C4: The C Compiler Concurrency Checker

Matt Windsor
Department of Computer Science
University of York
York, United Kingdom
matt.windsor@york.ac.uk

Alastair F. Donaldson
Department of Computing
Imperial College London
London, United Kingdom
alastair.donaldson@imperial.ac.uk

John Wickerson
Department of Electrical and
Electronic Engineering
Imperial College London
London, United Kingdom
j.wickerson@imperial.ac.uk

ABSTRACT

The correct compilation of atomic-action concurrency is vital now that multicore processors are ubiquitous. Despite much recent work on automated compiler testing, little existing tooling can test how real-world compilers handle compilation of atomic-action code. We demonstrate C4, a tool for exploring the concurrency behaviour of real-world C compilers such as GCC and LLVM. C4 automates a workflow based on generating, fuzzing, and executing litmus tests. So far, C4 has found two new control-flow bugs in GCC and IBM XL, and reproduced two historic concurrency bugs in GCC 4.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Software testing and debugging**; *Concurrent programming structures*; • **Computing methodologies** → *Concurrent programming languages.*

KEYWORDS

atomic actions, C compilers, concurrency, fuzz testing

ACM Reference Format:
Matt Windsor, Alastair F. Donaldson, and John Wickerson. 2021. C4: The C Compiler Concurrency Checker. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3460319.3469079

1 INTRODUCTION

C is ubiquitous as a systems language, despite attempts to replace it [14]. Along with the widespread use of multicore processors in modern computing, this makes the presence of bugs in the way C compilers support concurrency primitives, such as atomic actions (a language feature that expose the ability to load, store, and otherwise modify memory in an atomic way) a key concern. Worse, such bugs may be hard to detect due to the inherent nondeterminism and architectural sensitivity of concurrency. We demonstrate C4, a tool for checking the behaviour of compilers against the C11 memory model, which describes ordering constraints on memory actions.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3469079

1.1 Problem Specification

The problem C4 addresses is the effective randomised testing of the compilation of atomic-action concurrency for C programs that use the C11 weak memory model [12].

As with any randomised compiler testing technique, C4 must handle the *oracle problem* — how to determine what constitutes a 'correct' behaviour of a system under test over a test input [3]. Unlike approaches that focus on sequential compilation, C4 cannot rely on *differential* testing to circumvent the oracle problem, i.e. comparing the output of a program after compilation via several different compilers, and identifying bugs via result mismatches.

First, a concurrent program is inherently nondeterministic, so that a concurrent test case may have multiple valid final states. Result mismatches may be due to this nondeterminism rather than due to compiler bugs. Second, this problem is exacerbated by weak memory: modern processors, by default, provide weak guarantees as to when atomic writes in one thread can be observed by atomic reads in another thread. This is because writes can be held in caches before propagating to main memory, reads can observe stale cache entries, and processors and compilers can move instructions around for efficiency. Weak memory increases potential nondeterminism by expanding the number of possible orders for actions, and adds differences in behaviour between processor architectures (e.g. x86 machines have a stronger memory model than Arm machines).

1.2 Prior Work

Compiler testing is an active research area (see [8] for a recent survey), but no method currently exists to check *automatically* that *concurrency* constructs (including atomics) are compiled correctly by *mainstream* compilers. Random testing has considered concurrency to only a limited degree in the context of OpenCL and CUDA compilers [13, 19]: the concurrent test-cases used in those works are deterministic by construction; our work involves testing compilers on more general concurrent code. Morriset et al. [21] have used random testing to check that GCC preserves C's concurrency semantics, but they cannot handle atomics. Chakraborty et al. [7] check whether LLVM transformations preserve C's concurrency semantics, but they do not check the entire compilation process. The CompCert verified C compiler [18] has been formally proven to handle some concurrency correctly [6, 24], but similar proofs about mainstream compilers remain infeasible; our work treats the compiler as an opaque box, and the concurrency semantics as a parameter, and hence will be able to keep up with evolving compilers and language standards [17, 22]. The correctness of several compiler *mappings* has been proven [4, 5] or automatically checked [23, 25], but these mappings are only an abstraction of the full compiler.

