SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

High-coverage metamorphic testing of concurrency support in C compilers

Matt Windsor[1]*, Alastair F. Donaldson[2] and John Wickerson[2]

[1] *University of York, York, United Kingdom*
[2] *Imperial College London, London, United Kingdom*

SUMMARY

We present a technique and automated toolbox for randomized testing of C compilers. Unlike prior compiler-testing approaches, we generate *concurrent* test cases in which threads communicate using fine-grained atomic operations, and we study actual compiler *implementations* rather than abstract mappings. Our approach is: (1) to generate test cases with precise oracles directly from an axiomatisation of the C concurrency model, (2) to apply metamorphic fuzzing to each test case, aiming to amplify the coverage they are likely to achieve on compiler codebases, and (3) to execute each fuzzed test case extensively on a range of real machines.
Our tool, C4, benefits compiler developers in two ways. First, test cases generated by C4 can achieve line coverage of parts of the LLVM C compiler that are reached by neither the LLVM test suite nor an existing (sequential) C fuzzer. This information can be used to guide further development of the LLVM test suite, and can also shed light on where and how concurrency-related compiler optimisations are implemented. Second, C4 can be used to gain confidence that a compiler implements concurrency correctly. As evidence of this, we show that C4 achieves high strong mutation coverage with respect to a set of concurrency-related mutants derived from a recent version of LLVM, and that it can find historic concurrency-related bugs in GCC. As a by-product of concurrency-focused testing, C4 also revealed two previously-unknown sequential compiler bugs in recent versions of GCC and the IBM XL compiler.

Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: atomics, C11, concurrency, fuzz testing, program generation, weak memory

1. INTRODUCTION

Though often derided for a perceived inherent lack of safety, C remains ubiquitous in modern systems programming. One explanation [1] is that C provides a workable interface that connects disparate hardware and software systems through manipulation of shared memory. The safety of this set-up relies on the correct actions of both programmer (who must avoid undefined and unsafe behaviour) and compiler (which must faithfully map the programmer's intent to machine code) with respect to the C standard.

*Correspondence to: Matt Windsor, Department of Computer Science, University of York, York, United Kingdom. E-mail: matt.windsor@york.ac.uk

Contract/grant sponsor: UK Research Institute in Verified Trustworthy Software Systems

Contract/grant sponsor: UK Engineering and Physical Sciences Research Council; contract/grant number: EP/R006865/1

An increasingly important form of shared-memory manipulation, and one where ensuring safety is especially hard, is the use of atomic memory operations ('atomics'). Atomics are a key part of efficient concurrent algorithms, themselves crucial for performance in a multi-core world [2]. By manipulating memory in small-scale, atomic steps, one can design high-performance data structures such as the time-stamped stack [3] and the work-stealing queue [4]. Recognising this, the 'C11' revision of the C standard[5]† offers built-in support for atomics.

It is important that C compilers implement atomics correctly, both to ensure the reliability of concurrent systems code written in C11, and because mismatches between a compiler's expected and actual behaviour undermine source-level analysis and verification [6]. The impact of concurrency-related compiler bugs appears to be limited at present: if one searches the LLVM and GCC bug databases for relevant keywords such as 'atomic' and 'concurrent' it is apparent that concurrency-related compiler bugs are vastly outnumbered by regular, sequential compiler bugs. However, mistakes have been found in compiler mappings from C atomics to low-level instructions [7], even when such mappings were the subject of rigorous proof attempts. Furthermore, as concurrency becomes increasingly ubiquitous, compiler-writers are likely to implement more adventurous concurrency-related optimisations, with increased potential for introducing bugs.

In this article we investigate techniques for automatically generating concurrent test cases for C compilers, to arm compiler developers with a means of defending against such future bugs. Our goal is to generate concurrent tests that: (a) include oracles, so that they can be used to assess correctness of compilation, not merely freedom from internal compiler errors, (b) achieve high line coverage of the components of a compiler that relate to concurrency, which can be assessed by comparing to the line coverage obtained by standard test suites and existing test generators, and (c) are likely to be effective at identifying concurrency-related regressions in a compiler, which can be validated using mutation testing and with respect to historic concurrency-related compiler bugs.

1.1. State of the Art

Compiler testing is an active research area [8], and related techniques are also being used to test static analysers [9], debuggers [10, 11], and SMT solvers [12]. Several empirical studies related to compiler bugs and compiler testing have been conducted [13, 14, 15]. Yet no method currently exists to check *automatically* that *concurrency* constructs (including atomics) are compiled correctly by *mainstream* compilers.

- Several tools have successfully applied *differential testing* to compilers [16, 17, 18]: compiling one program with multiple compilers or compiler configurations and assuming that differences in the compiled programs' outputs reflect differences (and therefore bugs) in the compilers. Our work aims to test correct compilation of *concurrent* programs, whose inherent nondeterminism breaks this assumption.
- Conversely, *metamorphic testing* techniques [19, 20] can be applied in the context of compiler testing, by compiling two *equivalent* programs with the same compiler and assuming that the compiled programs will produce different outputs only if a miscompilation has occurred. Examples include equivalence-modulo-inputs testing (EMI) [21, 22, 23], where equivalent programs are derived from an original program by applying code mutations that, thanks to coverage analysis, are known to have no effect for particular inputs; the application of more general semantics-preserving transformations to create families of equivalent programs [24, 25, 26]; and generation of programs that are equivalent by construction [27]. Again, concurrency-induced nondeterminism makes the assumption on which these methods depend unsustainable.
- Formal approaches have been used to verify the correctness of peephole optimizations in LLVM [28], and to certify that particular LLVM optimizations lead to generation of correct code [29], but these approaches only relate to sequential code optimizations.

†These atomics technically derive from those added in the 2011 revision of C++; this said, the two systems are sometimes interchangeably referred to as 'C11 atomics'.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

- Sevcik et al. [30] and Beringer et al. [31] have both extended the CompCert verified compiler [32] to handle concurrent C. However, similar proofs about mainstream compilers would be infeasible. They could also be quite fragile, given that the interpretation of the C11 concurrency semantics is still not a completely settled matter [33, 34]. Our work treats the compiler as a black box, and the concurrency semantics as a parameter, and hence will be able to keep up with evolving compilers and language standards.
- Lidbury et al. [35] and Jiang et al. [36] have applied random testing to OpenCL and CUDA compilers respectively, but the concurrent test cases they use are deterministic by construction, unlike real-world examples. Our work involves testing compilers on more general concurrent code.
- Morriset et al. [37] have used random testing to check that GCC preserves C's concurrency semantics. However, they cannot assess the correctness of compiler transformations that involve atomics, which we do. Moreover, they only generate sequential test cases, whereas our work involves validating the compilation of multi-threaded programs. This said, some of the program transformations we consider in our work aim to build test cases with similar properties to those of Morriset et al.
- Chakraborty et al. [38] check whether LLVM transformations preserve C's concurrency semantics; our work involves checking the entire compilation process from source to assembly.
- Several authors have assessed the correctness of compiler *mappings*, whether by proof [39, 40] or by automatic checking [41, 42]. However, these mappings are only an abstraction of the full compiler; our work involves checking that mainstream compilers actually implement these mappings correctly in the presence of all the other optimisations that compilers perform.

The technique we present is based on metamorphic testing, and in the remainder of the paper we show how this differs from prior approaches to metamorphic compiler testing (discussed above) that assume sequential execution and identify bugs based on result mismatches between equivalent programs. Our extension to the concurrent setting is specific to the use of metamorphic testing in the context of compilers. A metamorphic technique has been proposed for testing debuggers [10], involving metamorphic transformations such as adding dead code to a program and adapting all break-points used in debugging accordingly. This technique also appears to assume sequential execution, so that the next break-point that the program under test will hit is deterministic. Adapting this technique to a concurrent setting would require different innovations to the ones we propose here for compiler testing.

1.2. Our approach

Any effective approach for testing the compilation of C11 atomics must address the difficulty of finding test oracles [43] for nondeterministic programs; there is no single 'ground truth' result for a particular test case. We thus cannot follow the approach used in experiments based on the CSMITH tool [18] and apply differential testing [16, 17], which identifies bugs via mismatches between the results of running a well-defined program compiled by multiple distinct compilers: in our context results may legitimately differ between runs due to concurrency-related nondeterminism.

If we can identify *all* ground truths of a concurrent program, we can flag any other results as a potential bug. This can be achieved in principle by exhaustively simulating the program against the C11 memory model using tools such as HERD [44]. However, simulation scales poorly as we introduce branching control flow to test cases, making this approach infeasible for the sorts of complex test cases we anticipate being required to achieve high-coverage testing.

We overcome this problem by combining memory-model-driven generation of small programs that *can* be exhaustively simulated, together with *metamorphic fuzzing* [20, 45, 24] to transform these programs into forms that are more complex but still amenable to the oracle obtained via simulation. We aim to challenge the compiler under test both *statically* (by mutating the source code in ways that could confuse the compiler, covering under-tested code paths) and *dynamically* (by repeatedly executing the program alongside several memory-hammering threads that could expose

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

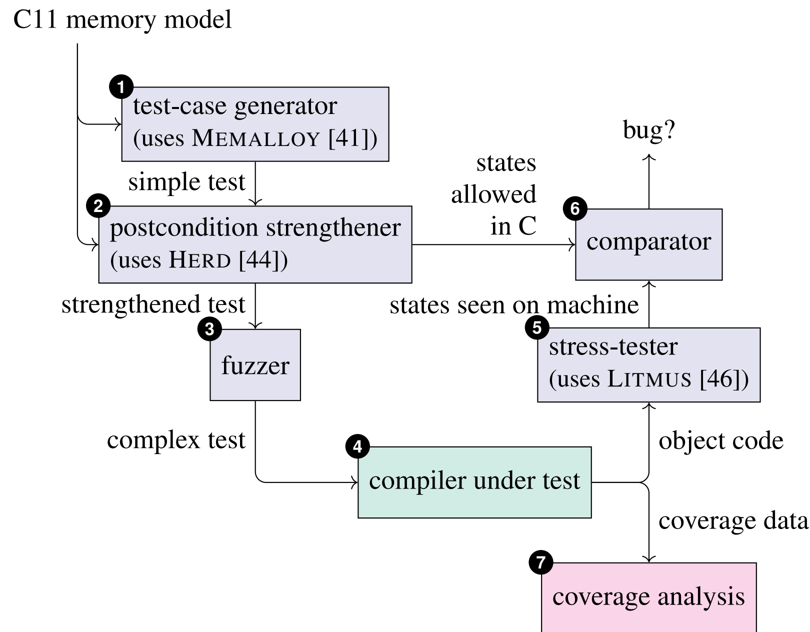

Figure 1. Our compiler-testing infrastructure.

places where the compiler has not inserted sufficient synchronisation instructions). The detailed workflow is depicted in fig. 1 and explained below.

