intra-Thread
Approach	Soundness of Idealised Mappings		Compiler Testing

Table 6.2: Comparison of Concurrent Program Interoperability Work.

production code bases are often compiled separately. For example, a programmer may develop a program, store it, and then later link that program against a new one that uses a different compiler or mappings from C/C++ atomic operations to assembly sequences, thereby inducing mixing bugs. Mix testing was motivated by this use case.

6.2.1 Heterogeneous Computing: Goens et al. (2023)

Goens et al. [73] model heterogeneous systems consisting of CPUs and GPUs. Heterogeneous systems involve processors, graphics devices, and specialised hardware (for AI processing workloads, for instance). Each subsystem, including CPUs and GPUs, has its own instruction set and memory model. Subsystems are then fused together with an *interconnect* that enables inter-thread communication through shared memory. Goens et al. present a *compound* memory model, that formalises the interactions between x86-TSO CPU and NVIDIA PTX GPU subsystems [143, 103].

Compound models address a related problem to mix testing. Such models operate on *compound* litmus tests, where each thread is written in one language whose semantics is defined by one model. Mixing CPU and GPU code on one thread is not considered. Mix testing checks intra-thread compatibility of different yet compatible architecture versions, but requires the whole program is executed under one architecture model. Compound models on the one hand are more powerful than mix testing, as they prove that the soundness of existing compilation schemes from C to x86-TSO or PTX still hold after mixing such languages. Mix testing on the other hand is more flexible, since it can check mappings that exist outside the domain of the compilation schemes, assuming there is a compiler to emit those mappings.

6.2.2 Inline Assembly: Emílio de Vilhena et al. (2024)

This work adds *inline assembly* support to the C/C++ memory model. Inline assembly is assembly code that is embedded into C code. Many libraries, kernels, and embedded applications use inline assembly for performance gains. The semantics of inline assembly is defined by the architecture memory model, whereas the semantics of surrounding code is defined by the C/C++ model. Emílio de Vilhena et al. contribute a *mixed* model, that combines the RC11 and Ex86 [49, 90, 129] (a variant of the x86-TSO [143] model). Their model formalises the intra-thread semantics of inline assembly and C/C++.

This work is complementary to mix testing and was published in the same conference as our mix testing paper. This work however uses *mixed programs* consisting of x86 and C/C++ code. The authors prove that mappings involving both C/C++ and assembly are sound with respect to their mixed model. While verified soundness of mixed programs is a valuable contribution, formalising a mixed C/C++ model requires a lot of work and both the proof and model would need to be modified after changes to any model or mapping. In addition, a family of mixed models requires support for each architecture that is used in inline assembly. Mix testing uses simpler components, notably mappings and C/C++ programs — we do not need to mix models.

6.3 Relaxed Memory Concurrency Tools

Tests for behaviours using small programs goes back at least as far as Collier's [24, 47] work in 1992. Shasha and Snir made the observation [146] that relaxed behaviours may be modelled as cycles in the partial order over a program's executions. Tests characterised in this way check whether a cyclic execution s can reach a state σ under some model $\mathcal{M}$ (as introduced in §2.3.1). Concurrency tools based on axiomatic models are also built around these ideas. We compare some of these tools below.

6.3.1 Litmus Test Generators (2010-2023)

Diy 'do it yourself': Alglave et al. (2010/12) [15] present the `diy` tool, which generates litmus tests from cycles. We used `diy` throughout this work. As the name suggests, `diy` requires the user to provide cyclic specifications (see §2.3.1 for an example). The `diy` tool has been maintained for a decade now, and can generate tests that contain subsets of C/C++, Arm AArch64, Armv7, IBM PowerPC, Intel x86, RISC-V, MIPS, and more. We failed to generate one of the bugs we found using `diy`, since the cycle syntax does not account for readless RMW operations. As far as we can tell, all test generators that use cycles as specifications have this issue.

The `diy` tool is a semi-automatic test generator. The `diy` tool accepts both cycles and optionally a set of relations as input. While individual cycles generate one test at a time, if the user additionally specifies a set of *safe* relations, then `diy` will iteratively relax each safe relation in the cycle to generate multiple litmus tests at a time. We used this mechanism to generate test suites for our empirical testing campaigns.

Memalloy: Wickerson et al. (2017) [158] present a constraint-satisfaction problem, of which litmus test generation is an instance. Their idea was that problems of test conformance, model comparison, program strengthening, and compiler testing can be represented as a constraint satisfaction problem to which the solution is either a program or a pair of programs. Given models $\mathcal{M}$ and $\mathcal{N}$, and a binary relation $\blacktriangleright$, by finding programs s,t, some state σ, a fixed initial state i, an initial state mapping $f : \sigma \to \sigma'$, and final state mapping $g : \sigma \to \sigma'$ then, they check for solutions where the litmus test for $s(i)$ cannot reach σ under $\mathcal{M}$, but $t(i')$ *can* reach σ' under $\mathcal{N}$ where $s \blacktriangleright t$, $i' = f(i)$, and $\sigma' = g(\sigma)$. The `Memalloy` tool implements this technique and is the first tool to use constraints to generate executions and hence programs. Generated litmus tests are then solutions or counterexamples to one of the four problems. For instance, the `Memalloy` tool was used to find bugs in mappings from OpenCL to PTX. The `Memalloy` tool generates either one or possibly two litmus tests

per constraint solution found, depending on the form of the problem; and furthermore that automatic generation even in this form is still an improvement over cycle construction by hand with `diy`.

The constraint satisfaction problem [158] must be extended so that Téléchat can be made an instance of it. Since Téléchat compares source and machine states, we extended the framework above with f and g as they were not in the original formulation. We can then instantiate Téléchat by setting $\mathcal{M}$ as a C/C++ model, $\mathcal{N}$ as the architecture model, $\blacktriangleright$ as the compiler, f as the symbol table, and g as the DWARF debug information extracted when compiling a given litmus test s with input i. While prior work assumes $\blacktriangleright$ is a bijection, mix testing suggests $\blacktriangleright$ is a one-to-many relationship, since a compiler can pick one of many mappings based on the compiler profile. Mix testing implies the solver would have to do a potentially exponential amount of work to find t, if every statement in s has a different mapping.

Litmustestgen: Lustig et al. (2017) [104] builds on Wickerson et al. by finding all solutions to the constraint satisfaction problem within bounds on test size. Given a model as input, `Litmustestgen` generates a *suite* of litmus tests (rather than one or two) that satisfy a *minimality criterion*. The criterion specifies that no relation in the executions can be relaxed without allowing new outcomes in the resultant litmus test. They present the `Litmustestgen` tool that implements this technique, also using the Alloy [81] framework. They apply the tool to a number of case studies, and are able to both prune redundant tests and find new ones. Solving for minimality is unfortunately super-exponential in the test size bound. This complexity is fine for generating regression suites as a one-off task, but may not be acceptable if mix testing increases the complexity by another exponential factor. Super-exponential complexity is not feasible for automatically searching for tests that differ between compiler revisions. Nevertheless, `Litmustestgen` represents a significant improvement for automated litmus test generation.

Kater: Kokologiannakis et al. (2023) [87]. Previous techniques only search

for counterexample tests up to a bounded test size. The `Kater` tool conducts an unbounded search by generating consistency checks on executions under a given model. Like `Memalloy`, the `Kater` tool focuses on model metatheory, e.g. for verifying idealised mappings, but could be tailored to the task of verifying real compiler mappings.

6.3.2 Relaxed Memory Simulators (2010-2022)

memevents, ppcmem, and RMEM: Alglave et al. (2009) and Sarkar et al. (2009) present tools for exploring architecture model behaviours [7, 136]. The `memevents` tool explored program behaviours under axiomatic x86-CC and x86-TSO models. The `ppcmem` and `ppcmem2` tools (that later evolved into `RMEM` [137]) enable the exploration of Armv8, IBM Power, and x86-TSO program executions under operational models. There are other earlier tools, such as `TSOtool` [76] and `Nemos` [163], but we focus on `memevents` and `RMEM` as they are close predecessors of `herd`.

Cppmem: Shortly after Sarkar et al. 2011, Batty et al. (2012) present a tool for exploring C/C++ model behaviours [30]. The `cppmem` tool implements the C/C++ model as formalised by Batty et al. [32, 31]. Like `RMEM`, the `cppmem` tool enables exploration of program behaviours under the C/C++ model. Both `RMEM` and `cppmem` were vital in early efforts to formalise source and architecture models through interactive exploration, random search, and bounded enumeration.

Herd: Alglave et al. (2014) parameterise simulation over models expressed in the Cat language [17, 5]. The `herd` simulator (which was an evolution of `memevents`) takes a litmus test and memory model as input, and returns the outcomes of executions. Unlike previous tools, `herd` models were provided as separate Cat files rather than as hard-coded OCaml implementations. The `herd` tool requires users to implement instruction semantics and may be incorrect as errors may creep into such implementations. For example, we only discovered the bug [60] in the C/C++ `exchange` implementation after the semantics of

the SWP instruction were corrected in herd, but this required no changes to Téléchat. Significant work was required to make herd suitable for compiled programs. Since the intrinsic computational complexity of simulating all program executions is high [38], the RMEM, ppcmem, cppmem, and herd tools suffer from scalability issues.

Dartagnan: Ponce De León et al. (2018) [126] expanded on the Memalloy work, by using SMT solvers to encode behaviours of litmus tests. The Dartagnan tool implements this technique and supports Cat models of CPUs, GPUs, and source code. Using solvers to check for satisfiability means Dartagnan returns one outcome at a time. As such Dartagnan must be run repeatedly to collect multiple behaviours, however Dartagnan's focus on satisfiability reduces the effect of state-space explosions, enabling scalability to larger programs.

GenMC: The algorithms first implemented in RMEM and cppmem, and later herd, are *stateless*. Stateless algorithms generate all executions by permuting events, before filtering out invalid executions using a model. Stateless algorithms do not scale as we have shown in our experiments. Kokologiannakis et al. (2019) [89] present an algorithm that adds events one at a time, checking for consistency at every step — thus reducing the impact on scaling. They present the GenMC tool, which implements this algorithm. The GenMC tool was used to verify implementations of concurrency libraries.

Isla and CerberusBMC: Armstrong et al. (2021) and Lau et al. (2019) present tools for exploring whole language semantics [21, 94]. The Isla tool [21] is a symbolic execution engine that takes a Cat architecture model and the semantics of the whole architecture (not just the concurrent parts) as provided by the Sail [23] specification, to explore the behaviour of litmus tests. Likewise, the CerberusBMC tool accepts a Cat C/C++ model, and supports a much larger proportion of the C/C++ language as implemented in the Cerberus C semantics. By scaling behaviour exploration using SMT solvers, and using two larger sources of language semantics these tools address the two key limitations of using herd.

NVLitmus: Lustig et al. (2022) [102] present a tool for reasoning about the NVIDIA PTX model. Previous models all share the same global view of memory, which is subject to certain coherence rules. Modern GPUs typically have custom hardware that is tightly coupled to the GPU datapath, but is not necessarily subject to the same coherence rules. To restore reasoning, Lustig et al. model non-coherent datapaths, capturing the non-standard memory behaviour that results. Their *proxy* model strictly generalises prior models, in that every fence or memory access has an additional tag, which specifies the domain where synchronisation must apply. Prior models are then a special case, as their accesses are all tagged with the baseline generic proxy. Similar to other SMT-based tools, the `NVLitmus` tool takes an Alloy [81] model, a PTX litmus test, and checks whether certain states are reachable.

Nothing in our techniques are specific to `herd` or the Cat models. The SMT-based tools, improved algorithms, and Alloy models could help in scaling compiler testing. We defer these ideas to future work.

