Compiler Testing with Relaxed Memory Models

Luke Geeson
University College London, UK
luke.geeson@cs.ucl.ac.uk

Lee Smith
Arm Ltd, UK*
lee.d.smith@acm.org

***Abstract*—Finding bugs is key to the correctness of compilers in wide use today. If the behaviour of a compiled program, as allowed by its architecture memory model, is not a behaviour of the source program under its source model, then there is a bug. This holds for all programs, but we focus on concurrency bugs that occur only with two or more threads of execution. We focus on testing techniques that detect such bugs in C/C++ compilers.**

We seek a testing technique that automatically covers concurrency bugs up to fixed bounds on program sizes and that scales to find bugs in compiled programs with many lines of code. Otherwise, a testing technique can miss bugs. Unfortunately, the state-of-the-art techniques are yet to satisfy all of these properties.

We present the Téléchat compiler testing tool for concurrent programs. Téléchat compiles a concurrent C/C++ program and compares source and compiled program behaviours using source and architecture memory models. We make three claims: Téléchat improves the state-of-the-art at finding bugs in code generation for multi-threaded execution, it is the first public description of a compiler testing tool for concurrency that is deployed in industry, and it is the first tool that takes a significant step towards the desired properties. We provide experimental evidence suggesting Téléchat finds bugs missed by other state-of-the-art techniques, case studies indicating that Téléchat satisfies the properties, and reports of our experience deploying Téléchat in industry regression testing.

***Index Terms*—D.1.3 Concurrent Programming, B.1.2.b Formal models, B.1.4.b Languages and compilers, D.2.5.r Testing tools**

I. Introduction

Finding compiled program behaviours, or *bugs*, that are forbidden by the source program's language semantics, is key to ensuring compiler correctness. Finding concurrency bugs in compilers is especially important, as more programs are compiled for multicore processors each year. Unfortunately, finding such bugs can be tricky, as concurrent programs exhibit behaviours that can be unintuitive. To complicate matters, multi-core processors may execute each thread of a concurrent program *out-of-order*, influencing the execution of other threads through shared memory. This is *relaxed memory concurrency*, and is exhibited by processors based on the Arm architecture, Intel x86-64, IBM PowerPC, RISC-V, MIPS, and more. As such concurrency bugs can require conditions that rarely occur in practice. We address the problem of *how* to find concurrency bugs introduced by compilers when preparing programs for these architectures.

For a compiler to be deemed correct, the compiled program must behave as allowed by the semantics of its source [52]. The behaviour of a concurrent program can be defined by its set of *executions* - characterised by the communications between threads of execution through shared memory [75]. A *memory consistency model* [12], such as the ISO C/C++ [46] model M_S, describes the set of allowed executions of a C/C++ source program S. Likewise, the Armv8 model M_C in §B2.3.1 in the Arm Architecture Reference Manual [14] describes the allowed executions of a compiled program $comp(S)$. All widely used processor architectures have published memory models. A correct compiler must ensure that the allowed source program executions include the allowed compiled program executions for each well-defined concurrent program S:

$$\forall S. outcomes(exec(comp(S), M_C)) \subseteq outcomes(exec(S, M_S)) \quad \text{(eq.1)}$$

If the outcomes of source program executions do not include the outcomes of compiled executions under each respective model ($exec(P, M)$ runs P under a model M), then there is a bug. Of course, this holds for all programs and so we focus on bugs that are observable only with two or more threads.

We are motivated to test production compilers developed by Arm's engineers. Such compilers can undergo many revisions each day for which the repeated formal proof of eq.1 is infeasible [77]. Comparing unbounded executions under relaxed memory is also undecidable [16]. Instead, we conduct bounded *testing*. We assist Arm's compiler teams who wish to deploy automated compiler testing for concurrent C/C++.

```
{ *x = 0; *y = 0; } // fixed initial state

#define relaxed memory_order_relaxed
#define release memory_order_release
#define acquire memory_order_acquire

// Concurrent Program with threads
P0 (atomic_int* y,atomic_int* x) {
  atomic_store_explicit(x,1,relaxed);
  atomic_thread_fence(release);
  atomic_store_explicit(y,1,relaxed);
}
P1 (atomic_int* y,atomic_int* x) {
  atomic_exchange_explicit(y,2,release);
  atomic_thread_fence(acquire);
  int r0 = atomic_load_explicit (x,relaxed);
}

// Predicate over the final state
exists (P1:r0=0 /\ y=2)
```

Fig. 1. Litmus tests have a fixed initial state, a concurrent program and a predicate over the final state. Outcomes that satisfy the `exists` clause are forbidden by the C/C++ model [46]. When compiled the outcome is allowed by the Armv8 AArch64 [27] model. We found this bug [38] using Téléchat.

*Smith retired from Arm at the end of 2022.

We seek a technique with four properties. Firstly, a technique needs *coverage* of bugs up to fixed bounds on programs with a fixed initial state, loop unroll factor, and no recursion. A technique without coverage may miss bugs and cannot be reliably deployed in automated testing. Second, a *general* technique should support current and future models of each source language and assembly language supported by the compiler under test, else it can miss bugs as architecture specifications evolve. Thirdly, a technique should find bugs in given tests *automatically* - without further input. Of course, finding concurrency bugs can take days, which makes testing daily compiler revisions impractical. Testing should therefore *scale* to find bugs in programs with many threads and lines of code (LoC) per thread *quickly* (two minutes). Without these properties a technique can miss bugs, as we require a reliable and repeatable means of testing each compiler revision.

Unfortunately, the state-of-the-art tools [22], [66], [77] are yet to satisfy the four properties. The `C4` tool [50], [77], [78] exploits the scalability of hardware, but hardware may miss bugs [78], as bugs can occur perhaps once in thousands of runs of a compiled program, if hardware implements the required behaviour at all. `validc` [22] and `cmmtest` [66] compare all bounded executions, but require experts to find the bugs. As far as we know, these works are not deployed in industry.

We present the Téléchat compiler testing tool. Given a C/C++ program, Téléchat prepares source and compiled programs for testing, using the `herd` [12] simulator. Téléchat finds bugs when there is an outcome of executing the compiled program under the architecture model that is not an outcome of executing the source program under the source model.

We claim that Téléchat is the first tool to satisfy the four properties. By using the `herd` simulator, Téléchat finds bugs *automatically*. By relying on official models, Téléchat *covers* the behaviour allowed by authoritative C/C++ and architecture standards. Coverage is *general*, since we parameterise over both source and target models. Our technique *scales*, since Téléchat optimises compiled programs. Significant work was required to make testing scale, as `herd` is designed to test small programs, and execution time expands factorially as the test size increases, practically limiting its ability to scale much above programs of the order of 40-50 LoC. Téléchat makes significant steps towards scalable compiler testing in practice as checking compiled programs terminates in milliseconds.

Téléchat improves on the state-of-the-art for the task of finding C/C++ concurrency bugs. In other words, the set of bugs found by the state-of-the-art are a subset of bugs found by Téléchat. We contribute experimental evidence that suggests Téléchat finds behaviours missed by the state-of-the-art on the same inputs and models. As far as we know, Téléchat is the first publicly available compiler testing tool (for concurrency) to be deployed in industry.

The rest of this work is structured as follows. §II covers the background and literature review. §III covers the design, implementation, reproducible artefact, and documentation for Téléchat. We evaluate the efficacy of Téléchat in §IV and conclude in §V with lines of inquiry exposed by our work.

A. Our Contributions

Technique, Tool, and Artefact

- Novel compiler testing technique parameterised over source and architecture memory models.
- The Téléchat tool that implements our technique.
- An artefact is available to reproduce experiments using benchmark tests and documentation.

Bug-Finding Campaign

- Three new compiler bugs: Reported a run-time crash [36] induced by `const`-qualified atomic loads in LLVM for the Armv8 architecture, a wrong-endian bug [39] in the compilation of 128-bit atomic store instructions, (Armv8) a concurrency bug [37] in the compilation of 128-bit sequentially consistent [49] loads (Armv8), and an optimisation opportunity [40] in the GCC MIPS backend.
- A new model bug: Fixed a bug [35] in the unofficial Armv7 model that allowed behaviour forbidden on Arm hardware.
- One new bug type: Refute a claim made by Morisset et. al [66] and identify a new kind of bug [38] (Fig. 1).

Controlled Experiments

- Found a concurrency behaviour (Fig. 7) known to experts but missed by the state-of-the-art `C4` [77], [78].
- Found two concurrency behaviours: Conducted large-scale differential testing of LLVM and GCC for Armv8, Armv7, Intel, RISC-V, PowerPC, and MIPS architectures. With 9 million tests, it is the most extensive concurrency test campaign as far as we know.

Industry Experience

- Addressed a practical limitation of simulation. `herd` was designed to simulate *small* tests and many authors [32], [42], [66] claim it is unlikely to scale to finding bugs using large tests. We optimise compiled programs and `herd` runs much faster, often terminating in milliseconds.
- Answered queries from Arm's partners [58] concerning `LDAPR` and `LDAR` instructions.
- Deployed Téléchat in automated testing for Arm Compiler. As far as we know, Téléchat is the industry's first publicly available technique that is deployed in automated testing, and has tested Arm Compiler since June 2022.