❶ We **generate** a small test case containing a multi-threaded C program and a postcondition over its final states. Following Wickerson et al. [41] and Lustig et al. [47], these test cases are generated by using a SAT solver to search for executions that are forbidden by the C memory model, and so the postcondition describes precisely one unwanted outcome.

❷ We **strengthen** [48] the postcondition of the test case so that rather than detecting *one* forbidden outcome, it detects *all* forbidden outcomes of that program. To do so, we simulate the test with HERD [44] to produce the exhaustive set of outcomes that *are* allowed by the memory model, and have the postcondition require that all outcomes inhabit that set.

❸ We **fuzz** the test case: applying mutations (such as dead-code introduction [35]) in the hope of coaxing the compiler under test into revealing bugs. In the spirit of EMI testing [21], we design these mutations so as not to introduce new states over the variables in the original test case, and so the postcondition generated by step ❷ remains valid.

❹ We **compile** the test case using the compiler under test. If compilation fails, e.g. due to a crash or an internal compiler error, then we have found a bug.

❺ We **execute** the compiled object code on a real machine. This is done in a 'stressful' environment: leveraging the LITMUS tool of Alglave et al. [46], the compiled program is run many times, in the presence of extra concurrent threads that hammer on the memory system in various ways.

❻ We **check** whether any outcome produced by step ❺ is disallowed at the C level; if so, we have found a bug.[‡]

❼ If coverage information is available for the compiler under test, we can compare the coverage achieved via C4-generated tests with the coverage obtained by a source of *non*-concurrent programs (e.g. programs generated by CSMITH) or a standard manually-constructed regression test suite (such as the LLVM test suite [49]). This is useful to help

[‡]During development we discovered and debugged a number of defects in the transformations that C4 performs after the comparator reported unexpected states.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

understand the effectiveness of C4 [50], and to provide guidance on where the regression test suite could be improved.

1.3. Our contributions and key findings

To explore the approach in fig. 1, we present the design and implementation of C4 (C Compiler Concurrency Checker), an open-source tool for generating and running concurrent compiler tests. We report on experiments using C4 over several versions of four compilers (GCC, LLVM, the Intel C compiler, and IBM XL), and targeting four architectures (x86-64, POWER9, and both 32-bit and 64-bit Arm8).

- The first is a *line coverage* experiment (section 4.2) where we investigated the extent to which C4-generated tests provide coverage of the source code of LLVM, comparing this to the coverage achieved by fuzzing using CSMITH, and by running the LLVM test suite.
- Second, we report on a *mutation coverage* experiment (section 4.3) where we made systematic changes to the source code of LLVM, and ran C4 to see if it can detect those changes in the form of perceived bugs.
- Third, we performed *bug-finding* experiments (section 4.4) where we used C4 to test a variety of compilers.

Our key findings from these experiments are as follows.

- C4 covers a distinctive portion of LLVM not covered by CSMITH or the LLVM test suite, including large parts of the compiler that relate to handling of atomics. This includes optimisations focusing on removing idempotent modifications and simplifying certain types of comparison involving atomic actions. C4 thus has the potential to guard against new bugs in these parts of the compiler in a manner that CSMITH and the LLVM test suite cannot.
- C4 was able to achieve reasonably high strong mutation coverage over a set of 24 systematically-injected concurrency-related mutants: the tool killed 15 mutants with fuzzing disabled, rising to 17 mutants with fuzzing enabled.
- Using C4, we were able to independently discover two historical concurrency-related bugs in GCC 4.9, the first version of GCC to properly support C11 atomics, demonstrating that C4 can find real-world concurrency-related bugs.
- As a by-product of our testing campaign, C4 identified two previously unknown internal compiler errors in the IBM XL C/C++ compiler (confirmed) and a development build of GCC 11 (promptly fixed by the GCC developers), both related to compilation of sequential code.

While the extent and significance of concurrency-related bugs in today's compilers remains unclear, our results show that C4 is capable of achieving high line and mutation coverage of concurrency-related compiler code, and its ability to find historic bugs in GCC (bugs that we were not aware of when developing the tool) suggest that it is a useful line of defence against bugs related to future concurrency-related optimisations.

Companion material C4 is free and open-source software. It is available on GitHub at `https://c4-project.github.io/`. A brief announcement about C4 appeared in the tool demonstrations track at ISSTA 2021 [51], and some of the inspirations for C4's fuzzing actions were described in a workshop paper [52].

2. BACKGROUND

This section describes the background necessary to understand the rest of this article.

2.1. Atomic-action concurrency

In this work, we investigate *atomic-action concurrency*: concurrency in which threads exchange information through atomic updates (reads, writes, and other, more specialised read-modify-write

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

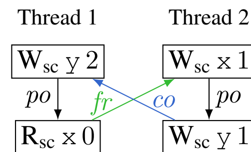

Figure 2. A 2-threaded execution forbidden by C11.

operations) to a shared memory. As discussed by Gramoli [53], atomic-action concurrency is key for producing high-performance algorithms without the use of heavyweight mutual exclusion mechanisms, but difficult to reason about both in terms of use and implementation. We hypothesise in this work that this difficulty extends to the handling of such concurrency by compilers as well as programmers.

2.2. *Weak-memory concurrency*

We further focus on situations in which atomic writes propagate to atomic reads with weaker guarantees than one would expect from the program orders of the threads ('sequential consistency'). For instance, modern processors may reorder atomic instructions to produce more efficient memory access, or the cache architecture of the processor may result in atomic reads drawing stale values from per-core buffers instead of newly-atomically-written values from another thread.

Such situations give rise to *relaxed*, or *weak*, memory consistency models [54]. Different processors have different memory models; often, reduced instruction set computers such as Arm and POWER have weaker models (fewer guarantees) than complex instruction set computers such as x86-64.

2.3. *The C11 memory model*

Programming languages, too, have their own memory models. In this work, the memory model is that of C11, as formalised by Batty et al. [55]. This model is a compromise between the weak behaviours of some processors and the strong guarantees needed by programmers in certain situations, and offers a sliding scale of guarantees ('memory orders') from *relaxed* (fewest guarantees) to *sequentially consistent* (most guarantees).

Many formalisations of the C11 memory model are based on an *axiomatic* semantics where one obtains the set of valid executions of a C11 program by enumerating all candidate executions, constructing particular relations over events in those executions and then removing those where the relations violate axioms of the memory model. These relations include:

- coherence (co), which associates writes w with later writes w' to the same location that overwrite the value written in w;
- from-read (fr), which associates reads r with writes w that occur after the value is read at r;
- program order (po), which orders reads and writes within individual thread bodies.

Figure 2 gives an example of an execution that comprises four write (W) and read (R) events, all using the default *sequential consistency* memory order parameter. Program order groups these events into two threads. Coherence order captures that the writing of 1 to y is overwritten by the writing of 2, while from-read captures that the writing of 1 to x overwrites the initial value of 0 seen by the reading of x. (Figure 3 shows a C11 test case that can give rise to this execution.)

The axiomatisation of the C11 memory model assigns increasingly large sets of axioms to increasingly strong memory orders. In the case of the above, the sequentially-consistent nature of the events means that the cycle between from-read, coherence, and program order is disallowed, and the execution in the diagram is not a valid outcome of a C11 program.

2.4. *Metamorphic testing and fuzzing*

Metamorphic testing [19] is the production of test cases by applying some form of metamorphosis to existing test cases, using known properties of the system under test to infer a relationship between

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

the output of the original and variant cases. We characterise this relationship with *metamorphic relations*. Following Lascu et al. [52], we define a metamorphic relation as a pair (R, S) of binary relations such that, given the system under test f, original test case t, and variant test case t',

$$(t, t') \in R \implies (f(t), f(t')) \in S.$$

In effect, R captures the metamorphosis and S some expected property that should hold over the output. A typical example with $f = \sin$ is:

$$R = \{(x, y) \mid y = x + \pi\}$$
$$S = \{(x, y) \mid y = -x\}.$$

Workflows involving metamorphic testing have been used to test compilers in the past [56, 21, 24], to avoid difficulties relating to the oracle problem [43]. Specifically, if we can construct two source code files t and t' with some known relationship R (for example, a particular semantics-preserving transformation), and infer a similar relationship S that we expect to see between the outputs of executing their compiled programs (in this instance, equality), then we can check $(f(t), f(t')) \in S$ for any compiler f without needing to know the precise nature of the expected behaviour of t or t'. (Strictly, f here denotes the process of compiling the given program and then also running the compiled executable.)

Part of the C4 workflow, which we discuss in section 3.3, involves a hybrid approach combining metamorphic testing and *fuzz testing*. Fuzz testing, or *fuzzing*, refers to techniques that test a system by feeding it inputs that have either been randomly generated or randomly mutated. Fuzzing techniques can be broadly categorised as *dumb* or *smart*, depending on whether they take advantage of knowledge of the input format of the system under test, and *black*-, *grey*- or *white*-box depending on the extent to which the fuzzing process is guided by the internals of the system under test. Early fuzzing techniques were successful in finding large numbers of bugs in UNIX utilities using dumb, black-box methods [57]. The CSMITH tool for compiler testing [18] is an example of a smart, black-box fuzzer: it has been designed to produce well-defined C programs, based on intimate knowledge of the C programming language, but generates programs based on a random seed, with no regard for the extent to which previously-generated programs exercised the compiler or compilers under test. The widely-used AFL [58] and libFuzzer [59] tools excel at finding security bugs in C/C++ programs by repeatedly mutating a corpus of inputs, using feedback on the coverage that an input achieves on the system under test to decide how to prioritise it for further mutation. Because the mutations that are applied are domain-agnostic, and only basic coverage information is used to guide the fuzzing process, this is an example of dumb grey-box fuzzing. White-box fuzzing techniques, such as SAGE [60], exploit more detailed information about the system under test by building a set of constraints characterising the execution of a given input, and then solving perturbed versions of these constraints to generate inputs that would follow different paths.