6.3.3 Program Improvement (2009-2023)

Last but not least, we summarise tools for program improvement. These works are not strictly related to compiler testing, but rather detect bugs in programs and attempt to fix them.

Sanitisers: There are many *sanitisers*, which take a program and check if a program exhibits a particular problem. The `ThreadSanitizer` tool [139] detects data races in large C/C++ programs. The `UndefinedBehaviourSanitizer` [44] tool detects large classes of undefined behaviour. The `AddressSanitizer` [138] tool can detect many kinds of buffer overflow and use-after-free bugs. The `MemorySanitizer` [148] tool can detect uninitialised memory. These tools are included as part of LLVM. We note that most of these tools were developed independently of memory modelling efforts, and so they may miss reordering behaviours that otherwise dedicated tools can catch.

Porting and Fence Insertion: Compiler testing is analogous to the *porting*

problem. The porting problem concerns the translation of concurrent programs in one language, such as x86-TSO, to a more relaxed language such as AArch64. To port successfully, additional synchronisation is required to ensure new reordering cannot occur. As such compilation and porting are instances of the same translation problem, where the source is typically less relaxed than the target. Beck et al. [33] tackle the problem of porting concurrent programs from one relaxed language to another. The `Atomig` tool [33] finds and fixes bugs amongst millions of lines of code when porting the MariaDB MySQL database management system. This tool detects concurrency patterns and adds implicit synchronisation to avoid using expensive fences. Rocha et al. [134] take a different approach, by making the whole application sequentially consistent, before *removing* barriers. Dually, the area of fence insertion aims to prevent reordering by inserting barriers. Nimal [121] observed that cyclic executions could be found using Tarjan's [151] strongly connected components algorithm. The `CImpact` tool [9] implements this technique, finding unexpected cycles in large C/C++ programs where fences can be inserted.

Chapter 7

Conclusions

We outline our contributions (§7.1), and highlight possible future work (§7.2).

7.1 Summary of Contributions

We begin by summarising our contributions, the limitations of our work, and the implications that follow.

Compiler testing under source and architecture models: We present the Téléchat technique that checks the compilation of concurrent C/C++ litmus tests by comparing the allowed outcomes of execution before and after compilation as permitted by source and architecture models respectively. A concurrency-related compiler bug arises when there is an outcome of a compiled program allowed by the architecture model that is not an outcome of the source program allowed by the source model. We found and reported a number of new concurrency bugs, and conducted a number of novel case studies on the state of concurrency compilation in GCC and LLVM. As far as we know, the Téléchat toolchain is the first tool of its kind to be deployed in industry testing.

Téléchat has a limitation in that it depends on model support. We assume the source and architecture models support the instructions under test. We found a bug in an unofficial implementation of the Armv7 model [65] that prior techniques miss. Since prior work [118, 41, 160] depends only on source models, this limitation is a consequence of testing using models rather than hardware. More generally we depend on models supporting all source or

assembly instructions in our tests.

Testing under models of source and processor architectures improves the likelihood of detecting bugs compared to testing on hardware. By employing models based on the C/C++ language standards and processor architectures, we obtain oracles that closely align with official specifications. While testing can only reveal the presence of bugs [112], it makes the process of identifying and fixing them repeatable. As the complexity of modern processors continues to rise, we expect that testing based on specifications will become more important.

Mix testing: We present the *mix testing* technique and `atomic-mixer` tool that checks whether compiler mappings (from C/C++ atomic operations to assembly sequences) are interoperable. In the past, C/C++ litmus tests were typically compiled under one mapping. Testing the interoperability of constituent C/C++ operations and their mappings to different (compatible) architectures had not been considered beyond Sewell's [144] mappings web-page. A *mixing bug* arises if any of the mixed tests exhibits a concurrency-related compiler bug in relation to the source litmus test. We found a number of new mixing bugs that cannot be caught by prior work, and worked with Arm's engineers to publish an Atomics Application binary interface [69].

The limitation of Mix testing is that it requires expertise and a lot of resources. For mix testing to be successful we rely on experts to pick compiler profiles that induce different atomics mappings, and we require a lot of computer time and space to generate the litmus tests reachable from those profiles. Fortunately, only a few compiler profiles introduce *new* atomics mappings, and so we only need to test those. Moreover, atomics mappings for a given compiler and architecture change rarely, perhaps twice a year, or whenever new instructions are introduced. As such, mix testing only needs to be run when mappings are changed, which amortises the cost over the long term.

One implication of mix testing is that it is necessary while multiple mappings exist. Our development of mix testing exposes the complexity of testing concurrent compilation, emphasising that the issue cannot be solved by testing

atomic mappings in isolation. Instead, it requires testing within the context of a vast number of possible mappings. This challenge will remain relevant for as long as compilers support multiple atomic mappings.

Another implication of mix testing is that interoperability is reducible to checking a specification. The complexity of mix testing can be alleviated by defining an atomic application binary interface (ABI). The ABI simplifies mix testing by narrowing down the number of implementations to test; though the test space remains exponential in theory; the ABI provides a useful validation framework for engineers to use in practice.

Assessing the limitations of models and simulators: We explore the limitations of using models and their simulators when put to the task of compiler testing. Téléchat and `atomic-mixer` have successfully found numerous concurrency-related compiler bugs. Despite their effectiveness in the validation of small concurrent litmus tests, today's executable models and simulators are limited in their applicability if they are unsound or incomplete. We conducted empirical studies on the interaction between concurrency compilation and other domains. We found bugs that are missed, models that do not support language features, and behaviours otherwise overlooked by axiomatic model simulators. In doing so we highlight the limits of the `herd` simulator and its models when used as parts of compiler testing oracles. We highlight some of the lessons we learned in using tools and models to solve problems faced by engineers in Arm's compiler teams.

An implication of this work is that today's tools and models need more work. Given the current limitations of available tools we believe that concurrency compilation testing cannot yet be automated to the same extent as sequential compilation testing. There is a wealth of fuzzers, generators, and test tools for sequential programs in use by engineers and experts alike. Our work suggests that operating concurrency testing tools still require an expert in the loop to either operate tools, improve them, or understand their output. We anticipate that significant engineering effort will be needed to develop the tools to a point

where they can be used by engineers who may not be specialists in concurrency. Chapter 5 examines several real-world challenges that engineers face in this domain, which require considerable effort to address.

Bugs: In total, we have so far found and reported nine concurrency-related compiler bugs and mixing bugs in LLVM and GCC. These bugs have either been fixed, or are currently triaged for fixing by Arm's engineers. This compares favourably with the numbers of bugs found by prior concurrency testing works.

7.2 Further Work

We end by outlining potential lines of inquiry.

Fuzzing source programs: We do not fuzz, or modify tests to increase the chance of observing bugs in, our source programs. Prior work [118, 41, 160] applies semantic-preserving transformations to small litmus tests, turning them into tests several hundred lines of code in size. Despite this, the number of bugs we found compares favourably with the state of the art. It is worth studying whether our technique is successful when fuzzing too.

Scaling simulation to real-world programs: We test compilation using small litmus tests as `herd` is designed to work on such tests. We do not consider complex optimisations however, which require large programs to induce bugs. Further, it is not yet clear to whether bugs that arise in large programs are reducible to minimal examples (along the lines of `creduce` [130]), and whether reduced tests can still trigger buggy code paths in the compiler, while being small enough to run in regular automated testing. Such questions may inform the development of scalable techniques.

Mix testing other architectures and domains: Mix testing is a general idea we expect to apply elsewhere. Indeed, in Chapter 6 we discuss related works where other authors independently use the term *mixed programs* (as opposed to our mix tests). Broadly speaking mixing can occur in sequential programs, or in any other domain. For instance, we are not aware of any mix testing of sequential programs using, for instance, CSmith [162].

Figure	Cycle Specification
Figure 3.11	`"PodWR Acq Fre Rel PodWR Acq Fre Rel"`
Figure 7.2	`"FenceReldWWRfeRlxRlxFenceAcqd?RFreRelRlx"`

Figure 7.1: Cyclic specifications of litmus tests.

Test generation for indirect accesses: We found some bug-inducing tests in this thesis manually. The running example in Chapter 3 is not automatically generated by today's tools (copied into Figure 7.2 below). Such tools generate tests from minimal specifications of explicit executions (see Chapter 6). The bug we found arises due to the absence of a read. It is worth considering whether it is possible to extend cyclic specifications to handle bugs that arise only through indirect accesses to data, or whether complementary techniques such as fuzzing are necessary.

Example 7.2.1. Using the cycle syntax of `diy` [15] we specify two tests in Figure 7.1. Note the explicit reads (`R`), writes (`W`), and fence (`Fence`) operations. Figure 3.11 is specified as follows. On thread `P0`, in program order (`Po`) there are two accesses to different (`d`) locations, where the first is a write (`W`) and the second is a read (`R`). On `P1` the first write, proceeds after the external read (`Fre`) on `P0`. Thread `P1` has the same pattern as `P0` (`PodWR`), and each thread has acquire (`Acq`) and release (`Rel`) annotations on its accesses. Finally, the write of `P0` proceeds after the read on `P1` with another `Fre` closing the cycle.

Example 7.2.2. Consider the bug that motivated Chapter 3. Figure 7.2 is a variant of message passing (MP) where the `exchange` operation discards the read of `y` on `P1` and instead writes 2 to `y` and checks `y` in the final state. This example is not produced by generators the use cycles as specifications [71, 67].

It is not clear that Figure 7.2 can be specified using `diy`. The problem is that Figure 7.2 explicitly omits the read of a read-modify-write (RMW), instead relying on indirect observation through `y`. Of course a special syntax may be created for readless-RMW operations, but does not change the problem in the current theory. The problem is that the bug arises outside the scope of

C/C++ Litmus Test

```
{ *x = 0, *y = 0 }

P0 () {
  store(x,1,rlx);
  fence(rel);
  store(y,1,rlx);
}
P1 () {
  exchange(y,2,rel);
  fence(acq);
  int r0 = load(x,rlx);
}

exists (P1:r0=0 /\ y=2)
```

C/C++ Outcomes

```
{ P1:r0=0, y=1 }
{ P1:r0=1, y=1 }
{ P1:r0=1, y=2 }
```

Assembly Litmus Test

```
{ *x = 0, *y = 0 }

P0               |P1
MOV W9,#1        |MOV W9,#2
STR W9,[X%P0_x]|SWPL W9,WZR,[X%P1_y]
DMB ISH          |DMB ISHLD
STR W9,[X%P0_y]|LDR W8,[X%P1_x]

exists (P1:W8=0 /\ y=2)
```

AArch64 Outcomes

```
{ P1:X8=0, y=1 }
!!{ P1:X8=0, y=2 }!!
{ P1:X8=1, y=1 }
{ P1:X8=1, y=2 }
```

Figure 7.2: Figure 3.1 repeated here.

the explicit cyclic specification approach used by state of the art generators [15, 158, 104]. It is not surprising that special (but not unique) cases analogous to Figure 7.2 exist. The success of randomised fuzzing and differential testing [162, 99, 95] in finding concurrency bugs is partly explained by their not being constrained to a particular test specification pattern (such as cycles). It stands to reason that cycles can miss compiler bugs off-the-beaten path. That said specifying, let alone testing, concurrent programs is tricky [38] and it is no surprise that cycles still drive the state of the art.

We conclude that more work is needed on concurrent test generators. Dead code elimination is a common compiler optimisation targeted by fuzzers. Likewise, message passing is well-understood. We expect that the bug induced by Figure 7.2 would be caught by first generating an MP test, and second fuzzing it in line with Windsor et al. [160, 159]'s metamorphic testing approach.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer*, 29, 12, 66–76. doi:10.1109/2.546611.