B. Téléchat Benefits

- **Futureproof**, Téléchat tests compilers against architectural memory models, and those memory models are typically designed to describe the limits of permissible orderings of the architecture, including permissible ordering behaviours that will only be seen on future hardware or on hardware that is not readily accessible, for example Morello [13].
- **Familiar** to engineers who are not necessarily concurrency experts. Arm's engineers are using litmus tests to discuss concurrency queries as they arise.
- **Authoritative** oracle. By using official architectural models, Téléchat approaches a ground truth for compiler testing - reducing the bug-finding problem to test generation.

II. Preliminaries

We illustrate the concepts involved using the example bug report [38] in Fig 1, known as *message passing*.

A. Litmus Tests and Memory Models

Litmus tests - like in Fig. 1 - are used to explore executions allowed by hardware or a model. Litmus tests define a fixed initial state, a concurrent program, and a predicate over the final state. A concurrent program defines multiple threads (`thd=P0,P1,...`) that read from or write (*events* `E`, in the terminology of §B.2 of [14]) to shared memory locations (`loc=x,y,z,...`). When threads communicate, they produce one or more *executions* as shown in Fig. 2. The `diy` tool [11] generates litmus tests from executions, such as the execution **dabc** in Fig. 2. Isla [15], Dartagnan [42], and Memalloy [76] use SMT solvers to explore executions in a similar fashion.

Definition II.1. *Execution*: A graph where nodes are events and edges are partial order relations over events [3]. An execution is *allowed* if it is exhibited by hardware or a model, else it is *forbidden*. The base relations are:

- *program-order (po)* $\triangleq$ $\{ (E_1,E_2) \mid thd(E_1)=thd(E_2) \wedge E_1;E_2 \}$ *ie* the order instructions are written on the page
- *reads-from (rf)* $\triangleq$ $\{ (W,R) \mid loc(W)=loc(R) \wedge val(W) = val(R) \}$
- *coherence (co)* $\triangleq$ $\{ (W_1,W_2) \mid loc(W_1)=loc(W_2) \}$
- *from-read (fr)* $\triangleq$ $\{(R,W_1) \mid \exists W_2.\ W_2 \xrightarrow{rf} R \wedge W_2 \xrightarrow{co} W_1\}$

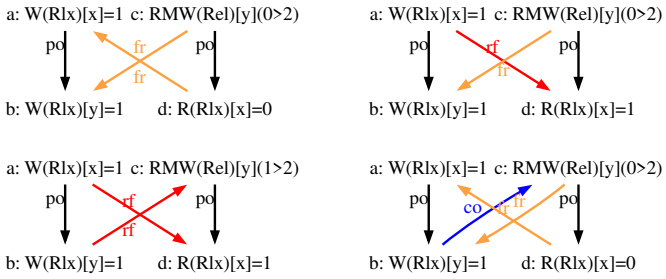

Fig. 2. Executions of Fig. 1, `a:R(Rlx)[x]=0` means event a reads the value 0 from location x with relaxed memory ordering. Top left is **acbd** or **cabd**, right **abcd**, bottom left **cdab**, right **dabc** (**dabc** is forbidden by RC11 [25]).

Executions abstract machine operations as mathematical objects. Executions model architecture features, such as pipelines or caches, as their effects on the order that reads and writes reach shared memory. For a given thread, the order that its accesses reach memory influences the executions of other threads, which influence memory in turn. When a litmus test is run from some fixed initial state, the interaction between all threads of execution produces a set of *candidate* executions.

A *memory consistency model* (or a model) filters out forbidden executions of a litmus test. Models define predicates on relations over events, forbidding - for instance - cyclic executions. The Cat [2] language specifies models of RC11 [25], Armv8 AArch64 [27] (official), Armv7 [8] (unofficial), RISC-V [61] (official), Linux [9], Intel x86-64 [65], IBM PowerPC [63], MIPS [64], and more. The `herd` *simulator* [12] enumerates the executions of small litmus tests (from fixed initial states up to fixed loop unroll factor with no recursion) allowed by a Cat model. Fig. 3 shows the *outcomes* of the executions of Fig. 1 allowed by the RC11 C/C++ model [25].

Definition II.2. *Outcome*: An outcome is the result of an execution (def. II.1) expressed as a set of assignments to shared memory (e.g. `y=2`) and thread-local data (e.g. `P1:r0=1`). The set of outcomes of executing a litmus test P is denoted $outcomes_P$. We defer other effects (like IO) to future work.

```
{ P1:r0=0; [y]=1; } // outcome acdb of Fig.1
{ P1:r0=1; [y]=1; } // abcd
{ P1:r0=1; [y]=2; } // cdab
```

Fig. 3. The outcomes of executions in Fig. 2. The `acyclic` constraint of the RC11 model [25] forbids **dabc** and its outcome {`P1:r0=0; y=2`}.

The Arm AArch64 and RISC-V models are maintained by their respective architecture specification teams. Other models used are from peer-reviewed publications. We build on these models and rely on their correctness.

The `litmus` tool [10] runs litmus tests on hardware to check if hardware correctly implements models. If hardware exhibits forbidden executions then either the model is wrong, or the hardware is incorrectly implemented. As silicon manufacturers may implement restricted variants of an architecture model, hardware executions may omit behaviours allowed by the model. The `litmus` tool is therefore of limited use to compiler testing. To be clear, it *is* necessary to validate hardware against models but that is a *separate* problem from validating *compilers* against models.

Tools that use hardware as an oracle for correct behaviour are unlikely to be reliable for compiler testing. Observing behaviours on hardware may require circumstances that rarely occur in practice, if at all. The chances of observing a behaviour depend on whether a given implementation supports it and whether the hardware is in a sufficiently stressed state. Observations may require sampling a vast array of hardware many (thousands of) times to reliably test a compiler revision. Even then, testing on hardware may miss bugs.

B. Compiler Testing

We use the testing terminology from Barr et al. [17]. To test is to stimulate a system under test and observe its response [17]. We stimulate the system under test *comp*, with a source program S and observe a response [31]:

- Internal Compiler Error: *comp* may crash during compilation because of a problem in *comp* or S; for example a segmentation fault in GCC.
- Functional Error: $comp(S)$ produces a compiled program C that exhibits behaviour B that differs from expected behaviour B' when C is run in a test environment $exec$; for example a run-time crash or concurrency bug.

We focus on concurrency bugs. A concurrency bug occurs when there are outcomes of executions of $comp(S)$ - run in test environment $exec$ - that are forbidden by S.

TABLE I
COMPARISON OF STATE-OF-THE-ART COMPILER TESTING TECHNIQUES - INSPIRED BY TABLE 1 OF [20]

Technique	Automation	Coverage	General	Scalability	*exec*	Comparison
Prose/Experts	✗	?	✓	✗	Human	Any
`cmmtest` [66]	?	✗	✗	✗	Human+hardware	executions (def. II.1)
`validc` [22]	?	✓	✗	✗	Human+models	executions
`C4` [50], [77], [78]	?	✗	?	✓	models+hardware	outcomes (def. II.2)
Téléchat	✓	✓	✓	✓	models only	outcomes

Definition II.3. *Concurrency Bug*: for a multi-threaded S,

$$outcomes_C(exec(C)) \not\subseteq outcomes_S(exec(S))$$

We test programs that exhibit bugs with at least two threads communicating via shared memory. Conversely, a *negative difference* occurs when $outcomes_C \subset outcomes_S$ when optimisations are applied. The state-of-the-art make $exec$ precise (§II-C). Like Leroy [52] we focus on deterministic programs (whose behaviour changes only in response to different initial states) and test environments (immune to changes in $exec$). We restrict bugs to executions that have different *outcomes*. For instance, in Fig. 2, the execution **dabc** - and its outcome {`P1:r0=0; y=2`} - is forbidden by the RC11 model [25] and ISO C/C++ model [46], but the compiled program allows it under the Armv8 model [27].

The choice of source model decides what is a bug. We use the RC11 [48] model to explore the behaviours of Fig. 1, but emphasize that ISO C/C++ standard permits behaviours forbidden by RC11. Conversely, the Linux model [9] permits behaviours that are forbidden by standard C/C++ [43]. The source model thus acts as an oracle with respect to the allowed behaviours of the system under test. Since standards (and their models) can change - it is especially important to parameterise testing under multiple models. We support testing under models of source and compiled languages supported in mainstream C/C++ compilers.

Chen et al.'s [24] survey identifies two compiler testing techniques explored by the state-of-the-art: *differential* and *metamorphic* testing. Fig. 4 illustrates both techniques. Differential testing (see CSmith [79]) compiles a program S with different compilers, comparing the behaviours of each. For instance, comparing the outcomes of running executables produced by `clang -O1` and `clang -O3`. Metamorphic testing (see Orion [51]) 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`.

We end this section mentioning related work that is out of scope. Donaldson et al. 2017 [30] and Lidbury et al. 2015 [53] test the compilation of GPU/OpenCL kernels, and graphics shaders. Neither test the compilation of concurrent programs [22] in the C/C++ sense, as multi-threaded GPUs support a parallel computation model.