In our hybrid 'metamorphic fuzzing' approach, instead of using metamorphic relations to replace oracles, we use them to preserve an oracle over a small program as we transform it into a large program. Specifically, we select a random combination of metamorphoses to form R, which can be seen as transformations on C11 programs designed to make the program ever more challenging for compilers to handle correctly; S in turn is a form of refinement relationship over the behaviour of the program that ensures that test oracles devised over the original program can still be used with the fuzzer outputs. This is an instance of smart, black-box fuzzing: our transformations are designed based on intricate knowledge of the C language, but the fuzzing process is not guided by feedback from the compilers under test.

3. DESIGN AND IMPLEMENTATION OF C4

We describe the four main stages of C4: test-case generation (section 3.1), postcondition strengthening (section 3.2), test-case fuzzing (section 3.3), and test-case execution (section 3.4).

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

initial state: $\{x = 0; y = 0;\}$

```
/* Thread 1 */              /* Thread 2 */
int r0=0;
atomic_store(y,2);          atomic_store(x,1);
r0=atomic_load(x);          atomic_store(y,1);
```

assert $(\neg(x = 1 \wedge y = 2 \wedge r0 = 0))$

Figure 3. MEMALLOY-generated test case capturing fig. 2.

3.1. Test-case generation using MEMALLOY

Our approach depends on the generation of small initial test cases for subsequent processing by our C4 tool. For this, we depend on existing tools; the novelty here is not the tools themselves, but our use of them as part of a wider compiler testing strategy. Once we have a corpus of initial test cases, we can apply C4 to them indefinitely.

The first step — test-case generation — uses the MEMALLOY tool of Wickerson et al. [41]. This tool builds on a technique by Lustig et al. [47], based on the Alloy model checker [61], that generates minimal, yet comprehensive (up to a bounded size), sets of executions that a given memory model disallows. The MEMALLOY tool, also based on Alloy, synthesises executable test cases from those executions. By combining the two techniques, we can produce a conformance suite for any given memory model, as has already been done by Chong et al. [62] for transactional memory models and by Raad et al. [63] for persistent memory models. In our case, we use an axiomatisation of the C11 memory model, leading ultimately to C11 programs that check against the existence of an execution forbidden by that axiomatisation.

Figure 3 shows the test case that MEMALLOY generates from the execution in fig. 2. The test is constructed such that if the assertion fails, a forbidden execution must have occurred.

3.2. Postcondition strengthening using HERD

Each test case generated by MEMALLOY has a very specific oracle (its postcondition) that forbids exactly one illegal outcome. We *strengthen* this oracle so as to forbid *any* illegal outcome.

One way to generate a postcondition with this property is to perform exhaustive simulation of the test case against the C11 memory model, then encode the reachable final states in disjunctive normal form. To do so, we run the exhaustive HERD simulator of Alglave et al. [44] on the MEMALLOY test case. For the example test case in fig. 3, the generated postcondition is:

$$\text{assert} \begin{pmatrix} (x = 1 \wedge y = 1 \wedge r0 = 0) \\ \vee\, (x = 1 \wedge y = 1 \wedge r0 = 1) \\ \vee\, (x = 1 \wedge y = 2 \wedge r0 = 1) \end{pmatrix}$$

Of course, this approach would not scale to large test cases, but it is well-suited to the small programs that MEMALLOY produces.

3.3. Test case fuzzing

The steps mentioned above generate a relatively small number of tests, each capturing a minimal witness for a particular execution. While we could use these as a basis for compiler testing, we hypothesise that their limited quantity and size means that finding bugs using them alone is unlikely.

We have implemented a *fuzzer* as part of C4, which randomly transforms each test case generated using MEMALLOY and HERD to produce several larger test cases. To give an example of the fuzzer in action, listing 1 reproduces one result of fuzzing thread 1 of fig. 3. Aside from reformatting, this output is unabridged and unedited. In what follows, we explain how it is produced.

The output of the fuzzer must 'observationally refine' the input. By this, we mean that the fuzzer must not add to the input test case any new behaviour that can be observed through the overall result changing from 'all states satisfy the postcondition' to 'at least one state does not satisfy the

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Listing 1: Example fuzzer output for thread 1 of fig. 3. The original code is highlighted.

```
1  bool corridor_10 = false; int volatile r0 = 0;
2  atomic_store_explicit(y, 2, memory_order_seq_cst);
3  atomic_thread_fence(memory_order_seq_cst);
4  daring_xylophone:;
5  atomic_thread_fence(memory_order_acquire);
6  r0 = atomic_load_explicit(x, memory_order_seq_cst);
7  if (atomic_fetch_add_explicit(x, *gopher - *gopher,
8      memory_order_relaxed) ==
9    atomic_fetch_xor_explicit(x, *gopher - *gopher,
10     memory_order_relaxed)) {}
11 if (!(0 == (atomic_fetch_or_explicit(y, 0,
12     memory_order_acq_rel) & r0 ^
13   (273 ^ 273 | -2366) ^ (-207853 ^ -207853))))
14 { atomic_signal_fence(memory_order_acq_rel); } else {
15   atomic_signal_fence(memory_order_acq_rel);
16   if (!false) {} else {
17     atomic_fetch_add_explicit(x,
18       atomic_fetch_xor_explicit(x, -160955 ^ -160955,
19         memory_order_seq_cst),
20       memory_order_release);
21     return;
22     atomic_fetch_sub_explicit(x, 0, memory_order_seq_cst);
23     atomic_store_explicit(y,
24       atomic_load_explicit(y, memory_order_acquire),
25       memory_order_seq_cst);
26     return;
27   }
28 }
```

postcondition'. With our strengthened postconditions, this means that we cannot introduce any new final states that differ from existing ones except through newly-created variables. The reason for this restriction is straightforward: we want to re-use the existing test oracles attached to the fuzzer input without needing to attempt to re-simulate the larger, less tractable fuzzer output with HERD. Notably, the fuzzer *may* introduce new variables, as well as forms of synchronisation between existing variables (for example, new non-relaxed loads from variables, or arbitrary memory fences); it may even make test cases fail to terminate.

We must also avoid inserting executable[§] undefined behaviour ('UB'), including data races. This is because the semantics of UB in C is 'catch-fire': a program that executes UB may produce arbitrary results according to the C language standard, so any fuzzer action that introduces UB fails to produce an observational refinement. This is an issue in practice because compilers exploit the 'catch-fire' semantics by assuming that code is UB-free during compilation [64].

C4 satisfies these requirements with a fuzzing tool that works by applying sets of *actions* (small-scale transformation templates, such as 'surround with a loop that iterates only once' or 'replace a store with an exchange'), combined with randomly generated parameters, to the input test case. The pool of actions that C4 is able to perform are summarised in table I. C4 chooses actions randomly from this pool. Actions can specify preconditions over the current fuzzer state, so that they are only applied in appropriate contexts, and an action can be configured to recommend follow-up actions whose preconditions might become satisfied as a result of applying the action.

The presence of actions in the C4 fuzzer, as well as the requirement of observational refinement, gives rise to its connection to metamorphic fuzzing. Following Lascu et al. [52], we can express

[§]We can, and do, generate code that would produce UB *if executed*. However, C4 uses its knowledge of the test case and the C semantics to generate such code only inside control flows that cannot possibly be reached at run-time. This helps us find bugs such as that in section 4.4.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Category	Action
atomic	*cmpxchg.insert.int.arbitrary*: Arbitrary compare-exchanges
	cmpxchg.insert.int.fail: Compare-exchanges that always fail
	cmpxchg.insert.int.succeed: Compare-exchanges that always succeed
	fetch.insert.cond.boundary: Atomic fetches in conditionals that straddle a boundary condition
	fetch.insert.cond.negated-addend: Atomic fetch-adds compared against the negation of the addend
	fetch.insert.int.dead: Arbitrary atomic fetches into dead code
	fetch.insert.int.redundant: Fetches with no effect
	store.insert.dead: Arbitrary stores into dead code
	store.make.normal: Stores to unused variables
	store.insert.redundant: Stores that do not modify their target
	store.xchgify: Promotes stores to exchanges
dead	*insert.early-out*: Dead return/break/continue statements
	insert.early-out-loop-end: Dead-code pockets through break/continues, maybe in a known-true if statement, at loop ends
	insert.goto: Dead GOTO statements
if	*invert*: Flips an if statement
	surround.duplicate: Lifts statements into both branches of an if statement with an arbitrary condition
	surround.tautology: Lifts statements into the true branch of if statement with an always-true condition; the false branch is dead code
loop	*insert.for.kv-never*: For-loops with always-false comparisons based on variable known values; dead-code bodies
	insert.while.false: While-loops with always-false conditions; dead-code bodies
	surround.do.dead: Surrounds statements in dead-code with an arbitrary do-while loop
	surround.do.false: Surrounds statements with a do-while loop with an always-false condition
	surround.for.kv-once: Surrounds statements with a for-loop with a comparison, based on variable known values, that holds only once
	surround.for.simple: Surrounds safe-to-loop statements with a for-loop looping a small number of times
	surround.while.dead: Surrounds statements in dead-code with an arbitrary while loop
mem	*fence*: Inserts fences
	strengthen: Strengthens memory orders
prog	*label*: Inserts random program labels
	make.empty: Inserts new, empty, threads
var	*assign.insert.int.dead*: Arbitrary non-atomic assignments in dead code
	assign.insert.int.normal: Assignments to unused variables
	assign.insert.int.redundant: Assignments with no effect
	make: New variables
	volatile: Adds volatile qualifier to random locals

Table I. The fuzzer actions understood by C4.

every C4 fuzzer action a as a metamorphic relation (R, S):

$$R = \{(t, t') \mid t' = a(t)\}$$
$$S = \{(c, c') \mid \forall o' \in \mathrm{obs}(c').\, \exists o \in \mathrm{obs}(c).\, o = o'\,|_{\mathrm{var}(c)}\}$$

Here, $\mathrm{obs}(c)$ is the set of final states reachable by executing the command c, and $o'\,|_{\mathrm{var}(c)}$restricts the state o' to those variables mentioned in c.