[2] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering–a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (ISCA '90). ACM, Seattle, Washington, USA, 2–14. ISBN: 0-89791-366-3. doi:10.1145/325164.325100.

[3] Yehuda Afek, Geoffrey M. Brown, and Michael Merritt. 1993. Lazy Caching. *ACM Trans. Program. Lang. Syst.*, 15, 1, 182–205. doi:10.1145/151646.151651.

[4] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '15). ACM, Istanbul, Turkey, 577–591. ISBN: 9781450328357. doi:10.1145/2694344.2694391.

[5] Jade Alglave, Patrick Cousot, and Luc Maranget. 2016. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531. http://arxiv.org/abs/1608.07531 arXiv: 1608.07531.

[6] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.*, 43, 2, Article 8, (July 2021), 54 pages. doi:10.1145/3458926.

[7] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2008. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming* (DAMP '09). ACM, Savannah, GA, USA, 13–24. ISBN: 978-1-60558-417-1. doi:10.1145/1481839.1481842.

[8] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion. *ACM Trans. Program. Lang. Syst.*, 39, 2, Article 6, (May 2017), 38 pages. doi:10.1145/2994593.

[9] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013. Software Verification for Weak Memory via Program Transformation. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (ESOP'13). Springer-Verlag, Rome, Italy, 512–532. ISBN: 9783642370359. doi:10.1007/978-3-642-37036-6_28.

[10] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (CAV 2013). Springer-Verlag New York, Inc., Saint Petersburg, Russia, 141–157. ISBN: 978-3-642-39798-1. doi:10.1007/978-3-642-39799-8_9.

[11] Jade Alglave and Luc Maranget. 2022. `ARM` memory model. `https://github.com/herd/herdtools7/blob/master/herd/libdir/arm.cat`. (2022).

[12] Jade Alglave and Luc Maranget. 2021. herdtools7. `https://github.com/herd/herdtools7`. Accessed: 2019-10-06. (2021).

[13] Jade Alglave and Luc Maranget. 2021. NEON Architecture Tests. `https://github.com/herd/herdtools7/tree/master/herd/tests/instructions/AArch64.neon`. Accessed: 2022-11-29. (2021).

[14] Jade Alglave and Luc Maranget. 2021. SVE Architecture Pull Request. `https://github.com/herd/herdtools7/pull/414`. (2021).

[15] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in Weak Memory Models (Extended Version). *Form. Methods Syst. Des.*, 40, 2, (Apr. 2012), 170–205. doi:10.1007/s10703-011-0135-z.

[16] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software* (TACAS'11/ETAPS'11). Springer-Verlag, Saarbrücken, Germany, 41–44. ISBN: 978-3-642-19834-2. `http://dl.acm.org/citation.cfm?id=1987389.1987395`.

[17] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36, 2, Article 7, (July 2014), 7:1–7:74. doi:10.1145/2627752.

[18] Arm-Limited. 2023. *Arm Architecture Reference Manual*. Arm Limited, Cambridge, UK. ISBN: 0201737191. `https://developer.arm.com/documentation/ddi0487/latest/`.

[19] Arm-Limited. 2021. Armv8 `aarch64` memory model. `https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat`. (2021).

[20] Arm-Limited. 2024. Load-Acquire and Store-Release instructions. `https://developer.arm.com/documentation/102336/0100/Load-Acquire-and-Store-Release-instructions`. (2024).

[21] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I* (Lecture Notes in Computer Science). Alexandra Silva and K. Rustan M. Leino, (Eds.) Vol. 12759. Springer, 303–316.

[22] Alasdair Armstrong et al. 2019. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. In number POPL, Article 71. Vol. 3. Association for Computing Machinery, New York, NY, USA, (Jan. 2019), 31 pages. doi:10.1145/3290384.

[23] Alasdair Armstrong et al. 2023. The Sail Instruction-Set Architecture (ISA) specification language. (2023).

[24] Arvind and Jan-Willem Maessen. 2006. Memory Model = Instruction Reordering + Store Atomicity. In *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*. IEEE Computer Society, 29–40. doi:10.1109/ISCA.2006.26.

[25] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Softw. Eng.*, 41, 5, (May 2015), 507–525. doi:10.1109/TSE.2014.2372785.

[26] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2022. c11_orig Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/c11_orig.cat. (2022).

[27] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2022. c11_partialSC Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/c11_partialSC.cat. (2022).

[28] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '16). Association for Computing Machinery, St. Petersburg, FL, USA, 634–648. ISBN: 9781450335492. doi:10.1145/2837614.2837637.

[29] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '12). Association for Computing Machinery, Philadelphia, PA, USA, 509–520. ISBN: 9781450310833. doi:10.1145/2103656.2103717.

[30] Mark Batty, Scott Owens, Jean Pichon-Pharabod, Susmit Sarkar, and Peter Sewell. 2019. CppMem: C/C++ memory model exploration tool. [web interface]. (2019). http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem.

[31] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Thomas Ball and Mooly Sagiv, (Eds.) ACM, 55–66. doi:10.1145/1926385.1926394.

[32] Mark John Batty. 2015. *The C11 and C++11 concurrency model*. Ph.D. Dissertation. University of Cambridge, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.708458.

[33] Martin Beck, Koustubha Bhat, Lazar Stricevic, Geng Chen, Diogo Behrens, Ming Fu, Viktor Vafeiadis, Haibo Chen, and Hermann Härtig. 2023. AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, (Eds.) ACM, 61–73. doi:10.1145/3575693.3579849.

[34] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. 2011. Nitpicking C++ Concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming* (PPDP '11). ACM, Odense, Denmark, 113–124. ISBN: 978-1-4503-0776-5. doi:10.1145/2003476.2003493.

[35] Hans Boehm. [n. d.] Defang and deprecate memory_order::consume. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3475r1.pdf. Accessed: 2025-27-11. ().

[36] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '08). Association for Computing Machinery, Tucson, AZ, USA, 68–78. ISBN: 9781595938602. doi:10.1145/1375581.1375591.

[37] Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding out-of-Thin-Air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness* (MSPC '14) Article 7. Association for Computing Machinery, Edinburgh, United Kingdom, 6 pages. ISBN: 9781450329170. doi:10.1145/2618128.2618134.

[38] Soham Chakraborty, Shankara Narayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2024. How Hard Is Weak-Memory Testing? In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* number POPL 51, Article 66. Vol. 8. Association for Computing Machinery, New York, NY, USA, (Jan. 2024), 32 pages. doi:10.1145/3632908.

[39] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (CGO '17). IEEE Press, Austin, USA, 100–110. ISBN: 978-1-5090-4931-8. http://dl.acm.org/citation.cfm?id=3049832.3049844.

[40] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-air Reads with Event Structures. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages* number POPL 46, Article 70. Vol. 3. ACM, New York, NY, USA, (Jan. 2019), 70:1–70:28. doi:10.1145/3290383.

[41] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating Optimizations of Concurrent C/C++ Programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (CGO '16). ACM, Barcelona, Spain, 216–226. ISBN: 978-1-4503-3778-6. doi:10.1145/2854038.2854051.

[42] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.*, 53, 1, Article 4, (Feb. 2020), 36 pages. doi:10.1145/3363562.

[43] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential reasoning for optimizing compilers under weak memory concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI 2022). Association for Computing Machinery, San Diego, CA, USA, 213–228. ISBN: 9781450392655. doi:10.1145/3519939.3523718.

[44] Clang. 2024. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed: 2024-09-27. (2024).

[45] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings* (Lecture Notes in Computer Science). Kurt Jensen and Andreas Podelski, (Eds.) Vol. 2988. Springer, 168–176. doi:10.1007/978-3-540-24730-2_15.

[46] Simon Colin. 2022. RC11 Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/rc11.cat. (2022).

[47] William W. Collier. 1992. *Reasoning about parallel architectures*. Prentice Hall. ISBN: 978-0-13-766098-8.

[48] Will Deacon. 2014. Arm64: atomics: fix use of acquire + release for full barrier semantics. http://lists.infradead.org/pipermail/linux-arm-kernel/2014-February/229588.html. (2014).

[49] Paulo Emílio de Vilhena, Ori Lahav, Viktor Vafeiadis, and Azalea Raad. 2024. Extending the C/C++ Memory Model with Inline Assembly. In *2024 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* number OOPSLA2, Article 309. Vol. 8. Association for Computing Machinery, New York, NY, USA, (Oct. 2024), 27 pages. doi:10.1145/3689749.

[50] Wilco Dijkstra. 2024. Alignment of _atomic structs incompatible between gcc and llvm. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=115954. (2024).

[51] Wilco Dijkstra. 2023. Bug 108891 - libatomic: AArch64 SEQ_CST 16-byte load missing barrier. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108891. (2023).

[52] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. In *2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* number OOPSLA, Article 93. Vol. 1. Association for Computing Machinery, New York, NY, USA, (Oct. 2017), 29 pages. doi:10.1145/3133917.

[53] Nathaniel Wesley Filardo et al. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ASPLOS '24). Association for Computing Machinery, La Jolla, CA, USA, 251–268. ISBN: 9798400703850. doi:10.1145/3620665.3640416.

[54] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '16). ACM, St. Petersburg, FL, USA, 608–621. ISBN: 978-1-4503-3549-2. doi:10.1145/2837614.2837615.

[55] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size concurrency: arm, power, c/c++11, and sc. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (POPL '17). Association for Computing Machinery, Paris, France, 429–442. ISBN: 9781450346603. doi:10.1145/3009837.3009839.

[56] GCC-Bugzilla. 2019. Atomic store of ___int128 is not lock free on aarch64. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=70814. Accessed: 2019-10-29. (2019).

[57] Luke Geeson. 2023. [AArch64]: 128-bit Const Atomic Load implemented using Store Pair instruction, induces Runtime Crash on Arm AArch64. https://github.com/llvm/llvm-project/issues/61770. (2023).

[58] Luke Geeson. 2023. [AArch64]: 128-bit seq_cst load can be reordered before prior RMW operations under LSE and above. https://github.com/llvm/llvm-project/issues/62652. Accessed: 2023-05-11. (2023).

[59] Luke Geeson. 2024. [AArch64]: 128-bit Sequentially Consistent load allows reordering before prior store when armv8 and armv8.4 implementations are Mixed. https://github.com/llvm/llvm-project/issues/81978. (2024).

[60] Luke Geeson. 2023. [AArch64]: Atomic Exchange Allows Reordering past Acquire Fence. https://github.com/llvm/llvm-project/issues/68428. (2023).

[61] Luke Geeson. 2023. [AArch64][CodeGen]: LD{AX}P/S{LX}TP endian swap. https://github.com/llvm/llvm-project/issues/61431. (2023).

[62] Luke Geeson. 2024. [Armv7-a]: Sequentially Consistent Load Allows Reordering of Prior Store when Implementations are Mixed. https://github.com/llvm/llvm-project/issues/65541#issuecomment-1709229837. (2024).

[63] Luke Geeson. 2024. [Armv7/v8 Mixing Bug]: 64-bit Sequentially Consistent Load can be Reordered before Store of RMW when v7 and v8 Implementations are Mixed. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=111416. (2024).

[64] Luke Geeson. 2022. Aarch32 memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch32.cat. (2022).

[65] Luke Geeson. 2022. Added dmb ish to arm model. https://github.com/herd/herdtools7/pull/385. Accessed: 2022-11-26. (2022).

[66] Luke Geeson. 2023. branch delay slots are not filled with atomic stores. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=110573. (2023).

[67] Luke Geeson. 2024. Weak Memory Demands Model-based Compiler Testing. In *The Future of Weak Memory Workshop (FOWM), Held as Part of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2024), London, United Kingdom, January 14–21.* paper and talk: https://lukegeeson.com/talks/2024-01-15-POPL24/. (Jan. 2024). arXiv: doi.org/2401.09474 [cs.PL].