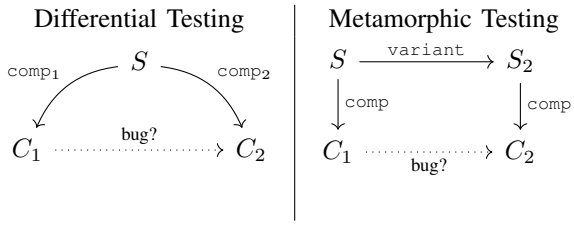

Fig. 4. Differential and Metamorphic Testing - §II-B.

C. State-of-the-art Techniques

We summarise the state-of-the-art in compiler testing with memory models. We compare works in terms of *Automation*, *Coverage*, *Generality*, and *Scalability* (see §I). A tool is *automatic* if it can be used by compiler engineers in regression testing with no intervention by concurrency experts to generate tests or interpret results. *Coverage* describes the set of potential bugs the tool will discover. *Scale* bounds the number of threads and LoCs of inputs. A tool is *general* if it supports multiple source and compiled languages. In Table I we state whether the solution fulfils the requirement with a ✓, does not ✗, and partially fulfils the requirements with ?.

1) Prose and Expertise: The first (non-)solutions involve reading prose language standards such as ISO C/C++ [46], or consulting memory model experts. Both approaches are manual and are effective in finding bugs. Both are prohibitively tedious and expensive for use in routine regression testing.

2) Semi-automatic Tools: The `cmmtest` tool [66] conducts differential testing by extending CSmith with concurrency support for an early C/C++ model. Given a *single-threaded* C/C++ program, `cmmtest` checks if the hardware execution of the optimised program is a sub-graph of an unoptimised hardware execution, else a bug may occur and a concurrency expert finds a test case reproducer. `validc` [22] builds on `cmmtest` [66] by matching *all* bounded executions of optimised LLVM IR programs against unoptimised IR.

Both techniques are manual as experts must reproduce bugs using the warnings output by the tools. Since execution matching is an instance of the sub-graph isomorphism problem [26] it will not scale in general [67]. Neither technique is general, as `cmmtest` relies on x86-only [45] tools, and `validc` accepts only LLVM IR programs. The `validc` tool covers bugs allowed by the C/C++ or LLVM models; however `cmmtest` may miss bugs as it relies on hardware.

3) Hardware-based Tools: The `C4` tool of Windsor et al. 2021/22 [50], [77], [78] conducts metamorphic testing of litmus tests by comparing the outcomes of hardware runs (using the `litmus` tool) against outcomes of source test simulations

(using the herd tool) under the RC11 model [25]. C4 is automatic and testing scales as hardware often runs quickly. Hardware test environments are nondeterministic and may omit behaviour - and hence bugs - allowed by an architecture model (§II-A). To improve coverage, Windsor et al. "stress-test" [77] hardware. Since C4 was developed in parallel with our work, we summarise it:

$$outcomes(\texttt{litmus}(comp(S), hardware)) \subseteq outcomes(\texttt{herd}(S, M_S)) \quad (\textbf{test}_{C4})$$

4) Our Solution - Testing with Models: The herd tool [12] simulates *both* source and compiled litmus tests under source M_S and architecture models M_C automatically. It follows that we can test compilers by comparing the outcomes (see Fig. 3) of executions of compiled programs under M_C against outcomes executions of a source program under M_S.

III. Design and Implementation of Téléchat

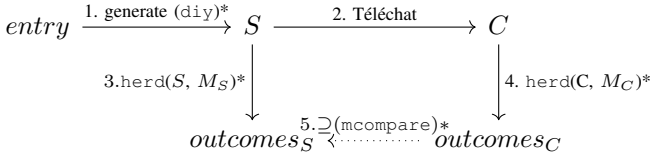

Fig. 5. Test environment $exec_{tv}$ including the Téléchat tool and tools we improved marked *. $\textbf{test}_{tv}$ checks the outcomes of simulating $comp(S)$ under its architecture model M_C against the source S under source model M_S.

A. Technique Design

We present the Téléchat automatic testing tool and technique $\textbf{test}_{tv}$; for compilers including GCC and LLVM. Fig. 5 details the test environment $exec_{tv}$ summarised by:

$$outcomes(\texttt{herd}(comp(S), M_C)) \subseteq outcomes(\texttt{herd}(S, M_S)) \quad (\textbf{test}_{tv})$$

The test environment of Fig. 5 ($exec_{tv}$) proceeds as follows:

1) Generate concurrent C/C++ litmus test S.
2) Téléchat prepares S for compilation, compiles it using $comp$ and disassembles the relocatable ELF file, then constructs an assembly litmus test C and state mappings m from outcomes of S to outcomes of C.
3) Simulate S using herd under one of the C/C++ memory models in the herd tool-suite [5] (see §II-A), collect allowed C/C++ outcomes $outcomes_S$.
4) Simulate C using herd under its architecture memory model in the herd tool-suite [5] (see §II-A). Get architecturally allowed outcomes $outcomes_C$.
5) Use mcompare from the herd tool-suite [5] to check if $outcomes_C \subseteq outcomes_S$ using state mappings m. If $outcomes_C \not\subseteq outcomes_S$ then there is a bug.

Téléchat enables the automatic testing of program outcomes by completing the graph in Fig. 5. To support compiled tests $comp(S)$, we extend the diy [11] test generator, the herd [12] simulator, and mcompare [5].

The $\textbf{test}_{tv}$ technique is remarkably simple. By checking that the *expected* outcomes of a source test under the source memory model include the *actual* outcomes of the compiled test under an architecture model we get a technique that is familiar to engineers who are not necessarily experts in relaxed memory concurrency. $\textbf{test}_{tv}$ is simple enough that Téléchat is cited in discussions by Arm's engineers [58].

$exec_{tv}$ is a deterministic (§II-B) test environment. In other words, we compare outcomes under source and architecture models - rather than relying on hardware or the operating system. Further, herd runs deterministic litmus tests: from a fixed initial state with a fixed loop unroll factor, under models of Armv8 AArch64 [27] (official), RISC-V [61] (official), RC11 [25], Armv7 (unofficial) [8], Intel x86-64 [65], MIPS [64], IBM PowerPC [63], and more.

The Téléchat tool-chain is run as part of regular automated compiler testing and is, as far as we know, the industry's first publicly available compiler testing tool that is deployed in automated compiler testing of concurrent C/C++. Nothing prevents deployment of Téléchat outside of Arm however.

B. Tool Implementation

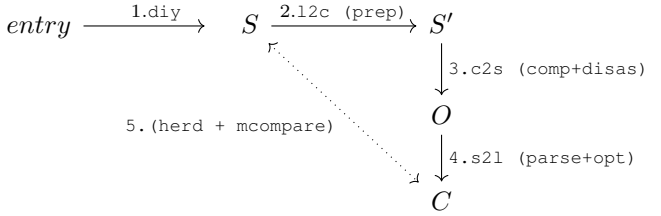

Fig. 6. Breakdown of the Téléchat tool. The l2c, c2s, and s2l tools are new. We modified diy [11], herd [12], and mcompare [5] to to accept compiler-generated tests.

Fig. 6 breaks down Step 2 (Téléchat) of Fig. 5 as follows:

1) *Input*: Generate a C/C++ litmus test S (step 1 of Fig. 5).
2) The *litmus2c* (l2c) tool prepares S for compilation, producing a C/C++ program S'. Optionally fuzz S'.
3) The *c2assembly* (c2s) tool compiles S' with ELF relocations (requires flags -c -g) and disassembles the object file using GNU or LLVM objdump. c2s returns an assembly file O and state mappings m.
4) The *assembly2litmus* (s2l) tool parses O and constructs an optimised assembly litmus test C.
5) *Output*: Pass S and C to herd for simulation under source and architecture models (steps 3-4 of Fig. 5).

C. Adoption, Usage, and Documentation

We provide a Docker artefact to reproduce our work. The artefact contains builds of LLVM and GCC for the architectures above, the Téléchat tool, and a Makefile to reproduce the experiments. A variant of the Docker artefact is deployed in automated testing for Arm's compiler teams.

For users we provide guides and example test suites. Téléchat ships a user guide with its artefact. The Docker file contains example tests suites to build on.

To reduce the risk of tool stagnation, we provide Arm's compiler team with documentation including internal conference talks on Téléchat, memory model training, and wiki pages. Arm's compiler engineers are increasingly using litmus tests to discuss concurrency queries as they arise [59].

D. Challenges Encountered During Implementation

We end by discussing the challenges encountered whilst implementing Téléchat. The first challenge arose in how to represent compiled programs as litmus tests. Compiled programs represent memory locations as binary addresses `0xf00` that can be manipulated with arithmetic instructions. ELF files layout multiple locations together in sections. Litmus tests represent memory locations as symbolic variables *x* that have no memory layout constraints. We use DWARF metadata to map numeric addresses to symbolic locations and symbol table information to gather memory layout constraints. As such our technique is as accurate as the metadata compilers provide. By converting between address formats, Téléchat bridges a gap between formal modelling tools and real-world systems.