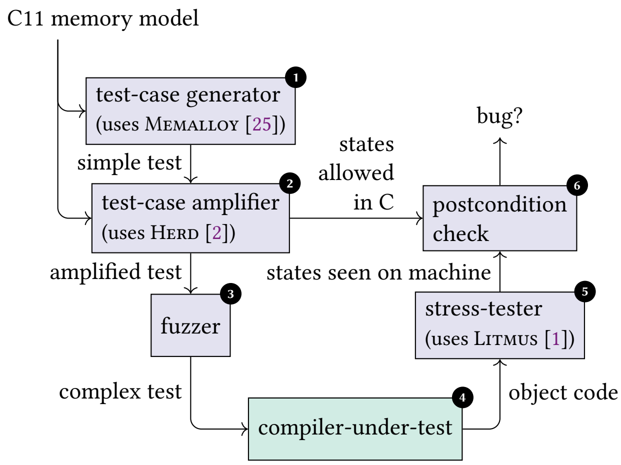

Figure 1: Our compiler-testing workflow.

1.3 Our Approach

Our approach to solving the problems is to automate the testing workflow shown in Figure 1. This workflow consists of these steps:

❶ We **generate** a small test-case containing a multi-threaded C program and a postcondition over its final states. Following Wickerson et al [25] and Lustig et al. [20], 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 **amplify** [10] the test-case: changing its postcondition 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 [2] to produce the exhaustive set of outcomes that *are* allowed by the memory model, setting the postcondition to require all outcomes to inhabit that set.

❸ We **fuzz** the test-case: applying mutations (such as dead-code introduction [19]) in the hope of coaxing the compiler-under-test into revealing bugs. In the spirit of EMI testing [16], 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, we may have found a compiler 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. [1], 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** that the amplified postcondition holds for all outcomes produced by step ❺; if not, they are disallowed by C11, and we may have found a compiler bug.

Our approach elegantly addresses the oracle problem: it uses exhaustive simulation to find all valid executions of a test-case small enough to simulate efficiently, then applies semantics-refining transformations to make the test-case sufficiently complicated to explore a good proportion of the compiler. It explores interesting interactions in weak-memory, atomic-action concurrency by appealing to the underlying test-case generator. By using the C11 memory model to generate and simulate test-cases, our testing campaigns can explore the relationship between behaviours of compiled test-cases and the expectations we have given their source code.

1.4 Our Tool — C4

C4 is a tool for checking the behaviour of compilers as they encounter C11 atomic-action concurrency. Its key components are:

- C4$_\mathrm{f}$, a *fuzzer* that adds random nontrivial control flow and redundant atomic actions to existing concurrent test-cases;
- C4$_\mathrm{t}$, a *tester* that runs unattended testing campaigns against multiple compilers on multiple machines.

C4 implements the approach in Figure 1 by leveraging existing technologies for exploring weak memory behaviour (based on the 'litmus test' format popularised by Alglave et al.), and implementing a scheme of program mutation similar to that of tools such as GRAPHICSFUZZ [11]. The novel contribution of our tool is to combine these approaches into a fully automated workflow specifically focused on exercising atomic-action compilation.

Target audience. We intend C4 to be useful to implementers of atomic-action concurrency in C compilers. Their work may include: adding a new architecture to an existing compiler, improving the optimisations performed by a compiler, safeguarding from regressions in existing functionality, or creating an entirely new C compiler.

Getting C4. C4 is free and open-source software, with source available at https://c4-project.github.io. We have also prepared a Docker image[1] that bundles C4, stock Debian GCC and LLVM compilers, and a sample corpus and initial configuration.

2 C4$_\mathrm{f}$: LITMUS TEST FUZZER

This section discusses C4$_\mathrm{f}$, the part of C4 that randomly transforms C litmus tests. It does so in a way that refines the observational semantics of the input test-case, with respect to its original variables.