[68] Luke Geeson, James Brotherston, Wilco Dijkstra, Alastair F. Donaldson, Lee Smith, Tyler Sorensen, and John Wickerson. 2024. Mix Testing: Specifying and Testing ABI Compatibility of C/C++ Atomics Implementations. In *2024 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* number OOPSLA2. Vol. 8. pre-print: https://lukegeeson.com/assets/publications/oopsla24/paper.pdf. Association for Computing Machinery, 442–467. doi:10.1145/3689727.

[69] Luke Geeson and Wilco Dijkstra. 2024. C/C++ Atomics Application Binary Interface Standard for the Arm® 64-bit Architecture. https://github.com/ARM-software/abi-aa/releases/tag/2024Q3. (2024).

[70] [SW] Luke Geeson and Lee Smith, CGO Artefact for Compiler Testing With Relaxed Memory Models Dec. 2023. doi:10.5281/zenodo.10204529, URL: https://zenodo.org/records/10411403.

[71] Luke Geeson and Lee Smith. 2024. Compiler Testing with Relaxed Memory Models. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. pre-print: https://lukegeeson.com/assets/publications/cgo24/paper.pdf. IEEE Computer Society, (Mar. 2024), 334–348. ISBN: 979-8-3503-9509-9. doi:10.1109/CGO57630.2024.10444836.

[72] [SW] Geeson, Luke and Brotherston, James and Dijkstra, Wilco and Donaldson, Alastair F. and Smith, Lee and Sorensen, Tyler and Wickerson, John, OOPSLA Artifact for Mix Testing: Specifying and Testing ABI Compatibility Of C/C++ Atomics Implementations Dec. 2024. doi:10.5281/zenodo.12671272, URL: https://zenodo.org/records/12671272.

[73] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation* number PLDI'44, Article 153. Vol. 7. Association for Computing Machinery, New York, NY, USA, (June 2023), 24 pages. doi:10.1145/3591267.

[74] David Goldblatt. 2019. There might not be an elegant OOTA fix. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1916r0.pdf. Accessed: 2021-10-06. (2019).

[75] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, and Martin Uecker. 2022. N3005: A Provenance-aware Memory Object Model for C. Working Draft Technical Specification ISO/IEC TS 6010:2023 (E). ISO/IEC JTC1/SC22/WG14 N3005 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n3005.pdf. (June 2022).

[76] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. 2004. Tsotool: a program for verifying memory systems using the memory consistency model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (ISCA '04). IEEE Computer Society, München, Germany, 114. ISBN: 0769521436.

[77] Mark Harman and Robert Hierons. 2001. An overview of program slicing. *Software Focus*, 2, 3, 85–92. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/swf.41. doi:https://doi.org/10.1002/swf.41.

[78] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. 1999. Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems. In *Proceedings of the 11th International Conference on Computer Aided Verification* (CAV '99). Springer-Verlag, Berlin, Heidelberg, 301–315. ISBN: 3540662022.

[79] Intel. 2013. Pin. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html. (2013).

[80] Open ISO-C-Std. 2022. ISO/IEC 9899:201x. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2912.pdf. (2022).

[81] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11, 2, (Apr. 2002), 256–290. doi:10.1145/505145.505149.

[82] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2019. An Empirical Validation of Oracle Improvement. *IEEE Transactions on Software Engineering*, PP, (Aug. 2019), 1–1. doi:10.1109/TSE.2019.2934409.

[83] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. Andreas Zeller and Abhik Roychoudhury, (Eds.) ACM, 247–258. doi:10.1145/2931037.2931062.

[84] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.*, 37, 5, (Sept. 2011), 649–678. doi:10.1109/TSE.2010.62.

[85] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (POPL 2017). ACM, Paris, France, 175–189. ISBN: 978-1-4503-4660-3. doi:10.1145/3009837.3009850.

[86] Theodoros Kasampalis. 2021. *Translation validation for compilation verification.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[87] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. 2023. Kater: automating weak memory model metatheory and consistency checking. *Proc. ACM Program. Lang.*, 7, POPL, Article 19, (Jan. 2023), 29 pages. doi:10.1145/3571212.

[88] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2019). Association for Computing Machinery, Phoenix, AZ, USA, 96–110. ISBN: 9781450367127. doi:10.1145/3314221.3314609.

[89] Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I.* Springer-Verlag, Berlin, Heidelberg, 427–440. ISBN: 978-3-030-81684-1. doi:10.1007/978-3-030-81685-8_20.

[90] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* Albert Cohen and Martin T. Vechev, (Eds.) ACM, New York, NY, USA, 618–632. doi:10.1145/3062341.3062352.

[91] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28, 9, (Sept. 1979), 690–691. doi:10.1109/TC.1979.1675439.

[92] William B. Langdon and David Clark. 2024. Genetic Improvement of Last Level Cache. In *Genetic Programming: 27th European Conference, EuroGP 2024, Held as Part of EvoStar 2024, Aberystwyth, UK, April 3–5, 2024, Proceedings.* Springer-Verlag,

Aberystwyth, United Kingdom, 209–226. ISBN: 978-3-031-56956-2. doi:10.1007/978-3-031-56957-9_13.

[93] Andrei Lascu, Matt Windsor, Alastair F. Donaldson, Tobias Grosser, and John Wickerson. 2021. Dreaming up Metamorphic Relations: Experiences from Three Fuzzer Tools. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*, 61–68. doi:10.1109/MET52542.2021.00017.

[94] Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. 2019. Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. In *Computer Aided Verification*. Isil Dillig and Serdar Tasiran, (Eds.) Springer International Publishing, Cham, 387–397. ISBN: 978-3-030-25540-4.

[95] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Michael F. P. O'Boyle and Keshav Pingali, (Eds.) ACM, 216–226. doi:10.1145/2594291.2594334.

[96] Sung-Hwan Lee. 2023. A miscompilation bug in LICMPass (concurrency). https://github.com/llvm/llvm-project/issues/64188. (2023).

[97] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020). Association for Computing Machinery, London, UK, 362–376. ISBN: 9781450376136. doi:10.1145/3385412.3386010.

[98] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52, 7, (July 2009), 107–115. doi:10.1145/1538788.1538814.

[99] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '15). ACM, Portland, OR, USA, 65–76. ISBN: 9781450334686. doi:10.1145/2737924.2737986.

[100] LLVM. 2024. AArch64 Dead register definitions. https://llvm.org/docs/doxygen/AArch64DeadRegisterDefinitionsPass_8cpp_source.html. (2024).

[101] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI 2021). Association for Computing Machinery, Virtual, Canada, 65–79. ISBN: 9781450383912. doi:10.1145/3453483.3454030.

[102] Daniel Lustig, Simon Cooksey, and Olivier Giroux. 2022. Mixed-proxy extensions for the NVIDIA PTX memory consistency model: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (ISCA '22). Association for Computing Machinery, New York, New York, 1058–1070. ISBN: 9781450386104. doi:10.1145/3470496.3533045.

[103] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '19). Association for Computing Machinery, Providence, RI, USA, 257–270. ISBN: 9781450362405. doi:10.1145/3297858.3304043.

[104] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '17). Association for Computing Machinery, Xi'an, China, 661–675. ISBN: 9781450344654. doi:10.1145/3037697.3037723.

[105] Sela Mador-Haim et al. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Proceedings of the 24th International Conference on Computer Aided Verification* (CAV'12). Springer-Verlag, Berkeley, CA, 495–512. ISBN: 9783642314230.

[106] Luc Maranget. 2022. RISC-V Memory Model. `https://github.com/herd/herdtools7/blob/master/herd/libdir/riscv.cat`. (2022).

[107] Luc Maranget and Jade Alglave. 2022. Encoding of SC Memory Model. `https://github.com/herd/herdtools7/blob/master/herd/libdir/sc.cat`. (2022).

[108] Luc Maranget and Jade Alglave. 2022. IBM PowerPC Memory Model. `https://github.com/herd/herdtools7/blob/master/herd/libdir/ppc.cat`. (2022).

[109] Luc Maranget and Jade Alglave. 2023. MIPS Memory Model. `https://github.com/herd/herdtools7/blob/master/herd/libdir/mips.cat`. (2023).

[110] Luc Maranget and Jade Alglave. 2023. x86-64 Memory Model. `https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso-mixed.cat`. (2023).

[111] John McCarthy and James A. Painter. 1966. Correctness of a compiler for arithmetic expressions. In https://api.semanticscholar.org/CorpusID:60914848.

[112] Robert M. McClure. 1969. Software engineering techniques. *Nato Software Engineering Conference*, 15–17.

[113] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C Semantics and Pointer Provenance. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. Proc. ACM Program. Lang. 3, POPL, Article 67. Also available as ISO/IEC JTC1/SC22/WG14 N2311. (Jan. 2019). doi:10.1145/3290380.

[114] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '16). ACM, Santa Barbara, CA, USA, 1–15. ISBN: 978-1-4503-4261-2. doi:10.1145/2908080.2908081.

[115] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33, 12, (Dec. 1990), 32–44. doi:10.1145/96267.96279.

[116] Mono Project. 2024. The Mono Project, atomic source code. https://github.com/mono/mono/blob/44e6226c31d8ffcae58f81350d71a728edecfe22/mono/utils/atomic.h#L209. Accessed: 2024-02-29. (2024).

[117] J. Strother Moore. 1989. A mechanically verified language implementation. *J. Autom. Reason.*, 5, 4, (Nov. 1989), 461–492.

[118] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13). ACM, Seattle, Washington, USA, 187–196. ISBN: 978-1-4503-2014-6. doi:10.1145/2491956.2491967.

[119] National Information Standards Organisation. 2022. ANSI/NISO Z39.104-2022, CRediT, Contributor Roles Taxonomy, (Feb. 2022). https://www.niso.org/standards-committees/credit.

[120] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (PLDI '00). ACM, Vancouver, British Columbia, Canada, 83–94. ISBN: 1-58113-199-2. doi:10.1145/349299.349314.

[121] Vincent P. J. Nimal. 2014. *Static analyses over weak memory*. Ph.D. Dissertation. University of Oxford, UK. http://ora.ox.ac.uk/objects/uuid:469907ec-6f61-4015-984e-7ca8757b992c.

[122] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In Advances in Computers. Vol. 112. Atif M. Memon, (Ed.) Elsevier, 275–378. doi:https://doi.org/10.1016/bs.adcom.2018.03.015.

[123] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 128–148. ISBN: 9783642548321. doi:10.1007/978-3-642-54833-8_8.

[124] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems* (TACAS '98). Springer-Verlag, Berlin, Heidelberg, 151–166. ISBN: 3540643567.

[125] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. In *Proceedings of the 46th. ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages* number POPL'19, Article 69. Vol. 3. Association for Computing Machinery, New York, NY, USA, (Jan. 2019), 31 pages. doi:10.1145/3290382.

[126] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2018. BMC with Memory Models as Modules. In (Oct. 2018). doi:10.23919/FMCAD.2018.8603021.

[127] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* number POPL'18, Article 19. Vol. 2. ACM, New York, NY, USA, (Dec. 2017), 19:1–19:29. doi:10.1145/3158107.

[128] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying systems C code with separation-logic refinement types. In *Proceedings of the 50th ACM SIGPLAN Symposium on Principles of Programming Languages*. (Jan. 2023). doi:10.1145/3571194.

[129] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022. Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. In *Proceedings of the 49th. ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages* number POPL'22, Article 22. Vol. 6. Association for Computing Machinery, New York, NY, USA, (Jan. 2022), 31 pages. doi:10.1145/3498683.