Developing Téléchat spawned many extensions to the herd tool-suite [5], including formalising the semantics of new instructions, new data types, and tools. For instance we added a vector datatype to model memory layout and store pair instructions that span contiguous locations. We also added a regression suite for the herd tool-suite itself. Our work was used by students in projects modelling the NEON [6] and SVE [7] extensions of the Arm architecture. To compare outcomes of tests of differing architectures or languages, we added state mapping support to `mcompare`.

IV. Evaluation

We evaluated Téléchat by conducting experiments using multiple compilers. Our results suggest Téléchat improves on the state-of-the-art (we found one existing behaviour missed by `C4`, and four new bugs [36]–[39]), proposes new lines of inquiry into test generation (a new kind of bug), questions interactions between sequential and concurrency semantics (`const` and `atomic`), and makes an impact in industry.

A. Comparison with The State-Of-The-Art: C4

Téléchat is similar to `C4` [50], [77], [78], but Téléchat relies only on simulation. Both tools rely on the `herd` tool-suite [5]. Téléchat differs in that it relies *solely* on simulation of both source and compiled litmus tests whereas `C4` relies on hardware executions to collect compiled test outcomes. Tab. II summarises the differences between `C4` and Téléchat. This small change in technique has consequences for whether behaviours are observable in the test environment.

We compare tools directly as both take litmus tests and compare outcomes. We pass 85 litmus tests to both tools and compare the outcomes (def. II.2) of each. Téléchat finds behaviours that Windsor et. al miss [78]. Consider Fig. 7 and its outcomes when simulated under the RC11 model [25] - Fig. 8 (left). We compile Fig. 7 using Téléchat and LLVM-11 (`clang -O3`) to get an Arm AArch64 litmus test. Fig. 8 (right) shows the outcomes of simulating the assembly litmus test under the Arm AArch64 model [27]. We observe the outcome { `P0:r0=1; P1:r0 = 1`} that is forbidden by the RC11 model (Fig. 8 (left)), but allowed by the AArch64 model (Fig. 8 (right)). Windsor et. al state that `C4` missed this behaviour, but they observe it under model simulation, which increases our confidence in Téléchat.

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

#define relaxed memory_order_relaxed
#define load atomic_load_explicit
#define store atomic_store_explicit

void P0(atomic_int* y,atomic_int* x) {
  int r0 = load(x,relaxed);
  atomic_thread_fence(relaxed);
  store(y,1,relaxed);
}
void P1(atomic_int* y,atomic_int* x) {
  int r0 = load(y,relaxed);
  atomic_thread_fence(relaxed);
  store(x,1,relaxed);
}

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

Fig. 7. The outcome { `P0:r0=1; P1:r0 = 1`} is forbidden under the proposed RC11 [48] model. When compiled the outcome is allowed by Armv8 AArch64 [27], Armv7 [8], PowerPC [63], and RISC-V [61] models.

RC11 [48] Outcomes	Arm AArch64 Outcomes
`{P0:r0=0; P1:r0=0;}`	`{P0:r0=0;P1:r0=0;}`
`{P0:r0=0; P1:r0=1;}`	`{P0:r0=0;P1:r0=1;}`
`{P0:r0=1; P1:r0=0;}`	`{P0:r0=1;P1:r0=0;}`
.	`{P0:r0=1;P1:r0=1;}<-C4 missed`

Fig. 8. (left) Fig. 7 outcomes allowed by the RC11 model [25]. (right) Outcomes of Téléchat-generated test allowed by the Arm AArch64 model [27].

We found hundreds of litmus tests that induce this behaviour under RC11 [25] when compiled by either LLVM or GCC, detailed in §IV-D. Fig. 7 implements the *load buffering* (LB) pattern, known by concurrency experts. Further, we observe the same behaviour when compiling to target Armv7 (unofficial), IBM PowerPC, and RISC-V (official).

We conclude that Téléchat is deterministic unlike `C4`. In other words, Téléchat observes the same test outcomes every time. `C4` requires that the hardware exhibits an outcome and that users 'stress-test' [77] the hardware to reproduce it. Silicon manufacturers may however implement restricted variants of an architecture model (§II-A). `C4` is not guaranteed to observe the same outcomes on different machines, or even the same machine. Indeed, Sarkar et. al [71] observe LB on an Apple A9 and Nvidia Tegra2 chips[1], but Windsor et. al miss it [78] using a Raspberry Pi. It is possible that developing `C4`'s metamorphic relations may increase the chance of finding bugs, provided the hardware provides a witness to miscompilation.

Téléchat is useful when hardware is inaccessible. For instance, we assisted Arm's engineers with a query from an Arm partner, who proposed to change the compilation of C/C++

[1]https://www.cl.cam.ac.uk/~pes20/arm-supplemental/arm001.html#toc5

TABLE II
C4 VERSUS TÉLÉCHAT.

Component	C4 [50], [77], [78]	Téléchat
Test Generator - §II-A	Memalloy [76]	diy [11]
Test Environment - §II-B	models+hardware	models only
Source *exec* - §II-B	herd [12]	herd
Target *exec*	litmus [10]	herd
Testing method - § II-B	Metamorphic Testing [51]	Metamorphic & Differential Testing [79]
Models involved	source	source and architecture
System under test (SUT)	Compiler + Hardware + OS	Compiler
Found Bugs?	Yes	Yes [36]–[39]
Automatic	No (must stress SUT)	Yes
Coverage	No	Yes (up to fixed bounds)
General	No	Yes (parameterised over models)
Scalable	Yes	Yes
Deterministic	No	Yes

atomic acquire loads when targeting Armv8.3-a [58]. The proposed change had promising performance characteristics on unspecified hardware, but correctness was untested beyond interpreting the Armv8.3-a specification. Arm's compiler teams accepted the proposal based on our findings.

Téléchat does not completely subsume the state-of-the-art as, for example, simulation does not terminate when checking huge systems (with thousands of LoC). C4 can do this. We expect the success of Téléchat depends on validity of the *small-scope hypothesis*, which we explore in §IV-E.

B. The Local Variable Problem

The local variable problem affects all state-of-the-art techniques (§II-C) and masks a kind of bug that has evaded detection. We discovered that, contrary to the state-of-the-art, there are optimisations affecting only thread-local state that influence concurrent program execution, giving rise to a new class of bug. The problem is that there are transformations allowed by the C/C++ model [22] that delete data required to detect bugs. We reported a new bug of this kind [38] (see Fig. 1), reproduced two bugs for Arm's engineers, and present our solution in Téléchat.

Consider the load buffering (LB) litmus test in Fig. 9. When simulated using herd [12] the values of the local variables P0:r0 and P1:r0 are recorded for use when checking outcomes. C/C++ memory models [25] allow the compiler to delete unused local data. Consequently, a litmus test that refers to *deletable* data [22] in its final state - like P0:r0 and P1:r0 - will have no data to refer to if the compiler removes it. When compiling Fig. 9 (left) with clang -O2 we get Fig. 9 (right - in C for illustration purposes). The only allowed outcome of Fig. 9 (right) is { P0:r0=0; P1:r0=0 } since herd assumes data is zero-initialised.

Local reads of shared data are snapshots of that data at particular points in a program's execution. Several concurrency patterns rely on local data to convey whether the reordering of accesses leads to forbidden outcomes. For example, LB demonstrates a notion of locality - for instance caches - that *buffer* loads during execution. Local data reordering is common in many processors - we cannot ignore it, even if the C/C++ model permits its removal.

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

void P0 (int* y,int* x) {
  int r0 = *x; // unused
  *y = 1;
}
void P1 (int* y,int* x) {
  int r0 = *y; // unused
  *x = 1;
}
exists
  (P0:r0=1 /\ P1:r0=1)
```

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

void P0 (int* y,int* x) {
  // deleted
  *y = 1;
}
void P1 (int* y,int* x) {
  // deleted
  *x = 1;
}
exists
  (P0:r0=1 /\ P1:r0=1)
```

Fig. 9. (left) Load Buffering (LB) litmus test. (right) Load Buffering test after clang -O2 deletes unused data.

Testing techniques may miss bugs in optimisations that delete local data. Such techniques cannot test the compilation of LB unless local data persists. It is possible that an assembly language register of the compiled program contains local data, but compilers often reuse registers to reduce spilling. The state-of-the-art (§II-C) relies *only* on source (e.g C/C++) models - without testing if local deletion masks bugs.

Authors either overlook the issue [22], [77] or claim [66] local optimisations cannot induce bugs (def. II.3). When asked about local optimisations at the European LLVM conference (2017)[2], Chakraborty and Vafeiadis [22] state they focus on "only the shared memory accesses". Windsor et. al do not address the issue [77]. Morisset et al. [66] claim that "optimisations affecting only the thread-local state cannot induce concurrency compiler bugs". We question this [66] claim.

Fig. 10 induces a bug when thread-local optimisations delete P1:r1. The outcome {P1:r0=0; y=2} is forbidden by the C/C++ model, but allowed by the LLVM or GCC compilation to Arm AArch64 when the assignment to P1:r1 by the read-modify-write (RMW) operation is deleted. Past versions of LLVM and GCC induce this bug when targeting Armv8.1-a with the Large-Systems Extension. This example shows that thread-local optimisations can induce concurrency bugs.