While most of the actions come from our intuition as to which changes to programs may introduce bugs, some were inspired by previous work: for instance, the *dead.insert.early-out-loop-end* and *loop.surround.for.simple* actions together attempt to introduce bugs similar to one reported by Morisset et al. [37]. We also use dead-code introduction techniques (such as *if.surround.tautology*) which resemble those in CLsmith [35] and in GraphicsFuzz [65]; these are also similar to the EMI

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

technique, which changes the static control flow of a program by manipulating code that is detected to be unreachable at run-time via profiling [21].

To support actions such as dead-code introduction, C4 produces statically-known values by comparing fuzzer-generated variables to the *known values* assigned to them at initialisation. For instance, C4 can generate known Boolean values through expressions that compare the variable to its known value in ways that will be true, or false, if the variable still holds that value; it can generate zeroes through algebraic properties such as $x - x = 0$. C4 tracks reads of known values, restricting future writes to their variables; it also erases known values when writes invalidate them.

3.4. Running tests with C4 and LITMUS

Finally, C4 contains a test runner that automates multi-machine, multi-compiler tests. The tester takes a *corpus* of test cases and a configuration outlining which machines (and compilers hosted on those machines) are available, alongside other parameters such as parallel worker counts and cut-off thresholds. It then runs, in parallel for each machine, an endless loop of stages ❸–❺ from fig. 1.

A key step of our approach is the use of LITMUS to 'stress-test' the output of the compilers under test. LITMUS lifts C test cases into a C harness that runs the tests repeatedly in a stressful environment (one that contains large amounts of background concurrency, such as threads reading and writing to unused memory locations). The harness notes the final state after each execution, and outputs the results as a histogram of states.

There are various ways to involve the compiler under test in the building of the LITMUS harness. Our approach is to compile *both implementation and harness* with the compiler under test. This requires us to bypass the LITMUS build system and compile everything directly in one go. This simplifies the build process, but forces us to subvert the intended use of LITMUS to a greater degree.

In our experiments, we ran C4 and LITMUS on an x86-64 machine, compiling and running LITMUS harnesses on other remote test machines over SSH. This has advantages and disadvantages over running C4 separately on each machine. Copying data between machines incurs network overhead, and each machine must have its part of the infrastructure kept up to date. This said, our approach lets us configure and monitor a multi-machine experiment from one site, and we need not install anything on remote machines other than the compilers under test and the machine node.

4. EXPERIMENTS

This section reports on experiments we have performed with C4.

We performed two experiments relating to coverage: a line-by-line differential analysis (section 4.2), and an analysis of strong mutation coverage (section 4.3). We did so because although the number and nature of bugs found is a popular measure for use in evaluating a fuzzer [66], the reliability of the compilers under test is a clear confounding factor. A weak implementation may be riddled with bugs, making any reasonable fuzzer look effective according to this metric (e.g., simple fuzzing techniques recently found numerous bugs in FPGA synthesis tools [67], and an investigation of OpenCL compilers found several implementations too weak to be worthy of in-depth study [35]). At the other extreme, a fuzzer should not be expected to find *any* defects in a formally-verified implementation (as discussed in the context of testing the formally-verified CompCert compiler [32] with CSMITH [18]).

We then discuss experiments on a set of compilers under test that led to the discovery of real-world bugs using C4 (section 4.4).

4.1. Experimental setup

Our mutation and bug-finding experiments used three machines, one for each of the PPC64, x86-64, and Arm8 architectures. Table II lists the specifications of these machines.

For each compiler, we explored a variety of optimisation levels (`-O` arguments) and machine-specific optimisation profiles (`-march` and `-mcpu` arguments). C4 randomises both settings after each cycle of 20 fuzzed test cases, to ensure good coverage of the various permutations of available

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

	Processor	Cores	RAM	OS
PPC64	POWER9	160	120GB	Ubuntu 18.04.3
x86-64	Core i7-7700K	8	15.5GB	Ubuntu 18.04.3
Arm8	BCM2711 (Cortex-A72)	4	4GB	Raspbian *buster*

Table II. The experiment machines and their specifications.

compiler optimisations. For `-O` levels, we considered `-O0` (no optimisation) through `-O3` (heavy optimisation), `-Ofast` (optimisation that may break the C standard), and `-Os` (optimisation for program size).

We used the following machine-specific profiles (where accepted by the compiler):

PPC64 `mcpu={power9, power8, power7, powerpc64le, native}`, or the empty profile;

x86-64 `march={skylake, broadwell, x86-64, native}`, or the empty profile; and

Arm8 `mcpu=cortex-a72` or `march={armv8-a, armv7-a}` (but *not* the empty profile or `march=native`, as some Arm versions omit instructions needed by LITMUS).

4.2. Line coverage

To investigate the extent to which C4 exercises interesting parts of a representative compiler's treatment of concurrency, and how it compares with other bug-finding techniques, we performed extensive analysis of its line-by-line coverage of LLVM version 10.0.1 (the most recent release at the time we conducted the experiment).

We start by investigating two preliminary research questions:

RQ1 How variable is the coverage achieved by C4 and CSMITH?

RQ2 To what extent does the coverage achieved via C4 and CSMITH continue to increase as we run more tests?

Answering RQ1 and RQ2 allows us to ask more advanced questions that involve C4 and CSMITH: due to the randomized nature of these tools it is important to understand how stable the coverage they achieve is (RQ1), and we should be confident that we are using a large enough corpus of test programs that we are approaching the limit of coverable code for each fuzzer (RQ2).

We then investigate the following research questions that specifically probe how useful C4 is for exercising concurrency-related parts of the compiler under test:

RQ3 How complementary is the coverage that C4 achieves vs. CSMITH and the LLVM test suite? Is the complement concurrency-related?

RQ4 Can C4 hit concurrency-related compiler optimisations?

We believe that this method for investigating the code coverage achieved by compiler fuzzers – and our approach to answering RQ1 and RQ2 in particular – is an important contribution of the article that will be useful to other researchers.

We now discuss our findings for each research question in turn.

RQ1: How variable is the coverage achieved by C4 and CSMITH*?* To answer RQ1 (and the other research questions), we compiled LLVM 10.0.1 with gcov-based coverage enabled, and gathered coverage data on this compiler separately for two 100 000-program corpora: one generated by CSMITH, and one by C4. For each corpus, we obtained separate coverage metrics for each chunk of 10 000 programs, and also for the first 10 chunks of 1 000 programs. To gain coverage information across a range of optimization levels, we compiled each program four times – once at each of the levels `-O0` to `-O3`.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

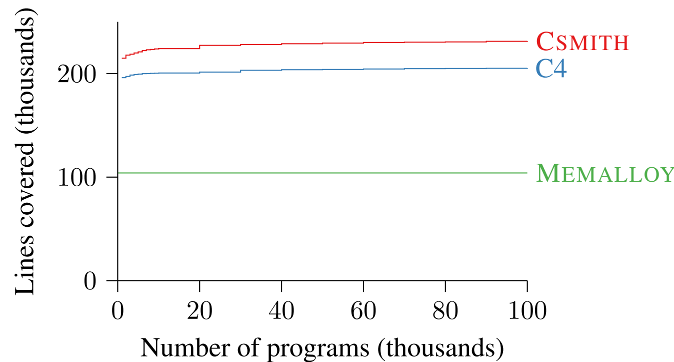

Figure 4. As we consider larger subsets of programs, the cumulative number of lines covered by those subsets increases.

We calculated the coefficient of variation (standard deviation divided by mean) for the line coverage achieved by each of C4 and CSMITH. We did this twice for each tool, once using results for the 10 chunks of 1 000 programs, and once using results for the 10 chunks of 10 000 programs. In all four cases, the coefficient was below 0.01.

Our answer for RQ1 is that the coverage achieved by all three tools has negligible variability if we use batches of at least 1 000 tests.

RQ2: To what extent does the coverage achieved via C4 and CSMITH *continue to increase as we run more tests?* Figure 4 shows the correlation between total number of programs considered and total line coverage, across C4 and CSMITH. For comparison, we also give the coverage achieved by the (unfuzzed) MEMALLOY corpus as a straight line. These figures derive from considering the coverage of each of the separate chunks discussed in section 4.2 cumulatively. While coverage continues to increase with each added chunk up to considering all 100 000 programs, much of it can be reached just with the first chunk of 1 000 programs.

Our answer for RQ2 is that there is little use in considering more than 100 000 programs for purposes of coverage analysis, and that in fact the increase from considering 10 000 programs to considering 100 000 is negligible.

RQ3: How complementary is the coverage that C4 achieves vs. CSMITH *and the LLVM test suite? Is the complement concurrency-related?* To answer this research question, we acquired coverage for the LLVM test suite [49] version 10.0.1,[¶] again using all optimization levels `-O0` to `-O3`. We used scripts provided with the gfauto tool chain [65] to investigate the differences in the coverage data obtained for our various workloads.

Figure 5 shows the line coverage achieved for LLVM by each of the CSMITH, C4, and LLVM test suite corpora, and the intersection relationships between the achieved coverage. We see that each approach reaches at least one thousand lines of code that the other approaches could not. It is unsurprising that CSMITH and the LLVM test suite have a clear advantage over C4 in terms of lines of code covered. CSMITH can generate sequential programs that cover a wide range of C language constructs, many of which (including the full range of arithmetic operators, as well as struct and union data types) cannot yet be generated by C4. Meanwhile, the LLVM test suite features a large set of stand-alone C and C++ programs and microbenchmarks, created by hand over many years with the express aim of providing diverse coverage. Finally, C4 focuses on exercising specific compiler features that occupy a small amount of each compiler's surface area.