[130] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012* number 6. Vol. 47. ACM, New York, NY, USA, 335–346. doi:10.1145/2345156.2254104.

[131] Alastair Reid. 2017. Who Guards the Guards? Formal Validation of the Arm v8-m Architecture Specification. In *2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* number OOPSLA, Article 88. Vol. 1. Association for Computing Machinery, New York, NY, USA, (Oct. 2017), 88:1–88:24. doi:10.1145/3133912.

[132] Alastair Reid et al. 2016. End-to-end verification of processors with isa-formal. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II* (Lecture Notes in Computer Science). Swarat Chaudhuri and Azadeh Farzan, (Eds.) Vol. 9780. Springer, 42–58. doi:10.1007/978-3-319-41540-6_3.

[133] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at large: A survey of engineering of formally verified software. *Found. Trends Program. Lang.*, 5, 2-3, 102–281. doi:10.1561/2500000045.

[134] Rodrigo C. O. Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Lasagne: A Static Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI 2022). Association for Computing Machinery, San Diego, CA, USA, 888–902. ISBN: 9781450392655. doi:10.1145/3519939.3523719.

[135] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Mary W. Hall and David A. Padua, (Eds.) ACM, 175–186. doi:10.1145/1993498.1993520.

[136] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The Semantics of x86-CC Multiprocessor Machine Code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '09). ACM, Savannah, GA, USA, 379–391. ISBN: 978-1-60558-379-2. doi:10.1145/1480881.1480929.

[137] Susmit Sarkar et al. 2023. RMEM: Executable operational concurrency model exploration tool for ARMv8, RISC-V, Power, and x86. [web interface]. (2023).

[138] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, (June 2012), 309–318. doi:10.5555/2342821.2342849.

[139] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (WBIA '09). Association for Computing Machinery, New York, New York, USA, 62–71. ISBN: 9781605587936. doi:10.1145/1791194.1791203.

[140] Jaroslav Ševčík. 2011. Safe Optimisations for Shared-memory Concurrent Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11). ACM, San Jose, California, USA, 306–316. ISBN: 978-1-4503-0663-8. doi:10.1145/1993498.1993534.

[141] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM*, 60, 3, Article 22, (June 2013), 50 pages. doi:10.1145/2487241.2487248.

[142] Peter Sewell and Shaked Flur. 2025. Relaxed Memory Multicore Semantics course notes. https://www.cl.cam.ac.uk/ pes20/multicore-sewell-notes.pdf. (2025).

[143] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53, 7, (July 2010), 89–97. doi:10.1145/1785414.1785443.

[144] Peter Sewell and Jaroslav Ševčík. 2016. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. (2016).

[145] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13). Association for Computing Machinery, Seattle, Washington, USA, 471–482. ISBN: 9781450320146. doi:10.1145/2491956.2462183.

[146] Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.*, 10, 2, (Apr. 1988), 282–312. doi:10.1145/42190.42277.

[147] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings* (Lecture Notes in Computer Science). Ilya Sergey, (Ed.) Vol. 13240. Springer, 143–173. doi:10.1007/978-3-030-99336-8_6.

[148] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO '15). IEEE Computer Society, San Francisco, California, 46–55. ISBN: 9781479981618.

[149] Bjarne Stroustrup. 2013. *The C++ Programming Language*. (4th ed.). Addison-Wesley Professional. ISBN: 0321563840.

[150] [SW] Ole Tange, GNU Parallel 20230222 Feb. 2023. doi:10.5281/zenodo.7668338, URL: https://doi.org/10.5281/zenodo.7668338.

[151] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1, 2, (June 1972), 146–160. doi:10.1137/0201010.

[152] Microsft MSRC Team. 2024. We need a safer systems programming language. https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/. Accessed: 2024-09-08. (2024).

[153] Kyrylo Tkachov. 2024. Enabling the LDAPR instructions for C/C++ compilers. https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/enabling-rcpc-in-gcc-and-llvm. (2024).

[154] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-graph Translation Validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11). ACM, San Jose, California, USA, 295–305. ISBN: 978-1-4503-0663-8. doi:10.1145/1993498.1993533.

[155] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. 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, POPL 2015, Mumbai, India, January 15-17, 2015*. Sriram K. Rajamani and David Walker, (Eds.) ACM, Mumbai, India, 209–220. doi:10.1145/2676726.2676995.

[156] Andrew Waterman and Krste Asanović. 2019. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20191213. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf. (2019).

[157] The Whitehouse. 2024. Back To The Building Blocks: A Path Toward Secure and Measurable Software. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf. Accessed: 2024-09-24. (2024).

[158] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Giuseppe Castagna and Andrew D. Gordon, (Eds.) ACM, Paris, France, 190–204. ISBN: 978-1-4503-4660-3. doi:10.1145/3009837.3009838.

[159] Matt Windsor, Alastair F Donaldson, and John Wickerson. 2022. High-coverage metamorphic testing of concurrency support in C compilers. *Software Testing, Verification and Reliability*, e1812.

[160] Matt Windsor, Alastair F. Donaldson, and John Wickerson. 2021. C4: The C Compiler Concurrency Checker. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2021). ACM, Virtual, Denmark, 670–673. ISBN: 9781450384599. doi:10.1145/3460319.3469079.

[161] Shafik Yaghmour. 2019. P1705R0: Enumerating Core Undefined Behavior. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r0.html. Accessed: 2025-02-16. (2019).

[162] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11). ACM, San Jose, California, USA, 283–294. ISBN: 978-1-4503-0663-8. doi:10.1145/1993498.1993532.

[163] Y. Yang, Ganesh Gopalakrishnan, G. Lindstrom, and K. Slind. 2004. Nemos: a framework for axiomatic and executable specifications of memory consistency models. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 31–. doi:10.1109/IPDPS.2004.1302944.

[164] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '12). Association for Computing Machinery, Philadelphia, PA, USA, 427–440. ISBN: 9781450310833. doi:10.1145/2103656.2103709.

Appendix A

Téléchat Artifact

The appendix consists of the Téléchat tool and scripts provided with this paper. Téléchat builds on the herd tool-suite [12] and its models. As such the results are liable to change. This artifact [70] was submitted to the CGO conference artifact evaluation process alongside our paper [71]. We were awarded all badges:

[SW] Luke Geeson and Lee Smith, CGO Artefact for Compiler Testing With Relaxed Memory Models Dec. 2023. doi:`10.5281/zenodo.10204529`, URL: `https://zenodo.org/records/10411403`

Artifact Checklist

1. **Algorithm:** Téléchat.
2. **Program:** `l2c, c2s, s2l` [71] and herdtools [12].
3. **Compilation:** Includes LLVM 11, GCC 9.2, GCC 10.
4. **Models:** From herd toolsuite [12].
5. **Data Set:** Tests generated using provided `c11.conf`.
6. **Test Environment/Binary:** Docker Ubuntu 20.04.
7. **Hardware:** Either x86-64 or Arm AArch64 machines.
8. **Run-time State:** not sensitive to run-time state.

9. **Metrics:** Outcomes of executing tests under models.
10. **Output:** Console and `.log` files.
11. **Experiments:** Makefile provided reproduces results.
12. **Disk-space requirements:** 5 GB for Docker image, +100 GB for the large-scale study (§3.4.4).
13. **Time needed to prepare workflow**: Everything is ready.
14. **Time needed to complete experiments**: $\sim$ 10 hours.
15. **Licences:** CeCILL-B licence.
16. **Workflow Frameworks:** Makefile, GNU Parallel [150].
17. **Archived(DOI):** `https://doi.org/10.5281/zenodo.10204529`
18. **Available:** Zenodo or Docker Hub[1] [70].

Description

How Delivered

The artifact is available on Zenodo and consists of a Docker container with the Téléchat tool, compilers under test, and scripts required to reproduce results.

Hardware Dependencies

Either an Intel x86-64 or Arm AArch64 based machine. The artifact was tested using a MacBook Pro with a dual-core Intel i7 CPU, a Lenovo P720 with 2xIntel Xeon Gold 5120T CPUs (56 cores), a MacBook Air with an 8-core Apple M1 (Arm AArch64), a Cavium Thunder X2 with 2x28-core CPUs (Arm AArch64), and under x86-64 emulation (using the M1 machine).

[1] `https://hub.docker.com/r/lukeg101/telechat-artefact/tags`

Software Dependencies

Téléchat requires a Linux distribution such as Ubuntu. Including:

- The C/C++ compiler under test (multi-lib cross-compilers work best on multiple platforms).
- GNU binutils, e.g. `binutils-riscv64-linux-gnu`.
- GNU Parallel [150], libxml2, time, and libc6

Installation

1. Download and install Docker. For example on Ubuntu 20.04 you can install docker using the official guide[2]
2. Download `telechat-artefact-arch.tar.gz` from Zenodo (where `arch` is either `arm64` or `x86`).
3. Load the Docker container:

   ```
   > docker load -i telechat-artefact-arch.tar.gz
   ```

4. Run the Image:

   ```
   > docker run -it lukeg101/telechat-artefact
   ```

   This runs the Ubuntu image and mounts the current directory into the container at `artefact-output`.

Alternatively, if you wish to install from Docker Hub, we provide Intel x86-64 and Arm AArch64 builds:

```
> docker pull lukeg101/telechat-artefact:latest
```

Then run:

```
> docker run -it lukeg101/telechat-artefact:latest
```

[2] `https://docs.docker.com/desktop/install/ubuntu/`

Experiment Workflow

A `Makefile` drives the Téléchat toolchain, and examples of how to use it are provided in the `README.md`. For example, to run the "smoketest" in the docker container, type:

```
artefact> make examples
```

The Readme contains instructions on how to customise testing and generate different test benchmarks.

Updates since artifact publication

- The figure numbers have changed when adapting the paper and artifact documentation for the thesis. All figures that follow in this chapter are for figures in the thesis.
- We decided to change the spelling of 'artefact' to 'artifact' in the documentation, however the shell environment and files that were assessed by the evaluation committee retain the old spelling.
- The outcome `[0:r0=0; 1:r0=0;]` is the output of the `mcompare` tool. It corresponds to the outcome `{ P0:r0=0, P1:r0=0 }` in the thesis. The set notation in the thesis follows the formalism.
- Likewise, the output `[P0_r0]=0; [P1_r0]=0; [P2_r0]=0;` is from `herd` and corresponds to the outcome `{ P0_r0=0, P1_r0=0, P2_r0=0 }`.
- The paper used the terminology of positive of negative differences. This is unintuitive, and so we replace such terms with $\not\subseteq$ and $\not\supseteq$, respectively.

Paper Claims

1. Figure 3.7 (top left) has outcomes in Figure 3.7 (top right). under the RC11 model [90], when compiled for Arm AArch64, Figure 3.7 (bottom left) has the outcomes in Figure 3.7 (right) outcomes.

2. Windsor et al. miss [159] miss the load buffering behaviour of Figure 3.7. Téléchat observes it.

3. We exercise all the features in Table 3.1. when testing LLVM and GCC for the architectures listed.

4. We get the results in Table 3.2 under the RC11 model [90], but if we permit load-to-store reordering all $\not\subseteq$ differences disappear.

5. Compiling and Optimising Figure 3.10 using Téléchat enables its simulation to terminate in milliseconds.

A number of minor claims appear in the paper, like how we added a vector datatype to `herd`. To keep this appendix small we refer the reader to Téléchat generated tests that use these features. To validate the bug reports, please see our bug board[3]

Evaluation and Expected Results

We assume you are running with a clean directory.