P1 uses an atomic fetch_add_explicit RMW operation where the value read into P1:r1 is unused. This induces *two* bugs using past versions of LLVM and GCC,

[2]33:50 minutes in: https://www.youtube.com/watch?v=NR5OAhgdozc

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

#define relaxed memory_order_relaxed

void P0 (atomic_int* y,atomic_int* x) {
  atomic_store_explicit(x,1,relaxed);
  atomic_thread_fence(memory_order_release);
  atomic_store_explicit(y,1,relaxed);
}
void P1 (atomic_int* y,atomic_int* x) {
  int r1 = atomic_fetch_add_explicit(y,1,relaxed);
  atomic_thread_fence(memory_order_acquire);
  int r0 = atomic_load_explicit(x,relaxed);
}
exists (P1:r0=0 /\ y=2)
```

Fig. 10. Message Passing litmus test. The outcome {`P1:r0=0; y=2`} is forbidden under the C/C++ model [25], but allowed by the Arm AArch64 model [27], since an LLVM thread-local optimisations can remove `P1:r1`. The heisenbug arises if `P1` does not observe the read of the RMW operation.

first by targeting the incorrect Arm instruction and second by deleting `P1:r1`. In both cases the outcome {`P1:r0=0; y=2`} is allowed by the LLVM or GCC compilation to Arm AArch64, but forbidden by the C/C++ model. Engineers fixed the first bug replacing the `STADD` instruction with `LDADD`. The second bug is observed when the LLVM dead register definitions pass [54] zeroes the destination-register of `LDADD`, after `P1:r1` is deleted - the bug is observed as `LDADD` aliases `STADD` when the destination register is the zero register.

Finding these bugs without Téléchat required expertise and two engineer years of work. Discussions involved Linux Kernel maintainers, Arm AArch64 model authors, and GCC (and LLVM) developers [33], [34], [55], [56]. We assisted Arm's compiler teams by reproducing the bugs and showing that the latest versions of LLVM and GCC no longer exhibit them. We added support for Fig. 10 (with and without `int r1 = ...`) to `herd` [35], allowing us to validate the fix. We reported [38] a new bug of this kind in the implementation of `atomic_exchange` featured in Fig. 1, but it is unclear if more bugs like this exist.

Interestingly, these bugs disappear if one attempts to study them. Historically, message passing tests check the reordering of `P1:r0` and `P1:r1` - forcing the user to preserve local data. If instead we delete `P1:r1` and check the value of `y` in the final state, then we see reordering. In other words, you only find the bug through *indirect* observation - it is a new kind of Heisenbug! Since current test generators implement the historical case, it is no surprise that these bugs were discovered manually until now.

We implement a solution in Téléchat. Téléchat augments Fig. 9 with global variables that store local data at the end of each thread. 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 found four bugs thus far [36]–[39]. We are open to better solutions, if they exist.

C. Bug-Finding Campaign

Whilst developing Téléchat we reported two new concurrency bugs [37], [39] in LLVM, and a missed optimisation [40] for the MIPS backend of GCC. We propose two bug fixes that Arm's engineers are addressing, and note one line of inquiry for compiling atomics in practice - all untested until now.

First, we reported a bug [37] in the compilation of 128-bit sequentially consistent [49] loads. The bug occurs when a sequentially consistent [49] atomic load is implemented using a load pair instruction on Armv8.4. The Armv8.4 Large Systems Extension (v2) ensures load or store pair instructions are single-copy atomic [14], assuming accesses are 16-byte aligned to normal memory. This means you can use an `LDP` instruction in place of a potentially more expensive *compare-and-swap* (CAS) loop. `LDP` has no ordering requirements however - `LDP` can be reordered before a prior store of an atomic read-modify-write operation that uses a CAS loop. We propose to fix sequential consistency in LLVM by adding synchronisation, following GCC [28].

Next, we reported a wrong-endian bug [39] 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 the bits to fix the bug.

We reported [40] an optimisation opportunity in the MIPS backend of GCC. Whilst developing Téléchat, 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 (def. II.2) under models. Whether it is *still* valid to treat `atomic` as `volatile` is further work. The above bugs are in dark corners, difficult for experts to find even when an excellent memory model exists. Without Téléchat-style automation these faults are impossibly expensive to test for in routine regression testing.

D. Large-Scale Differential Testing

We use Téléchat to conduct differential testing of LLVM and GCC. We check compatibility between compilers, as code generated by LLVM and GCC is often mixed together at link-time or by operating systems, potentially inducing latent bugs at runtime. We test compilation targeting multiple architectures using commonly used flags for a large suite of tests. We ran over 9 million tests that have 2 to 5 threads, up to 5 shared variables, and up to 50 lines of compiled assembly code. On each thread Téléchat removes around 4 lines of (compiled) code per access. As far as we know this is the most extensive concurrency testing campaign to date. We test:

- Compilers: LLVM and GCC compiling C/C++ to target Armv8 AArch64 (64-bit), Armv7-a (32-bit), RISC-V, Intel x86-64, IBM PowerPC, and MIPS.

TABLE III
WE TEST COMBINATIONS OF C/C++ constructs × Compiler Under Test × Flags × Arch.

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 IV
TEST RESULTS - TAKES 9 HOURS AND 40 MINUTES ON A 224 CORE THUNDERX2 USING 100GB RUNTIME FOOTPRINT. CLANG DOES NOT SUPPORT `-Og` FLAG. 167,184 C TESTS INPUT, 9,027,936 COMPILED TESTS OUTPUT, TOTAL: 9,195,120. TOTAL % = SUM(ROW)/COMPILED TESTS OUTPUT * 100 (3 SF). THESE RESULTS WERE COLLECTED USING THE RC11 MODEL [48], ALL POSITIVE DIFFERENCES DISAPPEAR IF LOAD BUFFERING IS PERMITTED.

	-O1	-O2	-O3	-Ofast	-Og	Total %
Armv8 AArch64 (64-bit)			clang/gcc			
+ve	2352/2352	2352/2352	2352/2352	2353/2352	-/2352	0.23%
-ve	44300/44300	44300/44300	44300/44300	44300/44300	-/44300	4.42%
Armv7-a (32-bit)			clang/gcc			
+ve	2352/3480	2352/2352	2352/2352	2352/2352	-/2352	0.25%
-ve	68228/69890	68228/70220	68228/70220	68228/70220	-/70220	6.91%
RISC-V (64-bit)			clang/gcc			
+ve	2352/2352	2352/2352	2352/2352	2352/2352	-/2352	0.23%
-ve	34204/70772	34204/70772	34204/70772	34204/70772	-/70772	5.44%
IBM PowerPC (64-bit)			clang/gcc			
+ve	2352/2352	2352/2352	2352/2352	2352/2352	-/2352	0.23%
-ve	43956/43956	43956/43956	43956/43956	43956/43956	-/43956	4.38%
Intel x86-64 (64-bit)			clang/gcc			
+ve	0/0	0/0	0/0	0/0	-/0	0.0%
-ve	64112/64112	64112/64112	64112/64112	64112/64112	-/64112	6.39%
MIPS (64-bit)			clang/gcc			
+ve	0/0	0/0	0/0	0/0	-/0	0.0%
-ve	69664/72488	69664/72008	69664/72008	69664/72008	-/72488	7.09%

- Optimisation levels for each compiler: `-O1`, `-O2`, `-O3`, `-Ofast`, and for GCC `-Og`.
- Compare a compiler with itself at increasing levels of optimisation, e.g. `clang -O1` vs. `clang -O2`.
- Compare LLVM with GCC at each optimisation level, e.g `clang -O1` vs. `gcc -O1`.

Tab. III defines all the combinations of test, compiler, and architecture under test. Our tests feature code that perturbs the order accesses hit memory including control-flow, atomic operations, non-atomic operations, fences, and straight-line code. We test using both signed and unsigned integers ranging from 8-bits up to 64-bits in size. We test both LLVM and GCC with the optimisations and architectures above.

Following the steps in §III-A, we generate multiple source C/C++ *test sets* enumerating the features in Tab. III using `diy` [11]. For each test set, we use Téléchat with the compiler under test to generate *multiple* assembly test sets according to multiple *compiler profiles*. Each profile captures the compiler tool-chain (& flags), architecture (& model), disassembler (& flags), and symbol table reader. For instance, the `llvm-O3-AArch64` profile tests: `clang -O3` using the AArch64 GNU/Linux bare-metal tool-chain and `gnu-objdump`. Both source and target tests are passed to `herd` for simulation under the RC11 [25] and Arm AArch64 model [27] respectively (resp. target architecture models). Lastly, `mcompare` compares outcomes to find outcomes (def. II.2) of the compiled program $outcomes_C$ that are not outcomes of the source program $outcomes_S$:

- *positive differences (+ve)*: $outcomes_C \not\subseteq outcomes_S$
- *negative differences (-ve)*: $outcomes_C \subset outcomes_S$.

(negative differences can occur since both optimisations and architecture models can constrain behaviour).

Tab. IV details our results. Tab. IV suggests Téléchat is effective as it found tricky concurrency behaviours hidden in over 9 million compiled tests, given 167,184 tests as input. The 2352 positive differences common to Armv8 (official), Armv7 (unofficial), RISC-V (official), and IBM PowerPC are due to 294 variants of the load buffering pattern in Fig 7. When comparing `llvm-O1-ARM` +ve (2352) and `gcc-O1-ARM` +ve (3480) we discovered two behaviours in GCC and LLVM. Reordering is observed using `-O1` when a control dependency is removed, but the behaviour is masked at higher optimisation levels by a data dependency (`-O2` and above). Since Intel x86-64 implements the total-store order model [72] there are are no differences. This suggests Téléchat is an effective compiler testing tool.

To be clear, these positive differences are not *bugs* in today's compilers, since we used the RC11 model [48] 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 [46]), whereas RC11 forbids it. Téléchat is parameterised over models, and we repeat testing using a modified `rc11+lb.cat` model to show that all of the above behaviours disappear when load-to-store ordering is permitted.

Many differences in Tab. IV arise from data races. The C/C++ model flags data races as undefined behaviour, and we ignore false positives on that basis. Of course, we assume the models are correct, which is a limitation we accept given these promising results.

E. Limitations of Model-based Testing

Our technique has three limitations: model correctness, model completeness, and simulation scalability. When exploring each case we found new bugs detailed below.

We assume the source and target models are correct. We found a bug in the (unofficial) Armv7 model [8] that the state-of-the-art techniques miss. We found a bug when compiling for the Armv7-a architecture using a *Store Buffering* litmus test. The outcome of the test was allowed by the unofficial Armv7 model [8], but is forbidden by the RC11 model [25] and the Armv7 hardware we checked. The problem was that the Armv7 model was allowing accesses to be reordered when it should have been forbidden. We reported the bug and fixed the model [35]. As the state-of-the-art (§II-C) depends only on source models this limitation is unique to Téléchat.

Next, we assume that models support language features under test. We reported [36] a bug in the implementation of 128-bit `const` atomic loads. We found that `const` was miscompiled when loading constant atomic data - it crashes at run-time as the C/C++ load is implemented using a store instruction that attempts to write to read-only memory. Simulation under the Arm AArch64 model [27] will miss this bug, as `const` read-only memory is unsupported, and so we augment the model to flag `const` violations. Whilst conducting this study a fix was proposed in LLVM by engineers [57], but the problem remains as the fix only applies to Armv8.4 or above (similar code exists for Intel x86-64, RISC-V, IBM PowerPC backends). Upon discussing this bug with Arm's compiler engineers, we conclude there is no (lock-free) fix for Armv8.0, since the 128-bit load instruction is not guaranteed to be single-copy atomic unless the Armv8.1 Large Systems Extension is implemented.

Lastly, the state explosion problem limits the bounds of what `herd` can test. Consider Fig. 11 that extends Fig. 7 with an additional thread `P2`. If Fig. 11 is compiled and simulated under the Arm AArch64 model [27], then `herd` does not terminate with a one hour timeout. Since `herd` enumerates executions, it suffers from the state-explosion problem. Without optimisation, execution time of assembly litmus tests expands factorially as the test size increases, practically limiting `herd`'s ability to scale much above programs of the order of 40-50 lines of code.

We sidestep the state explosion problem by optimising compiled litmus tests. The problem depends on the size of compiled program executions. Whilst `herd` considers `int r0 = *x` to be one load of x and one store to `r0`, the compiled program uses many instructions. For every C/C++ access in the source program, LLVM or GCC generates at least three Arm assembly instructions: `ADRP` to calculate the pointer to x, a `LDR` to load the location x into a register, and a `LDR` to load the value of x. As each instruction generates multiple loads or stores, the number of events in target executions is an order of magnitude larger than executions of the source. Computing whether such a graph is allowed using `herd` induces a state explosion as each `LDR` contributes to the reads-from relation (def. II.1). We optimise `ADRP *x;LDR;LDR/STR x` $\rightsquigarrow$ `LDR/STR x` sequences in Téléchat, and contribute a suite of similar optimisations for each architecture we test. Using Téléchat, simulating the compiled Fig. 11 terminates in milliseconds. Checking the soundness of our optimisations is future work, but an informal argument is that the `herd` simulator uses symbolic locations. The locations associated with accesses we remove cannot be named by other threads and an access cannot side-effect other symbolic locations. Soundness depends on the non-interference of other threads after applying our optimisations. Scalability is a still problem in theory, but in practice we only see timeouts with large (5+ threads or 6+ shared variables) tests.

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

void P0 (int* y,int* x) {
  int r0 = *x;
  atomic_thread_fence(memory_order_relaxed);
  *y = 1;
}
void P1 (int* z,int* y) {
  int r0 = *y;
  atomic_thread_fence(memory_order_relaxed);
  *z = 1;
}
void P2 (int* z,int* x) {
  int r0 = *z;
  atomic_thread_fence(memory_order_relaxed);
  *x = 1;
}
exists (P0:r0=1 /\ P1:r0=1 /\ P2:r0=1)
```