2.1 Installation

C4$_\mathrm{f}$ is an OCaml program, and should work on most POSIX operating systems. It is not available on OPAM, but can be installed into an OPAM switch using `opam install .` in a source working copy.

2.2 Usage

C4$_\mathrm{f}$ contains two binaries: `c4f`, which performs fuzzing, and `c4f-c`, which provides helper functionality relating to C litmus tests. Both are self-describing: use `c4f help` and `c4f-c help` respectively.

To fuzz a C litmus test named `x.litmus`, use `c4f run x.litmus`. This chooses a random seed, and outputs the fuzzed test-case to standard output. Subsequent runs of `c4f` use different seeds, unless one is provided using `-seed`.

C4$_\mathrm{f}$ optionally accepts (argument `-config`) a file with overrides for parameters such as action weights and thread caps. See `c4f list-actions -v` and `c4f list-params -v` for information on what can be tweaked, and `c4f.conf.example` in the C4$_\mathrm{f}$ repository.

[1] https://hub.docker.com/r/captainhayashi/c4

Getting initial test-cases. $C4_f$ depends on having initial test-cases to mutate. Such test-cases (which can be generated once and reused for each testing campaign) should be in the C subset $C4_f$ understands, contain interesting concurrency patterns that would benefit from $C4_f$'s ability to insert unusual control flow, and have strong (ideally exhaustive) postconditions over allowed final states.

Our recommended approach to test-case generation is that mentioned in Figure 1: run Memalloy [25] over a representation of the C11 memory model to generate minimal test-cases with known possible disallowed states, then use Herd [2] to simulate those test-cases exhaustively, finding all states allowed by said model. We maintain a repository of test-cases generated in this way at https://github.com/c4-project/c4-corpora.

2.3 Implementation

The structure, and some features, of $C4_f$ take inspiration from tools such as GraphicsFuzz. The main unit of structure in $C4_f$ is *actions*—modules that perform a single transformation on a test-case, subject to a precondition over both the current test-case and the set of facts $C4_f$ holds over the test-case. Actions can accept randomised *payloads* as parameters.

Two runners select actions and apply them to the test-case. One randomises the number, choice, and payload details of actions; another replays a recorded action trace (useful for test-case reduction and regression testing).

The design and implementation of actions generally follows that of *metamorphic relations* in metamorphic testing tools. Lascu et al. [15] discusses the processes by which we chose actions for $C4_f$.

3 $C4_t$: AUTOMATED TEST RUNNER

This section discusses $C4_t$, the part of C4 that runs testing campaigns. $C4_t$ implements the overall workflow shown in Figure 1.

3.1 Installation

$C4_t$ is a cross-platform Go module containing several binaries. To install all of them into an existing Go setup, run `go install github.com/c4-project/c4t/cmd/...`.

3.2 Usage

The `c4t` binary is the main test runner. When properly configured, running `c4t` in a terminal will start a test campaign, opening a dashboard that monitors the progress (number of tests run, proportion of outcomes, current actions, and so on) across each configured machine. The campaign continues until a critical error occurs, the campaign reaches a deadline (configurable on the command line), or the user presses Ctrl-C in the terminal.

Configuration. $C4_t$ expects a configuration file listing the machines it can access, the execution backend to use (currently Litmus only), and the compilers on each machine. The `c4t-config` tool can generate basic configuration for the local machine, probing for existing compilers and the machine specification.

Remote machines. Some of our testing campaigns exercised compilers on an architecture (Arm) for which we only had a low-specification machine (a Raspberry Pi). Running the full testing cycle in Figure 1 on such machines harms throughput: only a small part of it is machine-dependent, and fuzzing in particular is CPU intensive. To remedy this, $C4_t$ supports multi-machine testing where one machine delegates compilation and execution to other machines but performs all other tasks. To do this, one puts a *machine node* binary, `c4t-mach`, in the `PATH` of the remote machine, and configures `c4t` appropriately. In this scenario, $C4_t$ uses `SFTP` to copy any C code to compile to the remote machine, runs the machine node over `SSH` to orchestrate compilation and execution, and retrieves the results over standard output.