[¶]We had to exclude the X-Ray benchmark, which did not compile successfully with the coverage-instrumented compiler. However, we searched the source code for this benchmark and ascertained that it does not feature atomic operations.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

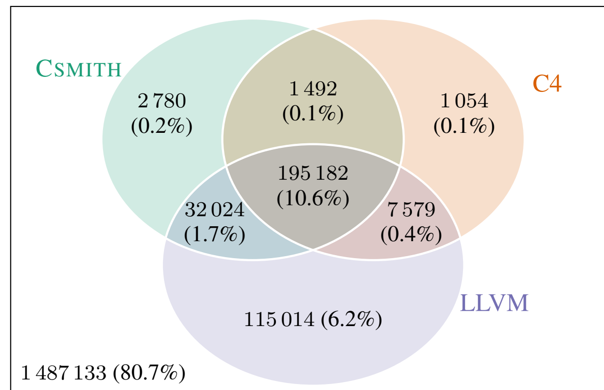

Figure 5. Intersection relationships between line coverage achieved for CSMITH, C4, and the LLVM test suite. All percentages to 3 figures.

	All	`-O1` only	∈ CSMITH	∈ LLVM
`CGAtomic`	316	0	2	141
`AtomicOrdering(.h)`	11	2	6	11
`AtomicExpandPass`	169	14	18	106
`InstCombineAtomicRMW`	57	57	0	30

Table III. For files containing 'atomic' in their name, C4 coverage: at all optimisations; at `-O1` but not `-O0`; also achieved by CSMITH; and also achieved by the LLVM test suite. No new lines were covered at `-O2` or `-O3`, or against MEMALLOY.

The number outside the Venn diagram indicates the number of lines in the compiler code base that were not covered by any of these corpora. It is unsurprising that this number is large because LLVM contains a lot of supporting infrastructure, not all of which is covered during compilation, as well as back-ends for many CPU and GPU architectures, where our experiments only exercise x86.

As we did not have prior knowledge of which parts of the LLVM codebase are related to atomics, to understand how much of the differences between coverage captured by C4 and the other corpora we investigated are related to atomic concurrency, we used heuristics to sample the coverage data. The first, and most coarse-grain, was to consider files whose names contain the word 'atomic', assuming that these files contain only code relating to atomics.[‖] The other two heuristics involve searching through the remaining files for lines containing one or more case-insensitive 'atomic keywords': 'atomic', 'RMW', 'memorder', 'memory_order', and 'cmpxchg'.

Table III shows the difference in lines covered by C4 against CSMITH and the LLVM test suite, as well as between itself at `-O1` and at `-O0`, on the 'atomic files' defined above. Much of this coverage is absent from the LLVM test suite: for instance, over half of the coverage on `InstCombineAtomicRMW` (an optimisation pass).

For files not included in the above table, we performed a line-based analysis. For each file, we generated 'diff' files in which covered lines had an arbitrary prefix and three lines of context on either side. We used `ripgrep` to search these for atomic keywords, requiring the prefix on the line (to select covered lines *directly* containing a keyword). As this may be an under-approximation, we also searched without the prefix requirement, but did not find interesting instances of extra coverage.

Table IV shows that there are several files in which both CSMITH and the LLVM test suite are missing the same number of lines as compared to C4. As CSMITH does not generate atomics, this is a possible heuristic for finding general atomics-related code not covered by the test suite, and

[‖]We exclude `AMDGPUAtomicOptimizer.cpp` from the sample, as we did not compile against an AMD GPU, and the only lines covered concerned initialisation.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

File	Type	Covered lines
`CGBuiltin`	Clang codegen	11
`CGExpr`	Clang codegen	2
`CodeGenFunction`	LLVM codegen	2
`CodeGenTypes`	LLVM codegen	4
`DeclSpec`	Clang semantics	1
`ExprConstant`	Clang AST	7
`InstCombineSelect`	LLVM transform	4
`LoopStrengthReduce`	LLVM transform	2
`ParseDecl`	Clang parsing	7
`SemaExpr`	Clang semantics	5
`SemaType`	Clang semantics	11
`ValueTracking`	LLVM analysis	4

Table IV. Files where, when considering lines containing atomic keywords, the number of lines covered by C4 but not by CSMITH equals the number covered by C4 but not by the LLVM test suite.

so potential areas for extension on said tests. We note that, for instance, the missing coverage in `CGBuiltin` concerns Clang-level code generation for C11 fences. Further, in `CGAtomic`, this approach revealed that the LLVM test suite misses code related to atomic fetches, loads, and stores that C4 reaches.

Our answer to RQ3 is that there is evidence of a small but distinctive region of code that C4 reaches that neither CSMITH nor the LLVM test suite hit. Some of this region is clearly related to atomic actions—for instance, the 27 lines in `InstCombineAtomicRMW` hit only by C4.

RQ4: Can C4 hit concurrency-related compiler optimisations? Table III shows that two significant 'atomic' files get coverage at `-O1`: one in `AtomicExpandPass` and one in `InstCombineAtomicRMW` (with this file not being reached when optimisations are disabled).

In both files, the change between `-O1` and `-O0` includes enabling code to handle 'idempotent' read-modify-writes: fetches that do not modify their target location. Both files contain similar tables of what constitutes idempotence: for example, `atomic_fetch_sub(x, 0)` is idempotent, as is `atomic_fetch_and(x, -1)` in two's-complement arithmetic. We cover both of these tables, except for where, in `InstCombineAtomicRMW`, the table includes fetch operators not available in C11. At the time of the experiment, C4 had support for specifically generating idempotent fetch actions over addition, subtraction, and bitwise OR and XOR; the rule for bitwise AND was covered by chance, but has since been added to C4.

One interesting optimisation we *did* hit, likely by chance, occurs in `InstCombineCompares`: a pass reduces equalities between the expected value and old value of the object of a strong compare-exchange to its returned 'success' Boolean. While C4 generates strong compare-exchanges, it does so in limited scenarios, and the situation in which C4 generates such comparisons is unclear.

We saw two files where `-O3` enabled coverage in atomic lines; both received one hit. The hit in `X86ISelLowering` enables an X86-specific set of optimisations over fetch actions. The hit in `InstCombineSelect` involve an LLVM IR optimisation targeting situations similar to those of the `InstCombineCompares` optimisation, but where we are selecting between the old and expected value using the success Boolean; such selections simplify to the expected value.

To answer RQ4, we have seen evidence that C4 is exercising optimisations involving atomic fetches and compare-exchanges, but few of these arise entirely through specific targeting from C4.

4.3. Mutation coverage

Line coverage shows that C4 covers a good amount of concurrency-related code in LLVM, but does not prove that C4 can detect bugs arising from that code. The uncertain prevalence of concurrency bugs in compilers is an issue here; as such, while we showcase some bugs found during live

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

ID	n in	Description	Justification	Area of compiler
LTn	1..3	Leading fence #n becomes trailing	[69, §5.5.1]	AtomicExpandPass
TLn	1..3	Trailing fence #n becomes leading	[69, §5.5.1]	AtomicExpandPass
SLT		Swap leading fence #1 with trailing fence #1	[69, §5.5.1]	AtomicExpandPass
DDn	1..6	Drop DMB fence #n (Arm only)	[69, §5.5.2]	ARMISelLowering
DSn	1..4	Drop sync fence #n (PPC only)	[69, §5.5.2]	PPCISelLowering

Table V. Operators used in our mutation experiment.

C4 experiments in section 4.4, we discuss here a systematic experiment seeking *strong mutation coverage* [68].

In this experiment, we used C4 to test a mutated variant of the LLVM 11 compiler. We investigated three research questions:

RQ5 Can C4 kill mutants representative of concurrency bugs?

RQ6 Does fuzzing significantly raise the chance of kills?

RQ7 Do killed mutants reflect the nondeterministic nature of concurrency bugs?

RQ5: Can C4 kill mutants representative of concurrency bugs? To answer RQ5, we implemented a small set of bespoke mutation operators, shown in table V. Each manipulates the emission of fences during part of LLVM's 'atomic expand pass' in which strong atomic actions lower into fenced weak actions.

The first three operators permute *leading* and *trailing* fences, which are inserted before load and after store events, respectively; the third operator specifically swaps a matched pair of leading and trailing fences used to 'bracket' certain atomic actions. The last two operators inhibit the insertion of architecture-specific fences: the Arm 'data memory barrier' (DMB) instructions and POWER 'sync' (and 'lwsync' and 'cfence') instructions, respectively. None of these operators are effective on x86-64, which has a stronger memory model and does not surround events with fences in this manner.

We justify this focus by reference to existing compiler concurrency bugs. For example, a prototype OpenCL compiler for AMD graphics processors inserted a fence-like instruction *after* a load when it should have come *before* it [69, §5.5.1]; this inspired our `SLT` operator. The same compiler elsewhere omitted a different fence-like instruction in its implementation of RMW operations [69, §5.5.2]; this inspired our `DD` and `DS` operators.

Each operator conceptually replaces an expression `e` with a construct `Mutant==N ? e2 : e`, where `e2` is known not to be equivalent to `e`. By connecting `Mutant` to an environment variable governed by C4, we can modify the mutation during testing and need not recompile the mutated compiler separately for each mutant. This saves considerable time, especially on slower targets such as Arm8.

We devised and applied each mutation operator manually, directly editing the LLVM code. We did so because of the bespoke nature of our runtime mutation strategy. We made a best effort to apply each operator systematically, with every known valid operator target giving rise to a mutant.

With these operators applied, we ran C4 in two 24-hour sessions: one with fuzzing disabled, and one with fuzzing enabled. In both cases, we ran separate loops for the Arm8 and PPC64 machine in which we iterated over the mutants relevant to each architecture. These loops proceeded as described in section 3.4, except that we limited the Arm8 machine to considering only `-march=armv7-a`; this is because Arm version 8 does not use fence lowering.