Fig. 11. A C/C++ litmus test, when compiled targeting Arm AArch64 does not terminate under simulation.

We end by discussing our working hypothesis. Many authors [32], [42], [66] claim simulation is unlikely to scale. For instance, Morisset et al. claim [66]: *"[herd] is unlikely to scale to the complexity of hunting C11/C++11 compiler bugs."* There is however a decade of evidence to suggest small (two threads at around twenty LoC) litmus tests are effective in finding bugs in hardware implementations [10], Arm hardware designs [44], GPUs [1], the Linux Kernel [9], C/C++ [48], [74], and more. Since every concurrency compilation bug we know of can be demonstrated by a small litmus test, we question whether simulation *needs* to scale. We expect the *small-scope hypothesis* [68] holds: "that a high proportion of errors can be found by testing a program for all test inputs within some small scope". Whether there are bugs that are triggered by large-programs *only* is an area for future work.

F. Industry Impact

We improved compilers - used in industry - in two ways. We answered queries from Arm's partners [58] and deployed automated regression testing for Arm Compiler. Téléchat is the first tool of its kind to be deployed in industry.

We assisted Arm's engineers with a query from Google [58] engineers. Following compelling performance metrics on hardware, Google's 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 reorderings than `LDAR`, however experts failed to find a bug under this proposal. Reviewers were inclined to accept the proposal without a correctness proof, but the proof was estimated to take three months. With Téléchat, we provided experimental testing of the proposal and Arm's compiler team chose to accept the proposal based on our work [58].

Lastly, we deployed automatic regression testing of Arm Compiler. Arm's compiler teams wish to test whether Arm Compiler correctly translates concurrent C/C++ programs targeting the Armv7 and Armv8 architectures. We conduct differential testing of Arm Compiler and deployed Téléchat in their automated testing infrastructure using an artefact like the one we provide with this work. As far as we know, Téléchat is the first compiler testing tool (for concurrency) to be deployed in a production setting - it is an industry first.

V. Conclusions and Further Work

We present the Téléchat automated compiler testing technique for programs with relaxed memory concurrency semantics. We documented its design (§III-A) and implementation (§III-B) with a reproducible artefact. We show how it improves on the state-of-the-art (§IV-A) and the real-world benefits [58] Téléchat brings when assisting Arm's compiler team. We refute a claim [66] made by the state-of-the-art whilst exploring a novel kind of concurrency bug (§IV-B) that evaded detection until now. We conducted large-scale differential testing of LLVM and GCC (§IV-D), which is the most extensive concurrency test campaign to date as far as we know. We assisted Arm's compiler team with two queries [58] from Arm's partners, reported four new bugs [36]–[39], and found one behaviour known by concurrency experts but missed by the state-of-the-art (Fig. 7). We fixed a bug in the Armv7 model [35], and generated several new lines of inquiry whilst exploring the limitations of our work (§IV-E). Lastly, Téléchat is, as far as we know, the industry's first concurrency tool to be deployed in automated testing for Arm Compiler.

We sought a practical bug (def. II.3) finding technique for production compilers. By using official architecture models, we achieve this goal, but defer a soundness proof to future work. Proving soundness requires a model of ground truth [17]. Such a model changes with business pressures; thus fixing a ground truth is a challenge. Even the official Armv8 model [27] forbids many behaviours of the older Arm model as no implementer has built a machine that exploited its additional relaxations[3]. The C/C++ model sees similar evolutions [18], [19], [21], [48], [74] and challenges for compiler engineers to avoid out-of-thin-air behaviours. For soundness to hold as compilers and models are updated, automating model-based proof [23], [29], [69] for production compilers is desirable. In the absence of repeatable proof, Téléchat provides practical testing.

Test generation is an area for future work. Fig. 10 induces two bugs in past versions of LLVM and GCC. We expect that exploring the state-space of litmus tests or conducting mutation-based testing [47] will find more bugs. The Alive2 [60] tool finds bugs whilst exploring the state space of sequential tests; but it is unknown whether it scales to concurrent programs in light of the state explosion (§IV-E) problem. Windsor et al. [78] conduct metamorphic testing of LLVM, but miss the bugs we report [36], [37], [39] - we expect there are more bugs out there. We just need the tests to find them.