Other binaries. $C4_t$ also comes with several helper binaries besides `c4t` and `c4t-mach`. Binaries exist that run one stage of the tester (`c4t-plan`, `c4t-perturb`, `c4t-fuzz`, `c4t-lift`, `c4t-invoke`, `c4t-analyse`), for use in scripting tester workflows that differ from that implemented in `c4t`. $C4_t$ also contains utilities like `c4t-stat`, which permits probing of C4's statistics logging; `c4t-backend`, which permits access to C4's underlying backends (such as Litmus); `c4t-config`, which assists with configuration; and so on.

3.3 Implementation

$C4_t$ consists of several separate *stages* that manipulate a JSON test plan: initial *planning* from an input corpus; *perturbance* to introduce random sampling and compiler configuration; *fuzzing* using $C4_t$; *lifting* of test-cases to compilable code using Litmus; and *invocation* (compilation and execution). This setup facilitates combining and rearranging parts of the testing workflow, as well as validating such parts in isolation. The `c4t` binary runs loops of a standard progression of stages, one per machine.

$C4_t$ makes heavy use of the observer pattern in its architecture: most tester stages can accept any number of observers that receive progress updates on stage progress. This lets us support rich progress reports such as the dashboard, as well as lightweight progress bars, verbose logging, statistics logging, and forwarding of progress information from the machine node to the main tester.

While $C4_t$ does not yet expose any of the tester logic as a public library, there are few barriers to us doing so in future.

4 VALIDATION

This section discusses the validation we have done so far for C4. This validation includes bug-finding campaigns, code coverage comparisons with existing tools, and initial work on mutation testing.

Bug finding. We have run $C4_t$, using $C4_f$ to produce fuzzed test-cases, on and off for the past two years. These bug-finding campaigns have used multi-machine $C4_t$ to target x86-64, 32-bit Arm8, and POWER9. We have discovered four bugs: two historic concurrency bugs in GCC 4.9; and two previously-unreported control-flow bugs: one in a prerelease version of GCC 11 (now fixed),[2] and one in IBM XL (fix pending as of Oct 2020).[3] By constantly folding any new features in $C4_f$ and $C4_t$ into the bug-finding campaign, we have quickly detected any regressions and semantic corner-cases.

Differential coverage. We have measured the differential code coverage that C4 achieves on a version of the LLVM compiler, versus both Csmith [26] (a leading C program generator that focuses on sequential C) and the LLVM test suite. Our results are that C4 achieves

[2] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97501

[3] https://bit.ly/XLcompilerbug

interesting deltas on coverage against both (1054 unique lines not covered by either Csmith or the test suite), and that the use of $C4_f$ increases this coverage. While the deltas are small in terms of the whole compiler, this is because concurrency support is inherently a small and highly focused part of the compiler. Indeed, the coverage hits code that implements (such as `AtomicExpandPass`, where we covered 151 lines not covered by Csmith and 63 lines not covered by the test suite) and optimises (such as `InstCombineAtomicRMW`, with 57 and 27 lines respectively) atomic actions.

Mutation testing. Recently, we have started using *mutation testing* [9] to validate C4. This lets us validate the ability for C4 to discover faults in the concurrency support of recent compilers without relying on the presence of such faults in practice, and to do so in a controlled manner. We have a fork of the LLVM 11 compiler (https://github.com/c4-project/mutated-llvm) that contains manually inserted branches, selectable at run-time, that induce faults in the compiler's concurrency support. Such mutations include swapping leading and trailing fences, omitting fence emission, and inverting bits in memory order comparison truth tables. Our initial experiments (across the same architectures as used in bug finding) show that C4 is reasonably able to detect such faults, but this is quite sensitive to the architecture on which tests are being run.

5 CONCLUSIONS AND FUTURE WORK

We have demonstrated C4, a tool exploring how real-world C compilers compile C11 atomic-action concurrency. We discussed the usage, audience, and implementation of C4 and its components ($C4_f$ and $C4_t$). We also outlined our validation work so far.