In each loop, we moved onto the next mutant every 10 minutes, or as soon as we detected a kill (a miscompilation in which the mutant was reached at least once). In the latter case, we removed the

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

ID	Fuzzing off (∗=kill)			Fuzzing on (†=kill)		
	Test cases	Hits	Kills	Test cases	Hits	Kills
Arm8 (in Arm7 mode)						
∗†LT1	140	437	1	1 040	31 646	2
LT2	8 080	0	0	5 000	0	0
∗†LT3	100	227	1	120	331	1
∗†TL1	80	199	1	40	596	1
∗†TL2	400	945	1	540	1 691	3
TL3	8 300	20 486	0	200	604	0
∗†SLT	140	910	1	20	886	3
†DD1	8 540	7 106	0	420	1 834	1
∗†DD2	80	89	1	60	198	1
†DD3	9 780	0	0	4 360	10 834	1
∗†DD4	40	50	1	80	1 260	1
∗†DD5	60	123	1	20	210	2
DD6	8 980	0	0	4 860	0	0
Total	44 720	30 572	8	16 760	50 090	16
PPC64						
∗†LT1	60	263	1	180	4 169	2
LT2	20 700	0	0	2 620	0	0
LT3	20 400	0	0	6 760	0	0
∗†TL1	160	742	1	320	9 281	1
TL2	20 400	0	0	6 620	0	0
TL3	20 420	0	0	6 680	0	0
∗†SLT	280	2 494	1	180	8 510	1
∗†DS1	240	499	1	80	766	1
∗†DS2	200	238	1	60	447	1
∗†DS3	540	866	1	200	2 978	2
∗†DS4	3 180	2 828	1	1 660	15 822	8
Total	86 580	7 930	7	25 360	41 973	16

Table VI. Results of the mutation experiment.

mutant from further consideration. After trying every applicable mutant once, we iterated over the mutants not yet killed.[**]

Table VI shows the experiment results. For each architecture and mutant ID, we record the number of test cases we generated and compiled with that mutant enabled; the number of mutant hits (`Mutant==N` was true), and the number of mutant kills. In addition to the tabulated, mutants `LT1` and `SLT` induced compiler crashes on PPC64 (40 and 55 times respectively) in the fuzzed experiment.

Our answer to RQ5 is that these data show that C4 was able to kill a large number of mutants, especially on Arm7 where 8 out of 13 mutants (and 5 out of 7 architecture-independent mutants) were killed.

RQ6: Does fuzzing significantly raise the chance of kills? To answer RQ6, we consult the differences between the 'fuzz-on' and 'fuzz-off' runs. While both approaches killed the majority

[**]Technical issues in the 'fuzz-on' run meant we did not consider PPC64's `LT2` and `DS4` mutants; the results for these are from a previous experiment which had the same parameters, but did *not* remove killed mutants from consideration.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Machine	Fuzz	Min	Mean	Max	Median
Arm8	Off	1	1 012	4 707	7
Arm8	On	1	139	978	4
PPC64	Off	1	22	124	2
PPC64	On	1	5 254	31 528	21.5

Table VII. Killing state metrics for each machine and run.

	x86-64	POWER9	Arm8
GCC	*Daily*☣, 10 snapshot, 7.4.0, 4.9.0	`8949b98`, 10 snapshot, 4.9.4	`28290cb`, 10 snapshot, 4.9.0☣
LLVM	*Daily*, `ea9166b`, 9.0.1, 6.0.0, 3.6	`52ba4fa`, `ea9166b`, 9.0.1, 3.6	`6733b25`, `ea9166b`, 9.0.1, 7.0.1, 3.6
Intel C Compiler	19.1.2.254	–	–
IBM XL	–	16.1.1.0001☣	–

Table VIII. Compiler versions explored in our bug-finding runs; those with ☣ exhibited bugs. Monospace text denotes *git* revisions. 'Daily' builds were recompiled from *git* on a daily basis, and cover many revisions.

of the mutants, 'fuzz-on' killed two Arm8 mutants (`DD1` and `DD3`) that 'fuzz-off' did not; the latter was also never hit within 9 780 subjects. While not a large increase, the fact that one mutant went from zero hits to one kill suggests that a change in the detectable bug surface did occur.

While the numbers of subjects required to kill each mutant varied between the runs, with 'fuzz-on' seemingly finding some mutants easier to kill than 'fuzz-off' and vice versa, we find it hard to draw inferences from this as the numbers are susceptible to random noise (mutant kills being dependent on nondeterministic behaviour).

While fuzzing did make some mutants visible for killing, the data both in terms of subjects required to kill mutants and also the number of mutants switched from unkilled to killed is not strong enough to draw definitive conclusions about the use of C4 fuzzing in killing these types of mutant. Our answer to RQ6 is that we did *not* see a significant rise, but that we did see some evidence that there was *a* rise.

RQ7: Do killed mutants reflect the nondeterministic nature of concurrency bugs? To answer RQ7, we consult the logs recorded by LITMUS for the test that killed each mutant. As LITMUS records the number of times each final state was observed, we can determine whether 'killing states' (unwanted states that triggered a mutant kill) occurred nondeterministically: that is, the sum total of observations for each such state in its test case is less than the total number of times LITMUS ran that test case (fixed at 1 000 000).

Table VII reports aggregate metrics for these observation totals. As the highest total killing-state occurrence count was 31 528 (3.1% of states for its test case), and the median and mean were much lower, we have confidence that these kills occurred nondeterministically. Our answer to RQ7 is yes.

4.4. *Finding bugs with C4*

We have used C4 to test real-world compilers (listed in table VIII). We discovered four bugs: two historical and concurrency-related, two recent but not concurrency-related.

While we have tested each listed compiler at some point, we have largely focused on recent versions of the leading open-source compilers GCC and LLVM. This simplified the building, patching, and instrumentation of them for experimentation; further, as Yang et al. [18] point out, open-source compiler projects are more amenable to bug reporting and tracking. This focus also allowed us to explore the assertion that 'given enough eyeballs, all bugs are shallow' [70]. Between

Copyright © 0000 John Wiley & Sons, Ltd.

Listing 2: Hand-reduced test case for the conditional load bug.

```
1  struct ctx { atomic_int *x; int y; };
2  int P(void *v) {
3     struct ctx *a = (struct ctx *)v;
4     a->y == -1 || atomic_load(a->x);
5     return 0;
6  }
7  int main() {
8     atomic_int x; thrd_t thread; struct ctx ctx;
9     ctx.x = &x;
10    atomic_store_explicit(ctx.x, 0, memory_order_relaxed);
11    ctx.y = -1;
12    thrd_create(&thread, P, &ctx);
13    thrd_join(thread, NULL);
14    return 0;
15 }
```

Listing 3: Assembly output for the conditional load bug.

```
1  ldr   r3, [r0, #4] @ r3 <- a->y
2  cmn   r3, #1      @ compare r3 with -1
3  ldrne r3, [r0]    @ r3 <- a->x (ONLY IF r3 != -1)
4  lda   r3, [r3]    @ r3 <- *r3 (bug: missing conditional)
```

2020-10-19 and 2020-10-31, we ran tests only on the most recent builds of GCC and LLVM shown in table VIII; on x86-64, we also built the latest Git revisions of both compilers each day.

We found a wrong-code bug, and an erroneous rejection of valid code, in the historical 4.9 version of GCC. These issues were already fixed in newer versions by the time that we started experiments, but we were not aware of their existence until we discovered them using C4; this said, their discovery shows that C4 can detect and defend against similar bugs in future.

GCC4: crash on conditional load While developing C4, we discovered a historic bug in the way in which GCC 4.9 handles atomic loads in conditional expressions on Arm8. The bug exhibits as a bus error on executing the compiled code. Listing 2 gives a minimal working example of this bug, derived by manually reducing an instance of the *if.tautology* fuzzer action taken from a C4 test case.

The bug occurs in the redundant conditional at line 4, which `gcc-4.9 -O1 -march=armv8-a -std=c11 -pthread` compiles to the assembly in listing 3.

The last two instructions, which implement the `atomic_load_explicit`, should only execute when the first two instructions (implementing `a->y == -1`) yield a not-equal comparison. By mistake, GCC emits only the first such instruction as conditional on this result. When `a->y == -1` is true, the last instruction will fire, try to load memory at address -1 into `r3`, and cause a bus error.

Use of *Compiler Explorer* (minimal working example available at `https://godbolt.org/z/qXtsxy`) suggests that this bug was fixed between GCC 4.9 and 5.4 (which substitutes **`ldane`** for **`lda`**), and we could not reproduce it in GCC 8 and 9 (which use a **`beq`** branch).

GCC4: erroneous outcome on conditional store GCC 4.9 miscompiles one of the basic test cases produced by MEMALLOY, as seen in listing 4. When placed in a LITMUS harness and compiled as `-O1 -march=armv8-a -std=gnu11 -pthread`, this test case can return final states where `y` assumes neither 1 nor its initial value.

The problem lies in a miscompilation of `P1`, which we see in listing 5. 32-bit Arm8 supports instructions predicated on the results of previous comparisons, and GCC has attempted to use that support here. The last instruction should be **`stleq`**, but is **`stl`**, triggering even when `r0 == 1` is

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Listing 4: Test case for the conditional store bug.

```
1 void P0(atomic_int *y, int *x) {
2   int r0 = atomic_load_explicit(y, memory_order_acquire);
3   if (r0 == 1) *x = 1;
4 }
5 void P1(atomic_int *y, int *x) {
6   int r0 = *x;
7   if (r0 == 1)
8     atomic_store_explicit(y, 1, memory_order_release);
9 }
```

Listing 5: Assembly for thread P1 of listing 4.

```
1 ldr   r3, [r3, ip] @ r3 <- *x
2 cmp   r3, #1      @ compare r3 with 1
3 addeq r2, r2, ip @  IF equal, set r2 to address of y,
4 moveq r7, #1      @  and load 1 into r7, ready for stl
5 stl   r7, [r2]    @ BUG: unconditionally store r7 to y
```

Listing 6: C-Reduced test case for the XL goto bug.

```
1 int a;
2 void b(void) {
3   goto c;
4   if (!(a && 0)); else c: ;
5 }
```

false; this stores the (garbage) contents of `r7` to the (garbage) address pointed to by `r2`. Modern GCC either emits **`stleq`**, or uses branching instead of conditional instructions.