We present new lines of inquiry for compiler testing. The field of testing with (C/C++) models is over a decade old - we feared little progress could be made. Indeed, recent work [4], [20], [70] focuses instead on the *porting problem*. We uncovered new lines of inquiry that suggest there is still work to be done. One such line - inspired our `const` work (§IV-E) - is studying the interaction between sequential and concurrent C/C++. `const atomic` loads induce run-time crashes [36], but it is unclear *how* such types are used in practice. The call for clarity on the compilation of atomics increases as multi-core machines play an increasing role in our lives.

[3]The Armv8 model is experimentally stronger [62] than the Armv7 model.

Data-Availability Statement

The data that support the findings of this study are openly available in Zenodo https://doi.org/10.5281/zenodo.10204529, with the reference [41].

Acknowledgment

We thank supervisors James Brotherston and Earl Barr. Luc Maranget, Ana Farinha, Alastair Donaldson, John Wickerson, Tyler Sorensen, Shale Xiong, Alastair Reid, Peter Smith, Wilco Dijkstra, 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 paper are not endorsed by Arm or any other company mentioned.

Artefact Appendix

A. Abstract

The artefact consists of the Téléchat tool and scripts provided with this paper. Téléchat builds on the herd tool-suite [5] and its models. As such the results are liable to change. We acquired all badges. For comments please contact luke.geeson@cs.ucl.ac.uk.

B. Artefact Checklist

1) **Algorithm:** Téléchat.
2) **Program:** `l2c,c2s,s2l` and herdtools [5].
3) **Compilation:** Includes LLVM 11, GCC 9.2, GCC 10.
4) **Models:** From herd toolsuite [5].
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:** 5GB for Docker image, +100GB for the large-scale study (§IV-D).
13) **Time needed to prepare workflow**: Everything is ready.
14) **Time needed to complete experiments**: $\sim$ 10 hours.
15) **Licenses:** CeCILL-B license.
16) **Workflow Frameworks:** Makefile, GNU Parallel [73].
17) **Archived(DOI):** https://doi.org/10.5281/zenodo.10204529
18) **Available:** Zenodo or Docker Hub[4].

C. Description

1) How Delivered: The artefact is available on Zenodo and consists of a Docker container with the Téléchat tool, compilers under test, and scripts required to reproduce results.

2) Hardware Dependencies: Either an Intel x86-64 or Arm AArch64 based machine. The artefact 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).

3) 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 [73], libxml2, time, and libc6

D. Installation

1) Download and install Docker. For example on Ubuntu 20.04 you can install docker using the official guide[5]
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
```

[4]https://hub.docker.com/r/lukeg101/telechat-artefact/tags
[5]https://docs.docker.com/desktop/install/ubuntu/

E. 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.

F. Paper Claims

1) Fig. 7 has outcomes in Fig. 8 (left). under the RC11 model [48], when compiled for Arm AArch64, it has the Fig.8 (right) outcomes.
2) Windsor et. al miss [78] miss the load buffering behaviour of Fig.7. Téléchat observes it.
3) We exercise all the features in Table III. when testing LLVM and GCC for the architectures listed.
4) We get the results in Table IV under the RC11 model [48], but if we permit load-to-store reordering all positive differences disappear.
5) Compiling and Optimising Fig.11 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[6]

G. 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:

```
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;]
```

Claim 2 (< 1 minute checking manually): Windsor et. al [78] state:

> "*we experimented with using the stronger RC11 memory model of Lahav et al. [48] as the input to our test-case generator, RMEM [a simulator parameterised over architecture models, not part of C4] identified as a bug the 'load buffering' test that RC11 forbids, but C11 and AArch64 permit.*"

[6]https://lukegeeson.com/blog/2023-10-17-Telechat-Bug-Board/

We observe load buffering (Figs.7+8) in **Claim 1**.

Claim 3 (∼ 10 hours on a 224 core ThunderX2): Please run:

```
artefact> make all CONF_FILE=c11.conf
```

Warning: This requires a powerful machine to run.

Once done, the `Output` directory should reveal tests that contain the following: `fence`, `*x`, `if`, `atomic_load`, `atomic_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 (∼ 10 hours on a 224 core ThunderX2): Please run:

```
artefact> make all CONF_FILE=c11.conf
```

Once done, the numbers in Table IV should match the *+ve* and *-ve* differences listed on the console output. Since the C/C++ standards permit load-to-store re-ordering (ie load buffering), observe that all of the *+ve* 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) Fig.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):