While C4 can already detect some forms of concurrency bug, more work on both C4 and its validation will provide a clearer effectiveness argument. We now outline some future work avenues.

For $C4_f$, we intend to add more actions: for instance, producing more inter-thread interactions that use atomic actions but do not affect observational semantics. We may also add support for C types other than boolean and integer primitives; explore other languages such as C++11 and OpenCL; and add test-case reduction support.

For $C4_t$, we intend to improve test throughput (by enhancing parallelism and scheduling, reducing network round-tripping, and so on). We would also like to improve statistics collection (perhaps by adding a SQL database to allow complex queries on past results).

While mutation testing gave promising metrics, we need a broader campaign for strong validation. Our manual, curated approach means that adding and justifying more mutants requires work.

We intend to spend more time validating C4 using existing bug reports. One approach would be to set up a experiment containing multiple compiler versions known to have particular concurrency bugs, and show that C4 can detect them in reasonable time. If not, we can use the bugs as stimuli for $C4_f$ development.

ACKNOWLEDGEMENTS

This work was supported by the EPSRC through grants EP/R006865/1, EP/R011605/1, and a grant from the VeTTS Research Institute.

REFERENCES

[1] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware. In *TACAS'11*. Springer, 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014). https://doi.org/10.1145/2627752

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[4] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *POPL'12*. ACM. https://doi.org/10.1145/2103621.2103717

[5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL'11*. ACM. https://doi.org/10.1145/1926385.1926394

[6] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *ESOP'14*. Springer-Verlag, 107–127. https://doi.org/10.1007/978-3-642-54833-8_7

[7] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating Optimizations of Concurrent C/C++ Programs. In *CGO'16*. https://doi.org/10.1145/2854038.2854051

[8] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36.

[9] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report HKUST-CS98-01. The Hong Kong University of Science and Technology. arXiv:2002.12543 [cs.SE]

[10] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. 2019. A snowballing literature study on test amplification. *J. Syst. Softw.* 157 (2019), 110398. https://doi.org/10.1016/j.jss.2019.110398

[11] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. In *OOPSLA'17*. https://doi.org/10.1145/3133917

[12] ISO/IEC 9899:2011 2011. *Information technology – Programming languages – C*. Standard. International Organization for Standardization. https://www.iso.org/standard/57853.html

[13] Bo Jiang, Xiaoyan Wang, W.K. Chan, T.H. Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. [n.d.]. CUDAsmith: A Fuzzer for CUDA Compilers. In *COMPSAC'20*. https://doi.org/10.1109/COMPSAC48688.2020.0-156

[14] Stephen Kell. 2017. Some Were Meant for C: The Endurance of an Unmanageable Language. In *Onward!'17*. ACM, 229–245. https://doi.org/10.1145/3133850.3133867

[15] Andrei Lascu, Matt Windsor, Alastair F. Donaldson, Tobias Grosser, and John Wickerson. 2021. Dreaming up Metamorphic Relations: Experiences from Three Fuzzer Tools. In *MET Workshop 2021* (Online). https://doi.org/10.1109/MET52542.2021.00017

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI'14*. ACM. https://doi.org/10.1145/2594291.2594334

[17] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *PLDI'20*. ACM. https://doi.org/10.1145/3385412.3386010

[18] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[19] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *PLDI'15*. ACM. https://doi.org/10.1145/2737924.2737986

[20] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *ASPLOS'17*. https://doi.org/10.1145/3037697.3037723

[21] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *PLDI'13*. ACM. https://doi.org/10.1145/2491956.2491967

[22] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *ESOP'20*. 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

[23] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In *ASPLOS'17*. https://doi.org/10.1145/3037697.3037719

[24] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-Memory Concurrency and Verified Compilation. In *POPL'11*. ACM. https://doi.org/10.1145/1926385.1926393

[25] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *POPL'17*. ACM, 190–204. https://doi.org/10.1145/3009837.3009838

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI'11*. ACM, 283–294. https://doi.org/10.1145/1993498.1993532