GCC4: unnecessary memory model safety check We noticed that GCC4 rejected test cases containing `atomic_exchange_explicit` calls with the 'consume' memory order. This pairing, while exotic (and subject to potential changes or deprecations in further standard revisions[††]), is valid in C11, but old versions of GCC contained an explicit check forbidding it. While we have not been able to find a fault report about this behaviour, the check was removed in 2015.[‡‡]

While we did not find novel concurrency bugs with C4, we have found novel internal compiler errors (ICEs) in both GCC and IBM XL relating to unusual *sequential* control flows. We found this surprising, as we did not design C4 with this purpose; this points to a more general role for C4 in defending against compiler bugs.

IBM XL: internal compilation error While testing IBM XL C/C++ 16.1.1 on the POWER9 machine, we noticed many internal compiler errors traceable to the *dead.insert.goto* fuzzer action. As these were compiler crashes and not semantic errors, we were able to apply C-Reduce [71] to one test case, which, with some manual clean-up, led to the stimulus in listing 6. We reported this bug on the IBM Power Community forums and understand that the bug was reproduced by IBM engineers, with a fix forthcoming (per correspondence in `https://bit.ly/XLcompilerbug`). We note that CSMITH was also able to find a stimulus that reduces to a similar example (substituting **`if`** `(a || 1)`) in under one hour.

[††] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0371r0.html` and `http://eel.is/c++draft/atomics.order\#1.3`
[‡‡] `https://gcc.gnu.org/g:086f4e333c3d3b0e714a68a112cc582315a6344`

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Listing 7: Stimulus for GCC bug 97501, based on a C-Reduce reduction of a C4 test case. The stimulus exhibits the bug without the use of the variable 'c', but then has live undefined behaviour.

```
1 static int c = 0;
2 int main() {
3   int b = 0;
4   if (c) { for (;; b--) do b++; while (b); }
5 }
```

GCC snapshot: internal compilation error We detected an internal compiler error on the x86-64 machine's daily snapshot of GCC, occurring in range-verifying code in that compiler's tree optimisation phase. A pass of C-Reduce reduced the input test case to a pair of nested, overflowing loops – in the original test case, these correspond to loops introduced into dead-code – and, after adding some support code to make sure the undefined behaviour in the loops could not appear at run-time, we submitted the ICE as GCC bug 97501 (`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97501`). This bug has been fixed as of revision `5d53ec27` (`https://gcc.gnu.org/g:5d53ec27015b916640171e891870adf2c6fdfd4c`).

Unlike the XL bug, we were unable to detect the bug with CSMITH after running tests for a 24-hour period. We expect that this is related to the undefined nature of the code fragment triggering the bug: CSMITH actively avoids generating code with undefined behaviour, and does not discriminate between whether the code fragments it generates are reachable or not. It would thus appear that CSMITH cannot trigger compiler bugs that rely on unreachable code that would – if made reachable – lead to undefined behaviour.

4.5. Threats to validity

We note possible threats to validity in our experimental set-ups.

- When running on a real machine, its hardware may yield stronger memory behaviour than guaranteed by its architecture. This can mask compiler bugs (causing false negatives). Simulating the compiled programs (as discussed in section 5) could mitigate this threat.
- Hardware may contain bugs with respect to its architecture; we may mistake these for compiler bugs (false positives). These are rare, but not unheard of [72].
- While we believe the combination of (static) input fuzzing and (dynamic) LITMUS based stress-testing to be effective in stimulating concurrency-related compiler bugs, we cannot guarantee that it is strong enough to find *all* such bugs.
- We have thoroughly tested the fuzzer actions in C4, providing confidence that they observationally refine input tests, but bugs in C4 may remain, which could lead to false negatives or positives.
- Our coverage analysis was performed against LLVM only. The answers to our coverage-related research questions are not guaranteed to generalise to other compilers, such as GCC. We could readily investigate the questions for other open-source compilers.

5. CONCLUSIONS AND FUTURE WORK

We have presented the design, implementation and evaluation of C4, a tool for checking whether C compilers compile concurrency in accordance with the expected C11 semantics. Our coverage experiments show that C4 complements the coverage of other methods (CSMITH and the LLVM test suite), exercises some interesting code relating to atomic-action concurrency, and can detect fence insertion failures representative of real compiler bugs. C4 *did* find concurrency bugs, but only historical ones (sections 4.4 and 4.4). As we had no prior knowledge of these bugs, we claim that C4 would have helped find them had it been available; further, it can help defend against future bugs of a similar nature. Our results point to a natural C4 use case: assistance for compiler developers

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

working on compilation schemes for concurrency, or on improving concurrency optimisation passes. For instance, an LLVM developer producing a back-end for a new target could use the coverage of fence emission shown in section 4.3 to check that the right fences are produced in the right places, or to check for mistakes when implementing more aggressive behaviour for existing targets.

One avenue for future work is to extend the fuzzer with more actions, to help us tax the compilers under test, increasing the chance of finding bugs. Actions to consider include: more coverage of read-modify-write atomics (such as exchanges, fetches, and compare-exchanges); introducing **`volatile`** qualifiers for global variables; and inserting composite types, such as **`struct`** and **`union`**. Some of these actions would require bypassing syntactic limitations of LITMUS. Another line of work is to improve the usability of C4 via better test-case reduction. C4 supports basic reduction by trimming traces of bug-inducing fuzzer sessions, but more sophisticated reduction would aid the submission of C4-derived bug reports.

On the experimental front, while we targeted GCC, LLVM, IBM XL, and the Intel C Compiler (ICC) in our experiments, we mainly focused on the first two, due to their portability and free availability. Future work could involve more experimentation on IBM XL and ICC, as well as investigating other compilers; e.g. the Microsoft C compiler, once it supports C11 atomics. While we aimed for diversity in targeting x86-64, 32-bit Arm, and PPC64, we could test C4 on other architectures such as AArch64. The bugs we found in GCC 4.9 `armv8-a` suggest focusing on architectures without mature compiler support for atomics to aid early discovery of bugs. This could include recent revisions of existing architectures (e.g. `armv8.2`), or novel systems such as RISC-V.

While the workflow in this article depends on LITMUS, we have tried to make C4 agnostic to its underlying test-running backend, and considering other backends is a line of future work. We have had some initial success with RMEM [73], a high-performance simulator for AArch64, RISC-V, and other architectures: for instance, when we experimented with using the stronger RC11 memory model of Lahav et al. [74] as the input to our test-case generator, RMEM identified as a bug the 'load buffering' test that RC11 forbids, but C11 and AArch64 permit.

REFERENCES

1. Kell S. Some were meant for C: The endurance of an unmanageable language. *ACM SIGPLAN Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, ACM: New York, NY, USA, 2017; 229–245, doi:10.1145/3133850.3133867.
2. Sutter H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal* 2005; **30**(3):202–210.
3. Dodds M, Haas A, Kirsch CM. A scalable, correct time-stamped stack. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, ACM, 2015; 233–246, doi:10.1145/2676726.2676963.
4. Lê NM, Pop A, Cohen A, Nardelli FZ. Correct and efficient work-stealing for weak memory models. *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, ACM, 2013; 69–80, doi:10.1145/2442516.2442524.
5. 9899:2011 I. Information technology Programming languages C. *Standard*, International Organization for Standardization, Geneva, CH Dec 2011.
6. Thompson K. Reflections on trusting trust. *Commun. ACM* 1984; **27**(8):761–763, doi:10.1145/358198.358210.
7. Manerkar YA, Trippel C, Lustig D, Pellauer M, Martonosi M. Counterexamples and proof loophole for the C/C++ to POWER and armv7 trailing-sync compiler mappings. *CoRR* 2016; **abs/1611.01507**. URL `http://arxiv.org/abs/1611.01507`.
8. Chen J, Patra J, Pradel M, Xiong Y, Zhang H, Hao D, Zhang L. A survey of compiler testing. *ACM Comput. Surv.* 2020; **53**(1):4:1–4:36.
9. Cuoq P, Monate B, Pacalet A, Prevosto V, Regehr J, Yakobowski B, Yang X. Testing static analyzers with randomly generated programs. *NASA Formal Methods*, *Lecture Notes in Computer Science*, vol. 7226, Springer, 2012; 120–125.
10. Tolksdorf S, Lehmann D, Pradel M. Interactive metamorphic testing of debuggers. *ISSTA*, ACM, 2019; 273–283.
11. Lehmann D, Pradel M. Feedback-directed differential testing of interactive debuggers. *ESEC/SIGSOFT FSE*, ACM, 2018; 610–620.
12. Winterer D, Zhang C, Su Z. Validating SMT solvers via semantic fusion. *PLDI*, ACM, 2020; 718–730.
13. Chen J, Hu W, Hao D, Xiong Y, Zhang H, Zhang L, Xie B. An empirical comparison of compiler testing techniques. *ICSE*, ACM, 2016; 180–190.
14. Sun C, Le V, Zhang Q, Su Z. Toward understanding compiler bugs in GCC and LLVM. *ISSTA*, ACM, 2016; 294–305.
15. Marcozzi M, Tang Q, Donaldson AF, Cadar C. Compiler fuzzing: how much does it matter? *Proc. ACM Program. Lang.* 2019; **3**(OOPSLA):155:1–155:29.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