Claim 1 (< 5 minutes on an Apple M1 machine) Please run:

```
artefact> make examples
```

Check the log:

```
artefact> cat artefact-output/Output/logs\
  /examples_int_C_tests_llvm-O3-AArch64_mcompare.log
```

The source and compiled program outcomes are tabulated, `LB004_examples/_int_C_tests` has new behaviour as shown in Figure A.1:

Claim 2 (< 1 minute checking manually) Windsor et al. [159] state: "*we experimented with using the stronger RC11 memory model of Lahav et al. [90] as the input to our test-case generator, RMEM [a simulator parameterised over*

[3] `https://lukegeeson.com/blog/2023-10-17-Telechat-Bug-Board/`

```
c11_[...]_tests   a64_[...]_tests
[0:r0=0; 1:r0=0;] +[P0_r0=1; P1_r0=1;]
[0:r0=0; 1:r0=1;]
[0:r0=1; 1:r0=0;]
```

Figure A.1: Outcomes of C/C++ and AArch64 LB tests.

architecture models, not part of c4] identified as a bug the 'load buffering' test that RC11 forbids, but C11 and AArch64 permit."
We observe load buffering (Figure 3.7) in **Claim 1**.

Claim 3 ($\sim$ 10 hours on a 224 core ThunderX2) Please run:

```
artefact> make all CONF_FILE=c11.conf
```

Warning: This requires a powerful machine to run.

The `Output` directory should have tests using the keywords: `fence`, `*x`, `if`, `load`, `store`, `clang-11`, `gcc-10`, `-O1`, `-O2`, `-O3`, `-march=armv7`, `-march=x86-64`, `-march=mips64`, `powerpc-linux-gnu`, `aarch64-linux-gnu`, `riscv64-pclinux-gnu`, and so on...

Claim 4 ($\sim$ 10 hours on a 224 core ThunderX2) Please run:

```
artefact> make all CONF_FILE=c11.conf
```

Once done, the numbers in Table 3.2 should match the $\not\subseteq$ and $\not\supseteq$ differences, observe that all the $\not\subseteq$ differences go away when we use the `rc11+lb.cat` model:

```
artefact> make all CONF_FILE=c11.conf CMEM=rc11+lb.cat
```

Warning: This requires a powerful machine to run.

Claim 5 ($<$ 5 minutes on an Apple M1 machine) Please run:

```
artefact> make examples
```

And then you can see the compiled (and optimised) Figure 3.10:

```
artefact> cat artefact-output/Output\
  /examples_int_C_tests/tgt/llvm-O3-AArch64\
  /3.LB004_examples_int_C_tests.litmus
```

Simulation timings are in the `herd` log:

```
artefact> cat artefact-output/Output \
  /examples_int_C_tests/tgt/      \
  llvm-O3-AArch64/all_a64_llvm-O3-\
  AArch64_examples_int_C_tests.log
```

Observe that simulation took ∼3 milliseconds (subject to your CPU clock speed and memory latency) in Figure A.2:

```
Test 3.LB004_examples_int_C_tests Allowed
States 8
[P0_r0]=0; [P1_r0]=0; [P2_r0]=0;
[P0_r0]=0; [P1_r0]=0; [P2_r0]=1;
[...]
Time 3.LB004_examples_int_C_tests 0.03
```

Figure A.2: Running the three thread LB test took 3 milliseconds.

On the other hand, Consider the `unoptimised.litmus` test, adapted from LLVM-11 code taken from godbolt.org[4] (which is the same as `3.LB004`) that we have not seen terminate after running for 1 hour on an Apple M1 machine:

```
artefact> make dnf
```

Warning: It is unclear whether herd terminates with this input

Experiment Customisation

You can customise the experiments when invoking Make:

- Generate different C/C++ tests using a config file (default: None, options: `c11.conf`, `c11_acq.conf`):

  ```
  artefact> make examples CONF_FILE=c11.conf
  ```

- Set source model (default `rc11.cat`, options: `c11_partialSC.cat`, `c11_simp.cat`, `rc11.cat`, `rc11+lb.cat`):

[4] `https://godbolt.org/z/G9b4Pq1YK`

```
artefact> make examples CMEM=c11_simp.cat
```

- Set simulation timeout, (default `120.0` seconds):

```
artefact> make examples TIMEOUT=1.0
```

- Test other compilers. Add a profile to `profiles.json`, add the profile name (such as `llvm-O3-AArch64`) to the `PROFILE` variable in the `Makefile`, add the `MODEL_`*profile* to the `Makefile`, and re-run `./build.sh && ./run.sh`.

Available Benchmarks

The benchmarks used can be generated by providing a `CONF_FILE` parameter to the `Makefile`:

- §3.4.4: `c11.conf`: for the large-scale differential testing.
- §3.4.6: `c11_acq.conf`: for the LDAPR case study.

Appendix B

Mix testing Artifact

The artifact consists of the scripts to reproduce figures in the paper. This artifact [72] was submitted to the OOPSLA conference artifact evaluation process alongside our paper [68]. We were awarded the available and functional badges:

[SW] Geeson, Luke and Brotherston, James and Dijkstra, Wilco and Donaldson, Alastair F. and Smith, Lee and Sorensen, Tyler and Wickerson, John, OOPSLA Artifact for Mix Testing: Specifying and Testing ABI Compatibility Of C/C++ Atomics Implementations Dec. 2024. doi:`10.5281/zenodo.12671272`, URL: `https://zenodo.org/records/12671272`

Artifact checklist

1. **Algorithm:** mix testing (Figure 4.2).
2. **Program:** The `herd` [12] simulator.
3. **Models:** From herd toolsuite [12].
4. **Data Set:** Tests provided in `expected` directory.
5. **Test Environment/Binary:** Docker Ubuntu 20.04.
6. **Hardware:** Either x86-64 or Arm AArch64 machines.
7. **Run-time State:** not sensitive to run-time state.

8. **Metrics:** Outcomes of executing tests under models.
9. **Output:** Console and `.log` files.
10. **Experiments:** Makefile provided reproduces results.
11. **Disk-space requirements:** 5 GB for Docker image.
12. **Time needed to prepare workflow**: Everything is ready.
13. **Time needed to complete experiments**: 15 minutes.
14. **Licences:** CeCILL-B licence.
15. **Workflow Frameworks:** Makefile, GNU Parallel [150].
16. **Archived(DOI):** `10.5281/zenodo.12667763`
17. **Zenodo URL:** `https://zenodo.org/doi/10.5281/zenodo.12667763`
18. **Available:** Zenodo or Docker Hub[1] [72].

Description

How Delivered

The artifact is available on Zenodo and consists of a Docker container with the scripts to reproduce figures.

Hardware dependencies

Either an Intel x86-64 or Arm AArch64 based machine. The artifact was tested using a MacBook Pro with a dual-core Intel i7 CPU, a Lenovo P720 with 2xIntel Xeon Gold 5120T CPUs (56 cores), a MacBook Air with an 8-core Apple M1 (Arm AArch64), a Cavium Thunder X2 with 2x28-core CPUs (Arm AArch64), and under x86-64 emulation (using the M1 machine).

[1] `https://hub.docker.com/r/lukeg101/oopsla24-artifact/tags`

Software Dependencies

The artifact requires a machine that supports Docker running a Linux distribution such as Ubuntu. For example on Ubuntu 20.04 you can install docker using the guide[2].

Installation

Automatic Installation (easiest)

Install docker builds for either x86-64 or AArch64:

```
> docker pull lukeg101/oopsla24-artifact:latest
```

Then run:

```
> docker run -it lukeg101/oopsla24-artifact:latest
```

Manual Installation

1. Download `oopsla24-artifact-arch.tar.gz` from Zenodo (where `arch` is `arm64` or `x86`).

2. Load the Docker container:

   ```
   > docker load -i oopsla24-artifact-arch.tar.gz
   ```

3. Run the Image:

   ```
   > docker run -it lukeg101/oopsla24-artifact
   ```

   This runs the Ubuntu image and mounts the current directory into the container at `artifact-output`.

Kick-the-tires phase

A `Makefile` drives the `herd` tool to reproduce the provided program behaviours, to run the "kick-the-tires-phase" in the container, type:

```
artifact> make figs
```

[2] `https://docs.docker.com/desktop/install/ubuntu/`

Experiment Workflow

We assume you are running the artifact with a clean directory. To evaluate a claim, run each make command below, or run them all at once using:

```
artifact> make figs
```

Updates since artifact publication

The figure numbers have changed when adapting the paper and artifact documentation for the thesis. All figures that follow in this chapter are for figures in the thesis, except artifact commands that name figures from the paper explicitly.

Paper Claims

1. Running the code in Figure 4.1(a) through `herd`, using the `rc11+lb.cat` model, produces the outcomes in Figure B.1:

```
{ P0:t=0, P1:u=1 }
{ P0:t=1, P1:u=0 }
{ P0:t=1, P1:u=1 }
```

Figure B.1: The outcomes of Figure 4.1(a) permitted by the RC11+LB model.

2. Likewise, Running Figure 4.1(d) under `aarch32.cat` produces the outcomes in Figure B.2:

```
{ P0:R0=0, P1:R0=0 }
{ P0:R0=0, P1:R0=1 }
{ P0:R0=1, P1:R0=0 }
{ P0:R0=1, P1:R0=1 }
```

Figure B.2: The outcomes of Figure 4.1(d) permitted by the AArch32 model.

3. Running Figure 4.1(b) under `aarch32.cat` produces the outcomes that match those in Figure B.3 (after mapping source locations to machine registers).

```
{ P0:R0=0, P1:R0=1 }
{ P0:R0=1, P1:R0=0 }
{ P0:R0=1, P1:R0=1 }
```

Figure B.3: The outcomes of Figure 4.1(b) permitted by the AArch32 model.

4. Running Figure 4.6(left) under `aarch64.cat` produces the same outcomes as Figure 4.6(right).

5. Running Figure 4.7(top left) under `rc11+lb.cat` produces the outcomes in Figure 4.7(top right).

6. Running Figure 4.7(bottom left) under `aarch64.cat` produces the outcomes in Figure 4.7(bottom right).

7. Running Figure 4.8(top left) under `rc11+lb.cat` produces the outcomes in Figure 4.8(top right).

8. Running Figure 4.8(bottom left) under `aarch64.cat` produces the outcomes in Figure 4.8(bottom right).

9. Running Figure 4.10(top left) under `rc11+lb.cat` produces the outcomes in Figure 4.10(top right).

10. Running Figure 4.10(bottom left) under `aarch64.cat` produces the outcomes in Figure 4.10(bottom right).

11. Compiling and Running Figure 4.9 on GCC and LLVM produces different results.

A number of minor claims appear in the paper, like how we implemented a 128-bit signed integer type in `herd`. To keep this document small we refer the reader to the figures that use these features. To validate the bug reports, please see our bug board[3].

[3] `https://lukegeeson.com/blog/2023-10-17-Telechat-Bug-Board/`

Evaluation and Expected Results

Compare the expected results and timings of each by comparing the output of the following commands with the data in the provided `log` files (all run on an Apple M1 machine) in Table B.1:

Claim	Command	Time Required	Expected Result (should match on stdout)
1	`make fig1a`	< 1 minute	See ∼`/expected/fig1a/outcomes.log`
2	`make fig1d`	< 1 minute	∼`/expected/fig1d/outcomes.log`
3	`make fig1b`	< 1 minute	∼`/expected/fig1b/outcomes.log`
4	`make fig9`	< 1 minute	∼`/expected/fig9/outcomes.log`
5	`make fig11`	< 1 minute	∼`/expected/fig11/outcomes.log`
6	`make fig17`	< 1 minute	∼`/expected/fig17/outcomes.log`
7	`make fig12`	< 1 minute	∼`/expected/fig12/outcomes.log`
8	`make fig13`	< 1 minute	∼`/expected/fig13/outcomes.log`
9	`make fig14`	< 1 minute	∼`/expected/fig14/outcomes.log`
10	`make fig15`	< 1 minute	∼`/expected/fig15/outcomes.log`
11	`make fig16`	< 1 minute	stdout of running both programs will differ

Table B.1: Commands to compare the Figures' output with log files.