```
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
```

On the other hand, Consider the `unoptimised.litmus` test, adapted from LLVM-11 code taken from godbolt.org[7] (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
This motivates our need to optimise compiled litmus tests.

H. 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`):
  ```
  artefact> make examples \
    CMEM=c11_simp.cat
  ```
- Set simulation timeout, (default `120.0` seconds):
  ```
  artefact> make examples TIMEOUT=1.0
  ```
- Test other compilers. Outside the container, 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`.

I. Available Benchmarks

The benchmarks used can be generated by providing a `CONF_FILE` parameter to the `Makefile`:

- §IV.D: `c11.conf`: for the large-scale differential testing
- §IV.F: `c11_acq.conf`: for the LDAPR case study

This article represents a personal opinion that is not endorsed by Arm.

[7]https://godbolt.org/z/G9b4Pq1YK

REFERENCES

[1] ALGLAVE, J., BATTY, M., DONALDSON, A. F., GOPALAKRISHNAN, G., KETEMA, J., POETZL, D., SORENSEN, T., AND WICKERSON, J. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, p. 577–591.

[2] ALGLAVE, J., COUSOT, P., AND MARANGET, L. Syntax and semantics of the weak consistency model specification language cat. *CoRR abs/1608.07531* (2016).

[3] ALGLAVE, J., DEACON, W., GRISENTHWAITE, R., HACQUARD, A., AND MARANGET, L. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst. 43*, 2 (July 2021).

[4] ALGLAVE, J., KROENING, D., NIMAL, V., AND POETZL, D. Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion. *ACM Trans. Program. Lang. Syst. 39*, 2 (May 2017).

[5] ALGLAVE, J., AND MARANGET, L. herdtools7. https://github.com/herd/herdtools7, 2021. Accessed: 2019-10-06.

[6] ALGLAVE, J., AND MARANGET, L. NEON Architecture Tests. https://github.com/herd/herdtools7/tree/master/herd/tests/instructions/AArch64.neon, 2021. Accessed: 2022-11-29.

[7] ALGLAVE, J., AND MARANGET, L. SVE Architecture Pull Request. https://github.com/herd/herdtools7/pull/414, 2021.

[8] ALGLAVE, J., AND MARANGET, L. ARM memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/arm.cat, 2022.

[9] ALGLAVE, J., MARANGET, L., MCKENNEY, P. E., PARRI, A., AND STERN, A. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 405–418.

[10] ALGLAVE, J., MARANGET, L., SARKAR, S., AND SEWELL, P. 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* (Berlin, Heidelberg, 2011), TACAS'11/ETAPS'11, Springer-Verlag, pp. 41–44.

[11] ALGLAVE, J., MARANGET, L., SARKAR, S., AND SEWELL, P. Fences in Weak Memory Models (Extended Version). *Form. Methods Syst. Des. 40*, 2 (Apr. 2012), 170–205.

[12] ALGLAVE, J., MARANGET, L., AND TAUTSCHNIG, M. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst. 36*, 2 (July 2014), 7:1–7:74.

[13] ARM-LIMITED. Arm Morello Program. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello. Accessed: 2023-03-23.

[14] ARM-LIMITED. *Arm Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2023.

[15] ARMSTRONG, A., CAMPBELL, B., SIMNER, B., PULTE, C., AND SEWELL, P. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *In Proc. 33rd International Conference on Computer-Aided Verification* (July 2021).

[16] ATIG, M. F., BOUAJJANI, A., BURCKHARDT, S., AND MUSUVATHI, M. On the Verification Problem for Weak Memory Models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2010), POPL '10, Association for Computing Machinery, p. 7–18.

[17] BARR, E. T., HARMAN, M., MCMINN, P., SHAHBAZ, M., AND YOO, S. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Softw. Eng. 41*, 5 (May 2015), 507–525.

[18] BATTY, M., DONALDSON, A. F., AND WICKERSON, J. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL '16, Association for Computing Machinery, p. 634–648.

[19] BATTY, M. J. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, 2014.

[20] BECK, M., BHAT, K., STRICEVIC, L., CHEN, G., BEHRENS, D., FU, M., VAFEIADIS, V., CHEN, H., AND HÄRTIG, H. 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* (2023), T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds., ACM, pp. 61–73.

[21] BOEHM, H.-J., AND ADVE, S. V. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, Association for Computing Machinery, p. 68–78.

[22] CHAKRABORTY, S., AND VAFEIADIS, V. Validating Optimizations of Concurrent C/C++ Programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (New York, NY, USA, 2016), CGO '16, ACM, pp. 216–226.

[23] CHAKRABORTY, S., AND VAFEIADIS, V. Grounding Thin-air Reads with Event Structures. *Proc. ACM Program. Lang. 3*, POPL (Jan. 2019), 70:1–70:28.

[24] CHEN, J., PATRA, J., PRADEL, M., XIONG, Y., ZHANG, H., HAO, D., AND ZHANG, L. A Survey of Compiler Testing. *ACM Comput. Surv. 53*, 1 (Feb. 2020).

[25] COLIN, S. RC11 Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/rc11.cat, 2022. Accessed: 2022-06-30.

[26] COOK, S. A. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1971), STOC '71, ACM, p. 151–158.

[27] DEACON, W., AND ALGLAVE, J. Armv8 `AArch64` Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat, 2021.

[28] DIJKSTRA, W. Bug 108891 - libatomic: AArch64 SEQ_CST 16-byte load missing barrier. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108891.

[29] DODDS, M., BATTY, M., AND GOTSMAN, A. Compositional Verification of Compiler Optimisations on Relaxed Memory. In *Programming Languages and Systems* (Cham, 2018), A. Ahmed, Ed., Springer International Publishing, pp. 1027–1055.

[30] DONALDSON, A. F., EVRARD, H., LASCU, A., AND THOMSON, P. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017).

[31] EIDE, E., AND REGEHR, J. Volatiles Are Miscompiled, and What to Do About It. In *Proceedings of the 8th ACM International Conference on Embedded Software* (New York, NY, USA, 2008), EMSOFT '08, ACM, pp. 255–264.

[32] GAVRILENKO, N., PONCE DE LEÓN, H., FURBACH, F., HELJANKO, K., AND MEYER, R. *BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings*. 07 2019, pp. 355–365.

[33] GCC-MAILING-LIST. [PATCH, AArch64 v2 05/11] aarch64: Emit LSE stop instructions. https://gcc.gnu.org/legacy-ml/gcc-patches/2018-10/msg01960.html, 2018. Accessed: 2020-13-06.

[34] GCC-MAILING-LIST. [PATCH, AArch64 v2 05/11] aarch64: Emit LSE stop instructions. https://gcc.gnu.org/legacy-ml/gcc-patches/2018-10/msg02042.html, 2018. Accessed: 2020-13-06.

[35] GEESON, L. Added dmb ish to arm model. https://github.com/herd/herdtools7/pull/385, 2022. Accessed: 2022-11-26.

[36] GEESON, L. [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.

[37] GEESON, L. [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, 2023. Accessed: 2023-05-11.

[38] GEESON, L. [AArch64]: Atomic Exchange Allows Reordering past Acquire Fence . https://github.com/llvm/llvm-project/issues/68428, 2023.

[39] GEESON, L. [AArch64][CodeGen]: LD{AX}P/S{LX}TP endian swap. https://github.com/llvm/llvm-project/issues/61431, 2023.

[40] GEESON, L. branch delay slots are not filled with atomic stores. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=110573, 2023.

[41] GEESON, L., AND SMITH, L. CGO Artefact for Compiler Testing With Relaxed Memory Models. https://doi.org/10.5281/zenodo.10411403, Dec. 2023.

[42] HAAS, T., MEYER, R., AND PONCE DE LEÓN, H. CAAT: Consistency as a Theory. *Proc. ACM Program. Lang. 6*, OOPSLA2 (oct 2022).

[43] HEIDEKRÜGER, P., AND ELVER, M. Status report: Broken dependency orderings in the linux kernel. https://lpc.events/event/16/contributions/1174/, 2022. Accessed: 2022-26-11.

[44] HSIAO, Y., MULLIGAN, D. P., NIKOLERIS, N., PETRI, G., AND TRIPPEL, C. Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2021), MICRO '21, ACM, p. 679–694.

[45] INTEL. Pin. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, 2013.

[46] ISO-C-STD, O. ISO/IEC 9899:201x. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2912.pdf, 2022. Accessed: 2023-11-10.

[47] KUSANO, M., AND WANG, C. CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (2013), ASE'13, IEEE Press, p. 722–725.

[48] LAHAV, O., VAFEIADIS, V., KANG, J., HUR, C.-K., AND DREYER, D. Repairing Sequential Consistency in C/C++11. PLDI 2017, ACM, pp. 618–632.

[49] LAMPORT, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput. 28*, 9 (Sept. 1979), 690–691.

[50] LASCU, A., WINDSOR, M., DONALDSON, A. F., GROSSER, T., AND WICKERSON, J. Dreaming up Metamorphic Relations: Experiences from Three Fuzzer Tools. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)* (2021), pp. 61–68.

[51] LE, V., AFSHARI, M., AND SU, Z. Compiler Validation via Equivalence modulo Inputs. *SIGPLAN Not. 49*, 6 (June 2014), 216–226.

[52] LEROY, X. Formal Verification of a Realistic Compiler. *Commun. ACM 52*, 7 (July 2009), 107–115.

[53] LIDBURY, C., LASCU, A., CHONG, N., AND DONALDSON, A. F. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, ACM, p. 65–76.

[54] LLVM. AArch64 Dead register definitions. https://llvm.org/docs/doxygen/AArch64DeadRegisterDefinitionsPass_8cpp_source.html.

[55] LLVM-BUGZILLA. [AArch64] atomicrmw on Armv8.1-a memory ordering can be changed. https://bugs.llvm.org/show_bug.cgi?id=35094, 2019. Accessed: 2020-13-06.

[56] LLVM-PHABRICATOR. [AArch64] Fix for bug 35094 atomicrmw on Armv8.1-A+lse. https://reviews.llvm.org/D58348, 2019.

[57] LLVM-PHABRICATOR. AAArch64: use ldp/stp for 128-bit atomic load/store in v.84 onwards. https://reviews.llvm.org/rG13aa102e07695297fd17f68913c343c95a7c56ad, 2021.

[58] LLVM-PHABRICATOR. Add support for LDAPR. https://reviews.llvm.org/D126250, 2022. Accessed: 2022-11-22.

[59] LLVM-PHABRICATOR. [WoA] Use fences for sequentially consistent stores/writes. https://reviews.llvm.org/D141748, 2023.

[60] LOPES, N. P., LEE, J., HUR, C.-K., LIU, Z., AND REGEHR, J. *Alive2: Bounded Translation Validation for LLVM*. Association for Computing Machinery, New York, NY, USA, 2021, p. 65–79.

[61] MARANGET, L. RISC-V Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/riscv.cat, 2022.

[62] MARANGET, L. ARM model vs. AArch32 model. https://cambium.inria.fr/~maranget/cats7/aarch32/, 2023.

[63] MARANGET, L., AND ALGLAVE, J. IBM PowerPC Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/ppc.cat, 2022.

[64] MARANGET, L., AND ALGLAVE, J. MIPS Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/mips.cat, 2023.

[65] MARANGET, L., AND ALGLAVE, J. x86-64 Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso-mixed.cat, 2023.

[66] MORISSET, R., PAWAN, P., AND ZAPPA NARDELLI, F. 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* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 187–196.

[67] NIMAL, V. P. J. *Static analyses over weak memory*. PhD thesis, University of Oxford, UK, 2014.

[68] OETSCH, J., PRISCHINK, M., PÜHRER, J., SCHWENGERER, M., AND TOMPITS, H. On the Small-Scope Hypothesis for Testing Answer-Set Programs. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning* (2012), KR'12, AAAI Press, p. 43–53.

[69] PODKOPAEV, A., LAHAV, O., AND VAFEIADIS, V. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang. 3*, POPL'19 (Jan. 2019).

[70] ROCHA, R. C. O., SPROKHOLT, D., FINK, M., GOUICEM, R., SPINK, T., CHAKRABORTY, S., AND BHATOTIA, P. 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* (New York, NY, USA, 2022), PLDI 2022, Association for Computing Machinery, p. 888–902.

[71] SARKAR, S., SEWELL, P., ALGLAVE, J., MARANGET, L., AND WILLIAMS, D. Understanding POWER Multiprocessors. PLDI '11, ACM, pp. 175–186.

[72] SARKAR, S., SEWELL, P., NARDELLI, F. Z., OWENS, S., RIDGE, T., BRAIBANT, T., MYREEN, M. O., AND ALGLAVE, J. The Semantics of x86-CC Multiprocessor Machine Code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2009), POPL '09, ACM, pp. 379–391.

[73] TANGE, O. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine 36*, 1 (Feb 2011), 42–47.

[74] VAFEIADIS, V., BALABONSKI, T., CHAKRABORTY, S., MORISSET, R., AND ZAPPA NARDELLI, F. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. POPL '15, ACM, pp. 209–220.

[75] ŠEVČÍK, J., VAFEIADIS, V., ZAPPA NARDELLI, F., JAGANNATHAN, S., AND SEWELL, P. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM 60*, 3 (June 2013).

[76] WICKERSON, J., BATTY, M., SORENSEN, T., AND CONSTANTINIDES, G. A. Automatically Comparing Memory Consistency Models. POPL 2017, ACM, pp. 190–204.

[77] WINDSOR, M., DONALDSON, A. F., AND WICKERSON, J. C4: The C Compiler Concurrency Checker. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2021), ISSTA 2021, ACM, p. 670–673.

[78] WINDSOR, M., DONALDSON, A. F., AND WICKERSON, J. High-coverage metamorphic testing of concurrency support in C compilers. *Software Testing, Verification and Reliability* (2022), e1812.

[79] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 283–294.