16. McKeeman WM. Differential testing for software. *Digit. Tech. J.* 1998; **10**(1):100–107. URL `http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf`.
17. Groce A, Holzmann GJ, Joshi R. Randomized differential testing as a prelude to formal verification. *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, IEEE Computer Society, 2007; 621–631.
18. Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, 2011; 283–294, doi:10.1145/1993498.1993532.
19. Chen TY, Cheung SC, Yiu SM. Metamorphic testing: A new approach for generating next test cases. *Technical Report HKUST-CS98-01*, The Hong Kong University of Science and Technology 1998.
20. Segura S, Fraser G, Sánchez AB, Cortés AR. A survey on metamorphic testing. *IEEE Trans. Software Eng.* 2016; **42**(9):805–824.
21. Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2014, doi:10.1145/2594291.2594334.
22. Chen Y, Su T, Sun C, Su Z, Zhao J. Coverage-directed differential testing of JVM implementations. *PLDI*, ACM, 2016; 85–99.
23. Le V, Sun C, Su Z. Finding deep compiler bugs via guided stochastic program mutation. *OOPSLA*, ACM, 2015; 386–399.
24. Donaldson AF, Evrard H, Lascu A, Thomson P. Automated testing of graphics shader compilers. *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2017, doi:10.1145/3133917.
25. Donaldson AF, Lascu A. Metamorphic testing for (graphics) compilers. *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*, ACM, 2016; 44–47, doi:10.1145/2896971.2896978. URL `https://doi.org/10.1145/2896971.2896978`.
26. Donaldson AF, Thomson P, Teliman V, Milizia S, Maselco AP, Karpinski A. Test-case reduction and deduplication almost for free with transformation-based compiler testing. *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Freund SN, Yahav E (eds.), ACM, 2021; 1017–1032, doi:10.1145/3453483.3454092. URL `https://doi.org/10.1145/3453483.3454092`.
27. Tao Q, Wu W, Zhao C, Shen W. An automatic testing approach for compiler based on metamorphic testing technique. *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, Han J, Thu TD (eds.), IEEE Computer Society, 2010; 270–279, doi:10.1109/APSEC.2010.39. URL `https://doi.org/10.1109/APSEC.2010.39`.
28. Lopes NP, Menendez D, Nagarakatte S, Regehr J. Practical verification of peephole optimizations with Alive. *Commun. ACM* 2018; **61**(2):84–91, doi:10.1145/3166064.
29. Kang J, Kim Y, Song Y, Lee J, Park S, Shin MD, Kim Y, Cho S, Choi J, Hur CK, *et al.*. Crellvm: Verified credible compilation for LLVM. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2018; 631–645, doi:10.1145/3192366.3192377.
30. Ševčík J, Vafeiadis V, Zappa Nardelli F, Jagannathan S, Sewell P. Relaxed-memory concurrency and verified compilation. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, ACM, 2011, doi:10.1145/1926385.1926393.
31. Beringer L, Stewart G, Dockins R, Appel AW. Verified compilation for shared-memory C. *Europ. Symp. on Programming Languages and Systems (ESOP)*, Springer-Verlag, 2014; 107–127, doi:10.1007/978-3-642-54833-8_7.
32. Leroy X. Formal verification of a realistic compiler. *Commun. ACM* 2009; **52**(7):107–115, doi:10.1145/1538788.1538814.
33. Lee SH, Cho M, Podkopaev A, Chakraborty S, Hur CK, Lahav O, Vafeiadis V. Promising 2.0: Global optimizations in relaxed memory concurrency. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2020. To appear.
34. Paviotti M, Cooksey S, Paradis A, Wright D, Owens S, Batty M. Modular relaxed dependencies in weak memory concurrency. *Europ. Symp. on Programming (ESOP)*, 2020; 599–625, doi:10.1007/978-3-030-44914-8_22.
35. Lidbury C, Lascu A, Chong N, Donaldson AF. Many-core compiler fuzzing. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2015, doi:10.1145/2737924.2737986.
36. Jiang B, Wang X, Chan W, Tse T, Li N, Yin Y, Zhang Z. CUDAsmith: A fuzzer for CUDA compilers. *Computer Software and Applications Conference (COMPSAC)*, IEEE, 2020. To appear.
37. Morisset R, Pawan P, Zappa Nardelli F. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2013, doi:10.1145/2491956.2491967.
38. Chakraborty S, Vafeiadis V. Validating optimizations of concurrent C/C++ programs. *IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, 2016, doi:10.1145/2854038.2854051.
39. Batty M, Owens S, Sarkar S, Sewell P, Weber T. Mathematizing C++ concurrency. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, ACM, 2011, doi:10.1145/1926385.1926394.
40. Batty M, Memarian K, Owens S, Sarkar S, Sewell P. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, ACM, 2012, doi:10.1145/2103621.2103717.
41. Wickerson J, Batty M, Sorensen T, Constantinides GA. Automatically comparing memory consistency models. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, ACM, 2017, doi:10.1145/3009837.3009838.
42. Trippel C, Manerkar YA, Lustig D, Pellauer M, Martonosi M. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, doi:10.1145/3037697.3037719.

Copyright © 0000 John Wiley & Sons, Ltd.

43. Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.* 2015; **41**(5):507–525.
44. Alglave J, Maranget L, Tautschnig M. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. on Programming Languages and Systems (TOPLAS)* 2014; **36**(2), doi:10.1145/2627752.
45. Zhu H. Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods. *2015 Second International Conference on Trustworthy Systems and Their Applications*, 2015; 8–15, doi:10.1109/TSA.2015.13.
46. Alglave J, Maranget L, Sarkar S, Sewell P. Litmus: Running tests against hardware. *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2011, doi:10.1007/978-3-642-19835-9_5.
47. Lustig D, Wright A, Papakonstantinou A, Giroux O. Automated synthesis of comprehensive memory model litmus test suites. *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, doi:10.1145/3037697.3037723.
48. Danglot B, Vera-Perez O, Yu Z, Zaidman A, Monperrus M, Baudry B. A snowballing literature study on test amplification. *Journal of Systems and Software* 2019; **157**:110 398, doi:10.1016/j.jss.2019.110398.
49. LLVM Project. Test-suite guide 2020. URL `https://llvm.org/docs/TestSuiteGuide.html`.
50. Kochhar PS, Thung F, Lo D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE Computer Society, 2015; 560–564.
51. Windsor M, Donaldson AF, Wickerson J. C4: the C compiler concurrency checker. *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cadar C, Zhang X (eds.), ACM, 2021; 670–673, doi:10.1145/3460319.3469079. URL `https://doi.org/10.1145/3460319.3469079`.
52. Lascu A, Windsor M, Donaldson AF, Grosser T, Wickerson J. Dreaming up metamorphic relations: Experiences from three fuzzer tools. *6th IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2021, Madrid, Spain, June 2, 2021*, IEEE, 2021; 61–68, doi:10.1109/MET52542.2021.00017. URL `https://doi.org/10.1109/MET52542.2021.00017`.
53. Gramoli V. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, Association for Computing Machinery: New York, NY, USA, 2015; 110, doi:10.1145/2688500.2688501. URL `https://doi.org/10.1145/2688500.2688501`.
54. Adve S, Gharachorloo K. Shared memory consistency models: a tutorial. *Computer* 1996; **29**(12):66–76, doi:10.1109/2.546611.
55. Batty M, Donaldson AF, Wickerson J. Overhauling SC atomics in C11 and opencl. *POPL*, ACM, 2016; 634–648.
56. Tao Q, Wu W, Zhao C, Shen W. An automatic testing approach for compiler based on metamorphic testing technique. *2010 Asia Pacific Software Engineering Conference*, 2010; 270–279, doi:10.1109/APSEC.2010.39.
57. Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 1990; **33**(12):32–44, doi:10.1145/96267.96279. URL `https://doi.org/10.1145/96267.96279`.
58. Zalewski M. American fuzzy lop (afl). *URL: http://lcamtuf. coredump. cx/afl* 2017; .
59. Serebryany K. Continuous fuzzing with libFuzzer and AddressSanitizer. *2016 IEEE Cybersecurity Development (SecDev)*, IEEE, 2016; 157–157.
60. Godefroid P, Levin MY, Molnar DA. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 2012; **55**(3):40–44, doi:10.1145/2093548.2093564. URL `https://doi.org/10.1145/2093548.2093564`.
61. Jackson D. *Software Abstractions*. second edn., MIT Press, 2012.
62. Chong N, Sorensen T, Wickerson J. The semantics of transactions and weak memory in x86, Power, ARM, and C++. *PLDI*, ACM, 2018; 211–225.
63. Raad A, Wickerson J, Neiger G, Vafeiadis V. Persistency semantics of the intel-x86 architecture. *Proc. ACM Program. Lang.* 2020; **4**(POPL):11:1–11:31.
64. Wang X, Zeldovich N, Kaashoek MF, Solar-Lezama A. Towards optimization-safe systems: analyzing the impact of undefined behavior. *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, ACM, 2013; 260–275.
65. Donaldson AF, Evrard H, Thomson P. Putting randomized compiler testing into production. *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP 2020)*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. To appear.
66. Klees G, Ruef A, Cooper B, Wei S, Hicks M. Evaluating fuzz testing. *CCS*, ACM, 2018; 2123–2138.
67. Herklotz Y, Wickerson J. Finding and understanding bugs in FPGA synthesis tools. *FPGA*, ACM, 2020; 277–287.
68. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.* 2011; **37**(5):649–678, doi:10.1109/TSE.2010.62. URL `https://doi.org/10.1109/TSE.2010.62`.
69. Wickerson J, Batty M, Beckmann BM, Donaldson AF. Remote-scope promotion: Clarified, rectified, and verified. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, ACM: New York, NY, USA, 2015; 731747, doi:10.1145/2814270.2814283.
70. Raymond E. The cathedral and the bazaar. *Knowledge, Technology & Policy* 1999; **12**(3):23–49, doi:10.1007/s12130-999-1026-0.
71. Regehr J, Chen Y, Cuoq P, Eide E, Ellison C, Yang X. Test-case reduction for C compiler bugs. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2012; 335–346, doi:10.1145/2254064.2254104.
72. Alglave J, Maranget L, Sarkar S, Sewell P. Fences in weak memory models. *Int. Conf. on Computer Aided Verification (CAV)*, Springer-Verlag, 2010; 258–272, doi:10.1007/978-3-642-14295-6_25.
73. Pulte C, Flur S, Deacon W, French J, Sarkar S, Sewell P. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.* 2018; **2**(POPL):19:1–19:29.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

74. Lahav O, Vafeiadis V, Kang J, Hur CK, Dreyer D. Repairing sequential consistency in C/C++11. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, ACM: New York, NY, USA, 2017; 618–632, doi: 10.1145/3062341.3062352.

Copyright © 0000 John Wiley & Sons, Ltd.