Expected warnings or errors

- *Unrolling limit exceeded, legal outcomes may be missing.* This is a standard warning emitted by `herd`, for our tests this is fine.

Reusability guide

Core of the Artifact

The core of the artifact is the `herd` tool [12]. We provide the figures in the paper, and check their behaviour under models using herd. Documentation of the artifact is provided in the `README.md`.

Experiment Customisation

You can customise the experiments when invoking Make:

- Set source model (default `rc11+lb.cat`, options: `c11_partialSC.cat`, `c11_simp.cat`, `rc11.cat`, `rc11+lb.cat`):

  ```
  artifact> make figs CMEM=c11_simp.cat
  ```

- Set architecture model for figures featuring assembly tests (default `aarch32.cat`, options: `arm.cat`, `aarch64.cat`):

  ```
  artifact> make figs AMEM=aarch32.cat
  ```

- Set simulation timeout, (default `120.0` seconds):

  ```
  artifact> make figs TIMEOUT=1.0
  ```

warning: choosing an incompatible model will cause simulation to fail. This is because the model is used to pick which parser is used on the test in question, if the C Parser is used to parse an assembly test, it will fail. This is expected and a limitation of how herd works. Running each figure individually under a specific model (ie `make fig1d AMEM=arm.cat`) works in this case. If parsing succeeds (this can happen for instance with `arm.cat` and `aarch32.cat` tests which share the same parser), then the hash of the final outcomes may differ, once again causing a failure. This is also expected.

Replicating results without the artifact

It is possible to replicate the results by building `herd` from source, and running the tests in the `expected` directory. To build `herd` from source, please run:

```
> git clone https://github.com/herd/herdtools7
> cd herdtools7
> make # Note: you may need to install some dependencies
```

Once you have successfully built `herd`, you can run the tests provided in the `expected` directory:

```
> ./_build/install/default/bin/herd7 \
    -model herd/libdir/rc11.cat      \
    -I herd/libdir ~/expected/fig1a/fig1a.litmus
```

which should produce outcomes much like the artifact.

Note: the `rc11+lb.cat` model is not provided by `herd` — you will need to get it from the artifact.

Appendix C

Armv8 Atomics Application Binary Interface Specification

This appendix provides a copy of the C/C++ Atomics Application Binary Interface Standard for the Arm® 64-bit Architecture we developed with our Arm colleagues. The document was released as an official specification in the Q3 2024 release of the Arm ABI. The PDF can be found here:

`https://github.com/ARM-software/abi-aa/releases/download/2024Q3/atomicsabi64.pdf`

The document source can also be rendered on GitHub using the restructed text format: `https://github.com/ARM-software/abi-aa`

For updates on this document, please check the GitHub ABI page above. In what follows we produce a copy of this document as it was released in Q3 of 2024, except we remove the page numbers so that the thesis has one continuous sequence of numbers. Please observe the copyright for this document and the disclaimer that the views of the authors expressed in this thesis are not endorsed by Arm or any other company mentioned.

Luke developed this ABI as part of the mix testing work in Chapter 4 with assistance from Wilco Dijkstra from Arm's Compiler Teams. The ABI contains the current compiler mappings of LLVM and GCC and new or corrected mappings added as a result of the bugs we found.

C/C++ Atomics Application Binary Interface Standard for the Arm® 64-bit Architecture

2024Q3

Date of Issue: 5th September 2024

arm

1 Preamble

1.1 Abstract

This document describes the C/C++ Atomics Application Binary Interface for the Arm 64-bit architecture. This document lists the valid mappings from C/C++ Atomic Operations to sequences of AArch64 instructions. For further information on the memory model, refer to §B2 of the Arm Architecture Reference Manual [ARMARM].

1.2 Keywords

C++, C, Application Binary Interface, ABI, AArch64, C++ ABI, generic C++ ABI, Atomics, Concurrency

1.3 Latest release and defects report

Please check C/C++ Atomics Application Binary Interface Standard for the Arm 64-bit Architecture for the latest release of this document.

Please report defects in this specification to the issue tracker page on GitHub.

1.4 Acknowledgement

This ABI was written as part of Luke Geeson's PhD on testing the compilation of concurrent C/C++ with assistance from Wilco Dijkstra from Arm's Compiler Teams.

It is an offshoot from a paper that will be presented at OOPSLA 2024 [OOPSLA]: *Mix Testing: Specifying and Testing ABI Compatibility Of C/C++ Atomics Implementations* by Luke Geeson, James Brotherston, Wilco Dijkstra, Alastair Donaldson, Lee Smith, Tyler Sorensen, and John Wickerson.

1.5 Licence

1.6 About the license

1.7 Contributions

1.8 Trademark notice

1.9 Copyright

Contents

2 About this document

2.1 Change control

2.1.1 Current status and anticipated changes

The following support level definitions are used by the Arm Atomics ABI specifications:

Release

Arm considers this specification to have enough implementations, which have received sufficient testing, to verify that it is correct. The details of these criteria are dependent on the scale and complexity of the change over previous versions: small, simple changes might only require one implementation, but more complex changes require multiple independent implementations, which have been rigorously tested for cross-compatibility. Arm anticipates that future changes to this specification will be limited to typographical corrections, clarifications and compatible extensions.

Beta

Arm considers this specification to be complete, but existing implementations do not meet the requirements for confidence in its release quality. Arm may need to make incompatible changes if issues emerge from its implementation.

Alpha

The content of this specification is a draft, and Arm considers the likelihood of future incompatible changes to be significant.

All content in this document is at the **Alpha** quality level.

2.2 Change History

If there is no entry in the change history table for a release, there are no changes to the content of the document for that release.

Issue	Date	Change
00alp0	5th September 2024	Alpha Release.

2.3 References

This document refers to, or is referred to by, the following documents.

Ref	External reference or URL	Title
ARMARM	DDI 0487	Arm Architecture Reference Manual Armv8 for Armv8-A architecture profile
CSTD	ISO/IEC 9899:2018	International Standard ISO/IEC 9899:2018 – Programming languages C.
AAELF64	ELF for the Arm 64-bit Architecture (AArch64)	ELF for the Arm 64-bit Architecture (AArch64)
CPPABI64	C++ ABI for the Arm 64-bit Architecture (AArch64)	C++ ABI for the Arm 64-bit Architecture (AArch64)
RATIONALE	Rationale Document for C11 Atomics ABI	Rationale Document for C11 Atomics ABI
PAPER	CGO paper	Compiler Testing with Relaxed Memory Models

2.4 Terms and Abbreviations

The C/C++ Atomics ABI for the Arm 64-bit Architecture uses the following terms and abbreviations.

AArch64

The 64-bit general-purpose register width state of the Armv8 architecture.

ABI

Application Binary Interface:

1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the *Linux ABI for the Arm Architecture*.
2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the C++ ABI for the Arm 64-bit Architecture [CPPABI64], or ELF for the Arm Architecture [AAELF64].

Arm-based

... based on the Arm architecture ...

Thread

A unit of computation (e.g. a POSIX thread) of a process, managed by the OS.

Atomic Operation

An indivisble operation on a memory location. This can be a load, store, exchange, compare, or arithmetic operation. Atomics may be used to define higher level primitives including locks and concurrent queues. ISO C/C++ defines a range of supported atomic types and operations.

Concurrent Program

A C or C++ program that consists of one or more threads. Threads may communicate with each other through memory locations, using both Atomic Operations and standard memory accesses.

Memory Order Parameter

The order of memory accesses as executed by each thread may not be the same as the order they are written in the program. The Memory Order describes how memory accesses are ordered with respect to other memory accesses or Atomic Operations. ISO C/C++ defines a `memory_order` enum type for the set of memory orders.

Mapping

A mapping from an Atomic Operation to a sequence of AArch64 instructions.

3 Overview

AArch64 atomic mappings defines the mappings from C/C++ atomic operations to AArch64 that are interoperable.

Arbitrary registers may be used in the mappings. Instructions marked with * in the tables cannot use WZR or XZR as a destination register. This is further detailed in Special Cases.

Only some variants of fetch_<op> are listed since the mappings are identical except for a different <op>.

Atomic operations and Memory Order are abbreviated as follows:

Atomic Operation	Short form
atomic_store_explicit(...)	store(...)
atomic_load_explicit(...)	load(...)
atomic_thread_fence(...)	fence(...)
atomic_exchange_explicit(...)	exchange(...)
atomic_fetch_add_explicit(...)	fetch_add(...)
atomic_fetch_sub_explicit(...)	fetch_sub(...)
atomic_fetch_or_explicit(...)	fetch_or(...)
atomic_fetch_xor_explicit(...)	fetch_xor(...)
atomic_fetch_and_explicit(...)	fetch_and(...)

Memory Order Parameter	Short form
memory_order_relaxed	relaxed
memory_order_acquire	acquire
memory_order_release	release
memory_order_acq_rel	acq_rel
memory_order_seq_cst	seq_cst

If there are multiple mappings for an Atomic Operation, the rows of the table show the options:

Atomic Operation		AArch64
store(loc,val,relaxed)	ARCH1	option A
	ARCH2	option B

Where ARCH is either the base architecture (Armv8-A) or an extension like FEAT_LSE.

Suggestions and improvements to this specification may be submitted to the: issue tracker page on GitHub.

4 AArch64 atomic mappings

4.1 Synchronization Fences

Fence	AArch64
atomic_thread_fence(relaxed)	NOP
atomic_thread_fence(acquire)	DMB ISHLD
atomic_thread_fence(release) atomic_thread_fence(acq_rel) atomic_thread_fence(seq_cst)	DMB ISH

4.2 32-bit types

In what follows, register X1 contains the location loc and W2 contains val. W0 contains input exp in compare-exchange. The result is returned in W0.

Atomic Operation		AArch64
store(loc,val,relaxed)		STR W2, [X1]
store(loc,val,release) store(loc,val,seq_cst)		STLR W2, [X1]
load(loc,relaxed)		LDR W2, [X1]
load(loc,acquire)	Armv8-A	LDAR W2, [X1]
	FEAT_RCPC	LDAPR W2, [X1]
load(loc,seq_cst)		LDAR W2, [X1]
exchange(loc,val,relaxed)	Armv8-A	loop: LDXR W0, [X1] STXR W3, W2, [X1] CBNZ W3, loop
	FEAT_LSE	SWP W2, W0, [X1] *
exchange(loc,val,acquire)	Armv8-A	loop: LDAXR W0, [X1] STXR W3, W2, [X1] CBNZ W3, loop
	FEAT_LSE	SWPA W2, W0, [X1] *

Atomic Operation		AArch64

exchange(loc,val,release)	Armv8-A	loop:
		  LDXR W0, [X1]
		  STLXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	SWPL W2, W0, [X1] *

exchange(loc,val,acq_rel)
exchange(loc,val,seq_cst)
	Armv8-A	loop:
		  LDAXR W0, [X1]
		  STLXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	SWAL W2, W0, [X1] *

fetch_add(loc,val,relaxed)	Armv8-A	loop:
		  LDXR W0, [X1]
		  ADD W2, W2, W0
		  STXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	LDADD W0, W2, [X1] *

fetch_add(loc,val,acquire)	Armv8-A	loop:
		  LDAXR W0, [X1]
		  ADD W2, W2, W0
		  STXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	LDADDA W0, W2, [X1] *

fetch_add(loc,val,release)	Armv8-A	loop:
		  LDXR W0, [X1]
		  ADD W2, W2, W0
		  STLXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	LDADDL W0, W2, [X1] *

fetch_add(loc,val,acq_rel)
fetch_add(loc,val,seq_cst)
	Armv8-A	loop:
		  LDAXR W0, [X1]
		  ADD W2, W2, W0
		  STLXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	LDADDAL W0, W2, [X1] *

Atomic Operation		AArch64

`compare_exchange_strong(`
`loc,exp,val,relaxed,relaxed)`
	Armv8-A	`MOV W4, W0`
		`loop:`
		`LDXR W0, [X1]`
		`CMP W0, W4`
		`B.NE fail`
		`STXR W3, W2, [X1]`
		`CBNZ W3, loop`
		`fail:`

	FEAT_LSE	`CAS W0, W2, [X1] *`

`compare_exchange_strong(`
`loc,exp,val,acquire,acquire)`
	Armv8-A	`MOV W4, W0`
		`loop:`
		`LDAXR W0, [X1]`
		`CMP W0, W4`
		`B.NE fail`
		`STXR W3, W2, [X1]`
		`CBNZ W3, loop`
		`fail:`

	FEAT_LSE	`CASA W0, W2, [X1] *`

`compare_exchange_strong(`
`loc,exp,val,release,release)`
	Armv8-A	`MOV W4, W0`
		`loop:`
		`LDXR W0, [X1]`
		`CMP W0, W4`
		`B.NE fail`
		`STLXR W3, W2, [X1]`
		`CBNZ W3, loop`
		`fail:`

	FEAT_LSE	`CASL W0, W2, [X1] *`

`compare_exchange_strong(`
`loc,exp,val,acq_rel,acquire)`
`compare_exchange_strong(`
`loc,exp,val,seq_cst,seq_cst)`
	Armv8-A	`MOV W4, W0`
		`loop:`
		`LDAXR W0, [X1]`
		`CMP W0, W4`
		`B.NE fail`
		`STLXR W3, W2, [X1]`
		`CBNZ W3, loop`
		`fail:`

	FEAT_LSE	`CASAL W0, W2, [X1] *`

4.3 8-bit types

The mappings for 8-bit types are the same as 32-bit types except they use the `B` variants of instructions.

4.4 16-bit types

The mappings for 16-bit types are the same as 32-bit types except they use the `H` variants of instructions.

4.5 64-bit types

The mappings for 64-bit types are the same as 32-bit types except the registers used are X-registers.

4.6 128-bit types

Since the access width of 128-bit types is double that of the 64-bit register width, the following mappings use *pair* instructions, which require their own table.

In what follows, register `X4` contains the location `loc`, `X2` and `X3` contain the input value `val`. `X0` and `X1` contain input `exp` in compare-exchange. The result is returned in `X0` and `X1`.

Atomic Operation		AArch64

`store(loc,val,relaxed)`	`Armv8-A`	`loop:`
		`  LDXP  XZR, X1, [X4]`
		`  STXP  W5, X2, X3, [X4]`
		`  CBNZ  W5, loop`

	`FEAT_LSE`	`  LDP  X0, X1, [X4]`
		`loop:`
		`  MOV  X6, X0`
		`  MOV  X7, X1`
		`  CASP  X0, X1, X2, X3, [X4]`
		`  CMP  X0, X6`
		`  CCMP  X1, X7, 0, EQ`
		`  B.NE  loop`

	`FEAT_LSE2`	`STP  X2, X3, [X4]`

`store(loc,val,release)`	`Armv8-A`	`loop:`
		`  LDXP  XZR, X1, [X4]`
		`  STLXP  W5, X2, X3, [X4]`
		`  CBNZ  W5, loop`

	`FEAT_LSE`	`  LDP  X0, X1, [X4]`
		`loop:`
		`  MOV  X6, X0`
		`  MOV  X7, X1`
		`  CASPL  X0, X1, X2, X3, [X4]`
		`  CMP  X0, X6`
		`  CCMP  X1, X7, 0, EQ`
		`  B.NE  loop`

	`FEAT_LSE2`	`DMB  ISH` `STP  X2, X3, [X4]`
	`FEAT_LRCPC3`	`STILP  X2, X3, [X4]`

Atomic Operation		AArch64

store(loc,val,seq_cst)	Armv8-A	loop:
		  LDAXP XZR, X1, [X4]
		  STLXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_LSE	LDP X0, X1, [X4]
		loop:
		  MOV X6, X0
		  MOV X7, X1
		  CASPAL X0, X1, X2, X3, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

	FEAT_LSE2	DMB ISH STP X2, X3, [X4] DMB ISH
	FEAT_LRCPC3	STILP x2, X3, [X4]

load(loc,relaxed)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	CASP X0, X1, X0, X1, [X4]
	FEAT_LSE2	LDP X0, X1, [X4]

load(loc,acquire)	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	CASPA X0, X1, X0, X1, [X4]
	FEAT_LSE2	LDP X0, X1, [X4] DMB ISHLD
	FEAT_LRCPC3	LDIAPP X0, X1, [X4]

Atomic Operation		AArch64

load(loc,seq_cst)	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	CASPA X0, X1, X0, X1, [X4]

	FEAT_LSE2	LDAR X5, [X4]
		LDP X0, X1, [X4]
		DMB ISHLD

	FEAT_LRCPC3	LDAR X5, [X4] LDIAPP X0, X1, [X4]

exchange(loc,val,relaxed)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  STXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_LSE	MOV X6, X0
		MOV X7, X1
		loop:
		  CASP X0, X1, X2, X3, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		SWPP X0, X1, [X4]

exchange(loc,val,acquire)	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  STXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_LSE	MOV X6, X0
		MOV X7, X1
		loop:
		  CASPA X0, X1, X2, X3, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		SWPPA X0, X1, [X4]

Atomic Operation		AArch64

exchange(loc,val,release)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  STLXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_LSE	LDP X0, X1, [X4]
		loop:
		  MOV X6, X0
		  MOV X7, X1
		  CASPL X0, X1, X2, X3, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		SWPPL X0, X1, [X4]

exchange(loc,val,acq_rel)
exchange(loc,val,seq_cst)
	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  STLXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_LSE	LDP X0, X1, [X4]
		loop:
		  MOV X6, X0
		  MOV X7, X1
		  CASPAL X0, X1, X2, X3, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		SWPPAL X0, X1, [X4]

Atomic Operation		AArch64

fetch_add(loc,val,relaxed)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  ADDS X0, X0, X2
		  ADC X1, X1, X3
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	LDP X0, X1, [X4]
		loop:
		  MOV X6, X0
		  MOV X7, X1
		  ADDS X8, X0, X2
		  ADC X9, X1, X3
		  CASP X0, X1, X8, X9, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

fetch_add(loc,val,acquire)	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  ADDS X0, X0, X2
		  ADC X1, X1, X3
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	LDP X0, X1, [X4]
		loop:
		  MOV X6, X0
		  MOV X7, X1
		  ADDS X8, X0, X2
		  ADC X9, X1, X3
		  CASPA X0, X1, X8, X9, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

fetch_add(loc,val,release)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  ADDS X0, X0, X2
		  ADC X1, X1, X3
		  STLXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	LDP X0, X1, [X4]
		loop:
		  MOV X6, X0
		  MOV X7, X1
		  ADDS X8, X0, X2
		  ADC X9, X1, X3
		  CASPL X0, X1, X8, X9, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

Atomic Operation		AArch64

fetch_add(loc,val,acq_rel)
fetch_add(loc,val,seq_cst)
	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  ADDS X0, X0, X2
		  ADC X1, X1, X3
		  STLXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	LDP X0, X1, [X4]
		loop:
		  MOV X6, X0
		  MOV X7, X1
		  ADDS X8, X0, X2
		  ADC X9, X1, X3
		  CASPAL X0, X1, X8, X9, [X4]
		  CMP X0, X6
		  CCMP X1, X7, 0, EQ
		  B.NE loop

fetch_or(loc,val,relaxed)	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		LDSETP X0, X1, [X4]

fetch_or(loc,val,acquire)	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		LDSETPA X0, X1, [X4]

fetch_or(loc,val,release)	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		LDSETPL X0, X1, [X4]

fetch_or(loc,val,acq_rel)
fetch_or(loc,val,seq_cst)
	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		LDSETPAL X0, X1, [X4]

fetch_and(loc,val,relaxed)	FEAT_LSE128	MVN X0, X2 MVN X1, X3 LDCLRP X0, X1, [X4]
fetch_and(loc,val,acquire)	FEAT_LSE128	MVN X0, X2 MNV X1, X3 LDCLRPA X0, X1, [X4]
fetch_and(loc,val,release)	FEAT_LSE128	MVN X0, X2 MVN X1, X3 LDCLRPL X0, X1, [X4]
fetch_and(loc,val,acq_rel) fetch_and(loc,val,seq_cst)	FEAT_LSE128	MVN X0, X2 MVN X1, X3 LDCLRPAL X0, X1, [X4]

Atomic Operation		AArch64

compare_exchange_strong(
loc,exp,val,relaxed,relaxed)
	Armv8-A	loop:
		LDXP X6, X7, [X4]
		CMP X6, X0
		CCMP X7, X1, 0, EQ
		CSEL X8, X2, X6, EQ
		CSEL X9, X3, X7, EQ
		STXP W5, X8, X9, [X4]
		CBNZ W5, loop
		MOV X0, X6
		MOV X1, X7

	FEAT_LSE	CASP X0, X1, X2, X3, [X4]

compare_exchange_strong(
loc,exp,val,acquire,acquire)
compare_exchange_strong(
loc,exp,val,acquire,relaxed)
	Armv8-A	loop:
		LDAXP X6, X7, [X4]
		CMP X6, X0
		CCMP X7, X1, 0, EQ
		CSEL X8, X2, X6, EQ
		CSEL X9, X3, X7, EQ
		STXP W5, X8, X9, [X4]
		CBNZ W5, loop
		MOV X0, X6
		MOV X1, X7

	FEAT_LSE	CASPA X0, X1, X2, X3, [X4]

compare_exchange_strong(
loc,exp,val,release,relaxed)
	Armv8-A	loop:
		LDXP X6, X7, [X4]
		CMP X6, X0
		CCMP X7, X1, 0, EQ
		CSEL X8, X2, X6, EQ
		CSEL X9, X3, X7, EQ
		STLXP W5, X8, X9, [X4]
		CBNZ W5, loop
		MOV X0, X6
		MOV X1, X7

	FEAT_LSE	CASPL X0, X1, X2, X3, [X4]

compare_exchange_strong(
loc,exp,val,acq_rel,acquire)
compare_exchange_strong(
loc,exp,val,seq_cst,acquire)
	Armv8-A	loop:
		LDAXP X6, X7, [X4]
		CMP X6, X0
		CCMP X7, X1, 0, EQ
		CSEL X8, X2, X6, EQ
		CSEL X9, X3, X7, EQ
		STLXP W5, X8, X9, [X4]
		CBNZ W5, loop
		MOV X0, X6
		MOV X1, X7

	FEAT_LSE	CASPAL X0, X1, X2, X3, [X4]

5 Special Cases

5.1 Unused result in Read-Modify-Write atomics

`CAS`, `SWP` and `LD<OP>` instructions must not use the zero register if the result is not used since it allows reordering of the read past a `DMB ISHLD` barrier. Affected instructions are marked with `*`.

5.2 Const-Qualified 128-bit Atomic Loads

Const-qualified data containing 128-bit atomic types should not be placed in read-only memory (such as the `.rodata` section).

Before FEAT_LSE2, the only way to implement a single-copy 128-bit atomic load is by using a Read-Modify-Write sequence. The write is not visible to software if the memory is writeable. Compilers and runtimes should prefer the FEAT_LSE2/FEAT_LRCPC3 sequence when available.